TOWARDS THE DEVELOPMENT OF AN AUTOMATED SHIP ARRANGEMENT DESIGN TOOL

CHRISTOPHER OLSEN

0-612-42419-7

Towards the Development of an Automated Ship Arrangement Design Tool

Christopher Olsen

B.Sc. (Mechanical Engineering)
B.A. (History)

A thesis submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of
Master of Engineering (Naval Architecture)

Faculty of Engineering and Applied Science
Memorial University of Newfoundland

September 1998

St. John's *Newfoundland* *Canada*

Abstract

This thesis reviews Naval Architecture methods emphasising the algorithmic and computer-based design of ships. It is shown that the problem of General Arrangements is critical to design synthesis and yet lacks the systemisation found in other ship design problems. Design systemisation improves the solution by reducing development periods and therefore costs, and by making more time available for additional design iterations. The thesis addresses the systemisation of the General Arrangement problem through the analogous Industrial Engineering problem of Facility Layout.

While conceptually useful, the algorithms for computer-aided Facility Layout are limited primarily by their crude and out-dated representation of spatial information. For this reason, the bulk of this thesis describes a novel formulation for spatial data, replacing the traditional 2D block layout model. Named Semi-Solids, the representation employs planar mathematics to manipulate and identically model 3D faceted surfaces. The name implies a variation of a solid model because the unique formulation allows the computer to shape and position spatial objects without the direct guidance or interpretation of a human user.

Microsoft's *Access* database software was used to create an efficient relational database for the storage of constraints and qualitative and quantitative data. Code for the manipulation of this data was developed using Microsoft's *Visual Basic*, and because *Visual Basic* and *Access* are closely related, data is easily shared by the database and the coded algorithm. In addition, it was possible to include a number of analytical functions specific to the database within the *Visual Basic* code. The database and the Semi-Solids code have been named *Ship Arrangement Tool* (ShipArrT) in preparation for additional work.

The thesis concludes with two detailed research plans showing necessary and potential areas for future research. The first plan completes the Semi-Solids representation and evaluates its potential relative to other Solid Model representations. The second plan offers ideas and direction towards the completion of a modern and robust Facility Layout/General Arrangement algorithm.

Acknowledgements

It is not often that one is given an opportunity to indulge one's curiosity and I count myself quite privileged to say that this has been my experience at Memorial University. Prof. D.A. Friis and Dr. A.M. Aboul-Azm bravely took me under their wings, and the work which follows is the result. Their enthusiastic support cannot be understated. In particular Prof. Friis has been most generous with his time and knowledge in the face of my creative distractions, obtuse questions and stubborn idealism.

I would like to thank the professors of the Faculty of Engineering for their assistance and patience, as well as Associate Dean J.J. Sharp and his kind helper Mrs. M. Crocker. Ms. I. Bulgin, who volunteered for the role of copy editor, also belongs in this group. In addition to these, and too many to mention, are the students, staff, and faculty members throughout the university who have championed my cause and been enormously helpful and supportive.

Financially, I would like to thank Dr. Sharp for several T/As and the dribs and drabs he has been able to send my way. My two-year stint as a Proctor in Paton College not only helped to pay the bills but was a tremendous learning experience, and Dr. Ian Jordaan should be

mentioned for providing me with a Research Engineer's salary for a period of this work. The Faculty of Engineering and Applied Science unexpectedly provided me with office space, and Prof. Friis with the assistance of Dr. Aboul-Azm was able to obtain for me a computer and software. In addition to all this, my primary source of funds has been the Ontario and Canada Student Loan programs, for which I will be grateful until the day the repayments begin!

Many of the papers I reviewed for this project appeared to have been published for the sake of publishing and not because they make a significant contribution to the literature. Their numbers are discouraging to the researcher and reduce the time he/she can spend on valuable papers. However, a small number of papers and texts were well-written and insightful, and it is many of the concepts they presented which inspired this project. Even after many readings, I find that the writings of this group still had something to contribute to my understanding of the design problem. These authors figure prominently in the Endnotes and I am grateful for their efforts, without which I would not have known where to begin.

Table of Contents

List of Figures

List of Tables

1

Context

The award-winning ferry[1] depicted in **Figure 1** and **Figure 2** was proposed in 1991 and is a departure from traditional RO-RO ferries. While it contains no recognizably novel features, the article describing the vessel which appeared in the Royal Institution of Naval Architect's journal *The Naval Architect* concludes with the assertion that the "design shows much thought and considerable vision of future sea transport and deserves serious study[2]."

What makes this vessel noteworthy relative to other new designs is that its designer, Hervé Folliott, is a graduate of London's Royal College of Art and is neither a trained naval architect nor marine engineer. Folliott is not alone as someone without a marine background being directly involved in the development of new ships. Interior and industrial designers, civilian architects, and engineers from almost all disciplines are regularly called upon to make significant contributions to the creation of modern vessels. *Le Corbusier* stands out because it is a design of merit which was developed without the input of naval architects. While still an unusual occurrence, Folliott's design may be a portent of a decline in the role of the Naval Architect in the conceptualization and over-all design of ships. It certainly begs the question of the origins of new designs.

1.1 Introduction

This project briefly revisits the ideas of the few naval architects who have published on the topic of design and uses this work to introduce a research program which attempts to identify and surmount key aspects of the design process which contribute to the narrowing focus of Naval Architecture. The inability to model spatial aspects of the design problem has limited the systemisation, automation and optimization of ship design. The improvement of computer modelling and the enhanced capability of CAD systems which results from the material presented herein are expected not only to reduce costs for owners and builders, but to enhance the process by which ships are developed, provide tools which may lead to greater understanding of the design process, address the concerns of the authors who have published on the decline of design, and ultimately, lead to the creation of superior ships.

The thesis is intended to lead towards the development of a computer-based automated Ship Arrangement design Tool, referred to hereafter as *ShipArrT*. To this end two presentations are made in this document. The first involving the material found in Chapter 1 presents a case for and examination of Facility Layout Algorithms for ship design. The second, beginning in Chapter 2 and occupying the subsequent five chapters, outlines a key step in *ShipArrT*, the development of an automated three-dimensional representation for ship layout design.

1.2 Design Theory

There are many representations for the ship design process **[3][4]** but the traditional model is iterative, and takes the form of a spiral such as that shown in **Figure 3**. It is a graphical representation of the steps in the design process, and because of its formulation, the figure emphasizes the interrelationship of the topics. The headings shown in **Figure 3** are common but additional topics or sectors may also be included in the model. The order by which each segment of the spiral is examined relative to the others is largely unimportant so long as no segment is neglected. As ship design is a creative process and thereby iterative, the spiral form indicates a progression towards an optimal solution as the number of iterations increases. A sub-optimal design will be achieved if any of the sectors is overlooked or does not yield a local optimal solution.

The design spiral model has been criticized by many authors such as D.K. Brown who believes that

> "Any design spiral is essentially a one-dimensional representation of design in which each topic is investigated in isolation and in turn. The reality is very different as each topic interacts with many others to a greater or lesser extent**[5]**."

Brown suggested that a superior model to the design spiral is an "interaction mesh**[6]**" such as that shown in **Figure 4**. However, in practice, the manner of analysis is iterative within sections of an interaction mesh — that is, for a particular ship length we select an engine and then we adjust the engine size and update the ship length and so on back and forth to improve the balance between the two parameters. When seeking solution consistency, the mesh elements and their interactions are almost impossible to standardize from naval architect to naval architect or even ship design to design.

Brown's position reflects that of J.P. Hope who believed that the "design engineer's experience and judgement of design parameters continue to be the dominant factors in design decisions**[7]**." Unfortunately, the availability and reliability of that experience is becoming questionable. The number of ships developed by a Naval Architect is declining as designs become increasingly standardized, require more detail and take longer to produce. Promotion has led to younger, less experienced managers who may have had little exposure to design disciplines outside their specialties and the increased use of CAD software has replaced the experienced draughtsmen who might have been able to advise the Naval Architects[8]. The *Le Corbusier* ferry suggests that naval architectural experience may not even be necessary in the development of new ships.

Instead of challenging the Profession's ability to generate and apply experience through its members S. Erichsen observed that

> "When we fail in design it is in most cases due to a lack of an overview
> or of a systematic approach and not so much due to lack of creativity.
> [The] first task in developing the discipline of design in naval architecture
> [is] to obtain a greater understanding of the need for a systematic
> approach and an increased use of systematic design methods[9]."

While the design spiral may be an imperfect representation of the design process, it is a useful algorithmic representation through which to discuss such methods and ultimately iteration and optimization in ship design. It also provides an important step towards the algorithmic methods required for computer-based design.

The lack of formal structure in the current design process creates three problems. First, because the design spiral method is essentially manual, it can be both slow and difficult to resolve design changes between particular topics. Using engine selection as an example, a single change can affect design parameters such as weight, volume, noise, vibration, speed, fuel consumption and tankage, etc., many of which are themselves interrelated. Second, because the order of topics in the design has not been specified, it is possible to neglect topics, or to introduce unresolvable conflicts between topics. Third, the unstructured environment gives the user freedom to vary the depth of his analysis from topic to topic. Thus an assumption may be used to deal with one aspect of the design, a heuristic for another, and a detailed analysis for a third. Ships being the sum of their parts, four conclusions can be drawn:

1:	The validity of the design is consistent with the validity of decisions which created it.	— *Correctness*
2:	The design is only as complete as the topics which were included in its development.	— *Omissions*
3:	The accuracy of the design model is limited by accuracy of its components.	— *Accuracy*
4:	The level of optimization in the design is limited by the level of optimization of each of its components.	— *Optimization*

This paper takes the idea of synthesis one step further by seeking consistency — that not only is every topic understood and reviewed, but also that the analysis is carried out to the same level for each design topic or sector of the spiral. In order for this to take place satisfactory models must exist for every design topic.

1.3 Design Applications

The relatively recent application of scientific methods to the design of ships, exacerbated by the introduction of the digital computer, has encouraged specialization within the profession related to each sector of the spiral. Unfortunately, the depth of study of specific topics has not been uniform, and as a result some topics have been neglected or passed off to other engineering disciplines. The complexity of ship design makes this a problem because each change made anywhere in the ship affects other areas of the ship, whether they are internal or external. Ironically, the specialization of areas of ship design at the expense of others may prove self-destructive for the Profession as recent computer advances have allowed the automation of some specialties. By way of examples, hydrostatics have for many years been analysed through reliable automated software, and recently automated structures programs have been published by a number of regulatory organizations including ABS and Germanischer Lloyd.

In his 1980 RINA paper *Creative Ship Design* **[10]** D.J. Andrews suggested that

> "... naval architects have taken the method of designing ships for granted ... [and they] have not given it the attention that the more specialized areas of marine technology have received ... because [they are] not readily amenable to engineering mathematics **[11]**."

To resolve this problem, Andrews proposed two steps towards "a more creative ship design process **[12]**." One begins with a discussion of design theory in which Andrews employed the term *synthesis* to describe the comprehensive aspects of the design of ships. Building on the concept of design theorist C.J. Jones who stated that "synthesis is putting the pieces together in a new way **[13]**", Andrews added that synthesis also demands an "appreciation of the totality of the newly created form **[14]**." He believed that through a "review of new general techniques and design theories that these could be used to produce an open and creative design philosophy able to serve the ship designer in the future **[15]**." The holistic definition of design synthesis promulgated by Andrews may have been his reaction to a profession which is increasingly oriented towards the trees instead of the forest (i.e., the mathematically-based specializations).

Yet it is important to recognize that while Andrews sought to draw upon work originating from more artistic roots, he was still advocating a systematic engineering-style approach.

Andrews was particularly interested in preliminary warship design and suggested that modern approaches to the problem left decisions regarding the new vessel's General Arrangement to a point too late in the design process. Because spatial constraints prescribe the principle dimensions of the vessel and vice versa, Andrews believed that some sort of algorithm was required which would make spatial requirements part of the initial sizing of the ship. This led to his other proposal in *Creative Ship Design* which involved the application of Computer Aided Architectural Design (CAAD) models to current ship design software so as to make possible an exploration of "significant changes to ship internal layout and hence the total ship form**[16]**." The addition of CAAD models was an attempt to bring the computer, and ultimately the designer, closer to Andrews' concept of *synthesis* since the designer could explore in detail options previously studied only superficially if at all. Therefore Andrews proposed first to mathematise the empirical and hence neglected topic of General Arrangement and then to seek contexts and processes through which to encourage naval architects to focus less upon subsets and more upon the general design problem.

> "The urgent question for the profession, with ever increasing demands for understanding of the intricacies of the engineering components, is how we foster the task of integration and the architectural task of coordinating the design development. The only positive development I see in this regard is the growing capability of computers, as true aids to the designer rather than just powerful analytical tools; however, if they are to become real aids then the designers must direct their application to

> the architectural aspects. Thus I see my proposal to incorporate layout considerations in the earliest stage of the technical design, as not just a worthwhile development but an essential step towards naval architecture regaining its primacy in ship design**[17]**."

Andrews, in advocating ship synthesis, sought to ensure that each design was fully understood by each designer such that a single designer controlled the entire design process. In his comments to Andrews, Fuller agreed and stated, "Our profession must go down the track where you can comprehend the whole ship, its requirements, and its external relationships ...**[18]**" While broad or 'synthetic' approaches do not address the level of analysis, they are more likely to ensure that the intricate relationships of different parts of the design are recognized and accounted for.

When Andrews published *Creative Ship Design* almost 20 years ago he felt that the two steps of his thesis were necessary in the development of better ships. His concern regarding the mathematisation profession echoes the historic debate over art and science, or more specifically, architecture and engineering. Several authors have expressed concern that Naval Architecture was giving way towards ship engineering**[19][20]**. Andrews feared that, by considering only those topics amenable to mathematics, naval architects would ignore or approach haphazardly other topics which impact the overall success of the design. Speaking to the need to systematise layout design, R. Baker observed

> "Mathematics, or the ability to solve the technical aspects, gave a great boost to [the respect of a customer for the integrity and competence of the designer]. (The customer no longer has to worry as to whether the ship would sink, capsize, break up, stop, or not steer). Unfortunately, the

> success of this element on the prestige of the designer tended to obscure the importance of arrangement[21]."

At the same time, Baker also noted the importance of the layout to the overall success of the design and its effect on the reputation of the Naval Architect:

> "... if the layout fails (that is, not liked) factorial N complaints will propagate, for the customer or his agents have to live with the arrangement day in and day out, perhaps for years, and if they so live, even making do, a failure in this field is bound, at least, to erode respect and destroy all confidence, whereas an ultimate technical failure, even if terminal, is only an episode[22]."

The systemisation of the ship design problem, including its sub-problems, becomes an issue of credibility, with the potential of adversely affecting the position and prestige of the profession in the eyes of the maritime community. Therefore the future employability of the naval architect is now a function of the demonstrable application of mathematics and scientific methods to *all* aspects of design, including those topics which have been previously neglected.

Ultimately the goal of the ship design process is to develop better ships by optimizing every topic in the Spiral, both relative to the constraints of the particular design area and relative to the constraints imposed by other areas of the Spiral. Optimization in design requires iteration, but iteration can be enormously time consuming and has a diminishing value of return. In practice, time constraints limit the number of iterations to as few as one, and likely do not allow a full exploration of the problem since "few designers can manipulate more than three variables simultaneously with some six more in a 'quick recall memory' which can quickly be brought into play[23]." Not only are computers far more capable of coping with broad and complex multi-variable problems, but continuing advances in computer aided design "has enabled detail to be handled much earlier in the design process" thereby providing the designer with more information about the overall problem. In turn, this detail has led to a blurring of the line between Preliminary, Conceptual and Detail design as the same model is simply fine-tuned over the course of the project**[24][25]**. The most recent trend is the development of the virtual ship through the application of 3D Product Modelling in the US Navy. Based on CAD/CAM software, a 3D product model

> "contains not only 3D geometry, but also associative and parametric relationships, as well as other non-geometric information. [It] provides technical and logistical data necessary to describe and support a complete ship design [and] serves as the main information vehicle for ship design and production information, as well as the integrator for logistics and other life-cycle data**[26]**."

Essentially a shared data format, the 3D product model contains all data associated with the ship and provides a number of tools by which that data can be altered, viewed and managed by one or more users. **Figure 5[27]** shows the significant cost savings potential of CAD and virtual design. In addition, computer aided design facilitates concurrent engineering with benefits suggested in **Table 1[28]**.

Since the designer remains limited to the manipulation of a few variables, the advance offered by the computer lies in automation. Over the years, software automation has made possible graphical interfaces, input/output control, file management, a wide range of software applications, etc. The key to the successes which have been achieved stem from the ability to discretise problems sufficiently that each discrete step can be solved correctly and consistently and that the movement from one automated step to another can also be carried out correctly and consistently. Modern programs are now sufficiently complex that they are developed by teams of programmers working on specific modules of automated code. Although software becomes more complex all the time, the exponential improvements in computer hardware obviate the additional computation required. Despite the complexity of the ship design problem, advances in automation and the increasing capability of software led Andrews to write that "the momentum behind developments in preliminary [Computer Aided Ship Design] CASD to simplify the initial design 'synthesis' is no longer necessary or desirable**[29]**." Building on this idea, the author proposes naval architects should pursue more robust and sophisticated models, trusting to automated algorithmic methods to deal with details, just as one might trust software to display a graph without direct input or action.

Perhaps L.J. Rydill was on the right track when he asked, "With all the computer aids now available earlier — one has capabilities that were not previously available — how can they be exploited to improve the design process, as opposed to just improving the facility with which it is carried out**[30]**?"

1.4 General Arrangements and Facility Layout Problems

Andrews, in his discussion of synthesis, recognized that the universal problem can only be tackled once the critical General Arrangement sub-problem has been satisfactorily modelled and automated. To date layout problems have been poorly if at all modelled using the computer, either by the marine community or otherwise. Currently, General Arrangement problems are solved manually and instinctively. Computer use for General Arrangements is almost always representational in the form of a CAD drawing. The development of software which can automatically arrange objects with spatial definitions and generate such drawings would be an important step towards improving the process of design.

Layout problems are perhaps the most difficult problems to solve with the aid of a computer because they are spatially based as opposed to numerically based. The key difficulty lies in the representation and manipulation of spatial entities. Humans are quite adept in determining the solution of spatial problems but lag far behind the computer in coping with numbers and quickly evaluating new spatial configurations — a difference between implicit and explicit in that humans attribute meaning to spatial objects beyond the mathematical data required for their representation in the computer. It is for this reason that most design aids involve a user working interactively with the computer such that the human manipulates the spatial objects relative to one another, and the computer stores and evaluates the result. Unfortunately, a truly optimal solution requires an enormous range of configurations to be created and evaluated and for this to take place some sort of computerization of the spatial aspects of the problem must take place.

Based on barren literature and modern education curricula, Naval Architects appear to be uninterested in the architectural aspects of their problem, much less in finding systematic approaches for architectural design. In contrast, Industrial Engineers have made significant progress towards the development of algorithms for what they termed the Facility Layout Problem. Although material has existed for many years, it was not until the 1950s and later that progress appears to have been made towards the systemisation of the layout process **[31]**. A number of computer-based algorithms such as CRAFT and ALDEP built on this work in the early 1960s and made Facility Layout Problem solvers some of the very early computer applications. The Facility Layout Problem is data intensive as well as having a spatial component, and algorithms and subsequent computerization were developed as tools through which such data could be managed more effectively.

The Industrial Engineers considered the Facility Layout Problem to be an extension of their own work in the area of manufacturing in which a common problem was the balancing and optimization of assembly lines. They observed that labour and handling were significant per-item costs, and from this it was recognized that a relatively successful layout for a manufacturing facility is one in which the cost of transporting a product from work space to work space is minimized, generally achieved by minimizing the distances between departments. In addition, the Industrial Engineers recognized that the computer could be used for the arrangement of departments, and that they could quickly generate a score for the layout from the work-cell-to-work-cell distances, thereby providing a means for the comparison of different layouts. Even the terminology used for spatial layout has been developed along manufacturing lines such that the jargon refers to any region of a layout as a *department*. However, since many layout problems are not concerned with the efficient flow of materials through an assembly line, for the purposes of this project the generic term *space* will often be used to denote a room, area, department or work cell.

Despite many years of work, the Industrial Engineers have had little large-scale success with their algorithms. The limitations of the computers of the day forced them to use heuristics and crude models, and the resulting solutions were often found to be unsatisfactory and/or sub-optimal. Although computerized layout algorithms are still used, their application and acceptance is still limited and the majority of such problems are still solved manually through the designer's insight and intuition. The difficulty appears to be that, in principle, modern algorithms remain almost identical to the crude models developed in the 1960s.

The formulation of FLP's can be reduced to a simple process:

Select a placement or exchange
Perform the placement or exchange
Score the new arrangement
Compare the score with that of previous iterations

where scoring is performed by taking information from the layout, usually the distance between spaces, and multiplying it by some weighting value.

Data and constraints in FLP's can be loosely divided into two classes: distance-based and spatially-based. The two groups are distinguished by their means of evaluation and manipulation. Distance-based constraints lend themselves to be measured against a common scale such as cost and can be evaluated through simple summation. Spatial constraints are better modelled by inference engines such as those found in expert systems, since they require a decision to be made as to the case-specific importance of each constraint or piece of data. Further, spatial constraints are not easily defined and may be qualitative instead of quantitative which suggests that Uncertainty Theory might also play a role in the manipulation of this group of information. A. Cort and W. Hills pursued this concept with regard to Naval Architecture by discussing fuzzy sets in their paper *Space Layout Design using Computer Assisted Methods*[**32**].

The following list of potential distance-based constraints ignores the size and shape of the particular room or space as well as any spatial restrictions; it is instead concerned only with the relationships between a room and its neighbours.

The constraints in **Table 2** can be reduced to functions based on distance, and all encourage or discourage the proximity of one space to another. By use of multipliers distances can be treated as costs giving a measure of significance to each of the parameters. In essence, cost becomes a common denominator for each of the constraints, with the constraints acting as springs, drawing spaces closer together or pushing them further apart. In more generalized terminology, cost is used as a weighting function and serves to emphasize one constraint over another. In addition to distance-based constraints, there are a number of practical constraints which are not functions of distance as the items in **Table 3** suggest.

There are significant differences in the manipulation of distance-based and spatially-based constraints. Distance-based constraints are well suited to computerization since they essentially require the computation of a sum. This is quite unlike spatial constraints which generally require a decision process to determine which constraint takes precedence and which might be neglected for a particular layout. Unfortunately, it is difficult to formulate decisions regarding spatial constraints. For example, is area more important than the dimensions of length and width? The coordination of constraints is a knowledge-based problem and in the final chapter is proposed as an area of future work.

The chief difficulty faced by FLP algorithms lies in bringing together spatial information and numerical information such as the distance measurement suggested for the constraints in **Table 2**. To address this problem, typical Facility Layout algorithms employ a number of assumptions which allow them to employ a grid of uniform 2D blocks. This reduces the spatial problem to one which is binary. Conceptually, the algorithms are not significantly dissimilar to a child's word scramble game (**Figure 6**). By placing uniform blocks into a matching uniform grid, the Industrial Engineers were able to create an environment in which the computer could, with relative ease, find its way around the spatial aspects of the problem. Unfortunately, this approach fails to adequately model either the distance-based numerical constraints and data, or the spatially-based and often qualitative constraints and data. However, it does lend itself to solution by means of the well-studied Quadratic Assignment formulation.

1.5 *Quadratic Assignment Problems*

By far the most common algorithm for solving FLP's is the mathematically explicit Quadratic Assignment Problem (QAP) formulation. The QAP assumes that spaces can be represented as standard blocks, and that the design space into which the blocks will be inserted can be discretised into corresponding slots for these blocks. Mathematically the blocks can be described as the set $M=\{1, ..., m\}$ of m equally-sized units, and their potential locations as the set $N = \{1, ..., n\}$ of $n \geq m$ areas, each of which can house at most one unit. To address the distances between the blocks a *distance matrix* $A = \{a_{ij}\}$ is required. Finally, a *connection matrix* $B = \{b_{ij}\}$ completes the formulation and represents the weighting functions for the various scores between pairs of spaces. Then,

> "let the $m \times n$ decision variables x_{is}, $i \in M$, $s \in N$ be defined as: $x_{is} = 1$ if unit i is located at area s; otherwise, $x_{is} = 0$. If a pair $\{i, j\}$ of units are assigned to areas $\{s, t\}$, respectively, then the contribution to the objective function is $b_{ij}a_{st}$ which, with the decision variables introduced, can be expressed by the *quadratic* term $x_{is}x_{jt}b_{ij}a_{st}$. A 0-1 programming problem formulation of QAP is then[33] **Equation 1**:"

$$\min z = \sum_{i \in M} \sum_{j \in M} \sum_{s \in N} \sum_{t \in N} x_{is}x_{jt}b_{ij}a_{st}$$

Equation 1 Formula for the solution of the Quadratic Assignment Problem.

The solution of the QAP requires the time-consuming evaluation of every combination of blocks in the layout as suggested by **Table 4**. As a result, heuristics can be applied to facilitate the solution of the QAP through additional assumptions and by distinguishing between *construction* and *improvement*. Although the constraining data required by both classes is the same, they differ in start point and can also differ in their treatment of rules of simplicity, contiguity and utilization[34]. Construction algorithms such as that in **Figure 7** are used to create or *construct* layouts by placing the spaces into the design space in some optimal arrangement. Improvement algorithms (**Figure 8**) generally begin with an existing layout, either user defined or the product of a construction algorithm, and seek to improve it through the exchange of spaces. Because spaces often differ in area, during a guess, exchanges may be tolerated which violate one or more constraining rules.

The formulation of the QAP assumes a standard block size which is used for each space, regardless of the size of the required space. This in turn creates problems when the time comes to perform the layout with dimensionally correct spaces since the variety of sizes may affect the relative positions of the spaces. The distance matrix contains measurements of the distances from one slot of the solution grid to another. However, the distances are not necessarily correct because the methods of distance measurement may not be appropriate for the particular scenario. As shown in **Figure 9** these might include Euclidian or rectilinear measures originating from different points on the object such as a centroid or an edge. The proximity of spaces is encouraged by the impact of the weighting values found in the connectivity matrix on the overall score of the layout.

A generalized Facility Layout algorithm takes in user data and user preferences in the form of weighting functions, and is able to indicate the superiority of one layout over another. Ironically, research in Facility Layout has focused on the decision processes involved in the problem, and not on the model itself. To the author's knowledge, no attempt has been made to address the limitations of block layouts, nor to develop an alternative representation format. If one is prepared to neglect the problem of 'fit' for a moment and examine the configuration, a crude layout can be created simply on the basis of the relationships between cells or spaces**[35]** as shown in **Figure 10**. Essentially the layout problem can been solved without ever having to address the physical constraint of 'fit'. This is reminiscent of the computer generation of Pert Diagrams with the pitfalls shown in **Figure 11**.

1.6 Block Layouts and Placement

Spatial constraints can be added to the QAP formulation through the utilization of smaller blocks. In such a formulation, a user would choose a block size which could be used as a common denominator for all of the spaces in the layout. Then an appropriate number of blocks would be allocated to represent the floor area of each space. To address the problem of *homogeneity* — the need to keep the blocks which define a space adjacent to one another — a very high weighting value in the connectivity matrix is used. While this elegantly introduces spatial considerations to the QAP formulation, in practice it only crudely models the spatial problem. This can be demonstrated by testing the effectiveness of the block layout formulation in addressing the constraints in **Table 3** as summarized in **Table 5**

First, block layout assumes that the *size* of a space is fixed. However, the reality is that there is often a range of acceptable sizes. A bedroom would be a good example with a minimum, preferred and maximum size and an acceptable solution lying somewhere in this range. Also, block layout does not offer any means by which the *orientation* of a rectangular space can be prescribed where a long and narrow space is required. The examples in **Figure 12** illustrate these concerns by depicting some of the odd configurations which can result from manipulating block layouts**[36]**.

As previously discussed, *homogeneity* can be ensured by means of high score weighting between the blocks of a particular space. *Simplicity* and *contiguity* are encouraged by the same rules, but block layout can lead to instances such as those in ?. *Consistency* can also be forced by means of the high internal scoring weights, but this can adversely affect acceptable variations in shape/aspect ratio.

The complete *Utilization* of the layout region is ensured by the formulation's explicit definition of each of the blocks in the design space. Block layout does not lend itself to specific control of *access* details such as doorways and windows, nor can it cope with elements which could be *shared* in some configurations and independent in others. Finally, *accessibility* can be only approximated by the block layout formulation. Two approaches find application in these instances but each has disadvantages. First, including a corridor allowance in the area required for each space effectively removes the problem of *accessibility* from the formulation. At the same time, however, it can lead to configurations in which the position of the corridor is impractical or inefficient. In the second approach, a corridor can be defined as a separate and additional space with a high adjacency value. However, neither is there a means by which corridors can be defined which vary in size depending on traffic flow, nor can the size of transportable objects be modelled. There is also no facility through which corridors for two neighbouring spaces can be shared, thereby taking up less floor area in the layout.

Despite all of these problems and limitations, block formulations persist as the most common spatial representation found in Facility Layout problems. While the reasons may vary, the simplicity of the depiction and the underlying mathematics has great appeal when no obvious alternative exists.

"The problem of developing a layout planning decision aid appears to be this: a representation that is convenient for display and for mechanizing the drafting process is not well suited for the designer's purposes or for design algorithms. Conversely, a representation that is convenient for algorithmic manipulation is not well suited to display and drafting operations**[37]**."

1.7 *FLP Algorithms and Naval Architecture*

The field of Naval Architecture presents a unique problem for traditional Facility Layout designers. Moving beyond the spatial problem described in the previous section, a ship's General Arrangement calls for an integrated approach for aspects of the problem because of the unique shapes and problem details involved.

Generally a vessel's hull can be used to define a region for acceptable placement — spaces cannot be placed outside the hull, nor outside a prism which extends upward from the deck line. In the area within the hull it is desirable to fill the entire region — void space is wasted space. Above the hull, one of two situations can occur: either the layout will drive the sides of the superstructure to the boundary as might be the case in a bounded construction algorithm; or the layout will take place freely within the prism as might be the case for unbounded placement. This makes superstructure design a hybrid of bounded and unbounded construction (**Figure 14**) methods with their associated constraints. Further, it is desirable to allow variation of the layout during the improvement algorithm. That is, in instances where an unbounded superstructure has been created, improvement algorithms should be able to alter the shape of the superstructure as it exchanges spaces.

Neglect the superstructure for a moment and take the problem of arranging spaces within the hull as an example. If one were to take a slice of the hull similar to a waterline to use as a 2D design space one must first determine the elevation of the slice above the keel to achieve the correct deck heights. This is a difficult task without first examining the hull contents for their vertical dimensions and the potential for multiple decks. In addition, one is also faced with the problem of placing rectangular blocks against a curved boundary/design space regardless of the slice. The obvious solution would be to use smaller blocks so that the curved boundary can be better approximated; however, from the point of view of computational efficiency, more blocks require more computation for evaluation, alteration and scoring. Also, the exchange of small blocks may have only a negligible or even unevaluable effect on the score of the layout. It is also possible that the block exchange impacts the layout like a step function. For example, if one thinks of a parabolic objective function then the exchange of a pair of blocks could hop from one arm to the other without bringing the solution closer to the optimum. Problems such as those described above will be difficult to overcome given current algorithms.

Is a 2D approach reasonable? The hull form is actually a surface which curves in three dimensions and areas within the ship almost always conform to these curved surfaces. One need only examine the interior of a sailing yacht to see how much the shape of objects contained within the hull are affected by the hull/boundary. In order to address these characteristics a 3D design space comprised of small cubes may be considered. To reduce the number of cubes requiring examination it may be desirable to use polyhedrons which are the height of a t'ween deck space. However, while simplifying the problem in one respect, the contents of many spaces need not necessarily rest on a flat floor. By way of example, the placement of a desk against a canted wall may be considered quite successful despite the possibility that it is either overhung or undercut by the wall. Flat decks are also a crude assumption because ships commonly have camber and sheer. It is also common to find decreased head room in some areas of the ship even though the area may be on the same deck as a taller space. Each of these problems is difficult to model without a still further increase in block resolution — although one might argue that camber and sheer can be accommodated by using a measurement coordinate system which alters the height of the blocks for particular X and Y (length & beam) coordinates.

For example, if one sought to design the interior arrangement of a large cruise ship one might be dealing with a design space of 260m x 32m x 50m. Taking this to be rectangular for a moment and using a 1m-sided cube as a spatial unit one finds that one is dealing with 416,000 cubes. And this assumes that all spaces in the interior of the vessel are divisible by 1m. A more reasonable resolution would be litres instead of cubic metres, but this increases the number of cubes to 416 million. Even by using a block which is 10cm by 10cm by 2m, the quantity of polyhedrons to be solved is still impractically high.

Unlike many land-based layout problems, Naval Architecture requires the consideration of a number of constraints including the location of weight and the ship's stability. Also, a number of spaces must be placed in particular areas of the ship regardless of the efficiency values suggested by the scoring engine of a layout algorithm. To illustrate this point, consider the location of mooring winches and other equipment. The complexity of the layout is important because not only are services such as electricity used throughout the vessel, but the generation of that electricity must also be accounted for. Further, in many instances it may be more effective to distribute HVAC equipment throughout a cruise ship rather than distribute these services from a single central location.

In a subsequent paper to *Creative Ship Design*[38] called *An Integrated Approach to Ship Synthesis*[39] which appeared in 1982, Andrews proposed a computer-based algorithm not entirely dissimilar to the Quadratic Assignment algorithm. Unfortunately, Andrews was more interested in solving the Synthesis problem than the General Arrangement sub-problem and was unable to automate or adequately develop his layout algorithm. The poor results he achieved with his Synthesis algorithm could be attributed to his inability to effectively cope with the spatial problem — ironically because the goal of his work was to "incorporate a fuller design description in the initial synthesis of a new ship design through concurrent consideration of spatial disposition[40]." Andrews feared black box solutions, and the resulting layout algorithm called for the interactive and unsystematic manipulation of spaces which the computer would then score. The scores would then be used to update the remaining, automated, design modules of his Synthesis algorithm. Scoring took place on the basis of circulation densities (a measure of adjacency based on the traffic between different spaces) as the measurable quantity for the relative positioning of spaces within the layout.

Despite concentrating on just the General Arrangement problem, other authors remain trapped in an examination of scoring scoring and not representation and automation. J.P. Hope's paper, *The Process of Naval Ship General Arrangement Design and Analysis*[41] proposes several scoring principles and demonstrates a manual algorithm for their implementation. Similarly Cort and Hills, while concentrating their efforts on the application of Fuzzy Sets when they published *Space Layout Design Using Computer Assisted Methods*[42], finished with a representation and algorithm not dissimilar to the manual one used by Andrews.

For the purpose of Naval Architecture, an automated 3D representation would be desirable since it would be better able to model the unusual shapes and surfaces common to ships. In addition, any new formulation must be automated so that the process of General Arrangement design can advance beyond the stage of calculators and electronic drafting boards.

Figures Pertaining to Chapter 1

Figure 1 Model of *Le Corbusier* — a proposed RO-RO ferry design.

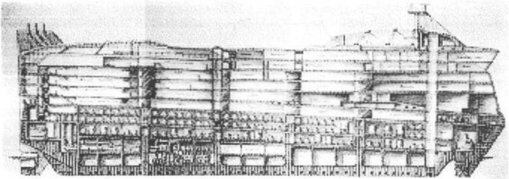


Figure 2 Cross-section view of *Le Corbusier* showing the ferry's General Arrangement.

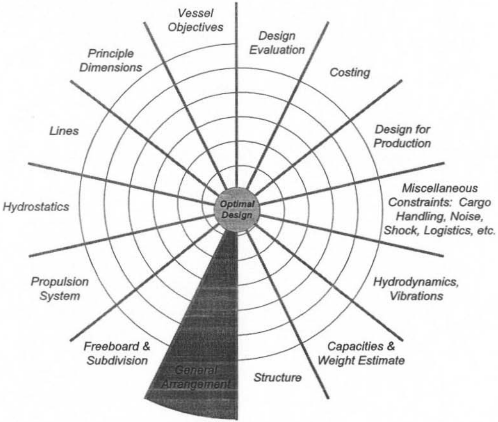


Figure 3 An example of a Design Spiral. The General Arrangement problem is shown in grey to denote its limited computerization.

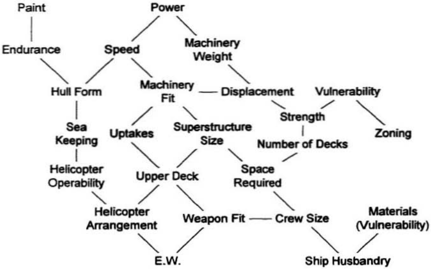


Figure 4 A depiction of an interaction mesh, very much like that originally proposed by D.K. Brown in *Naval Architecture*[6].

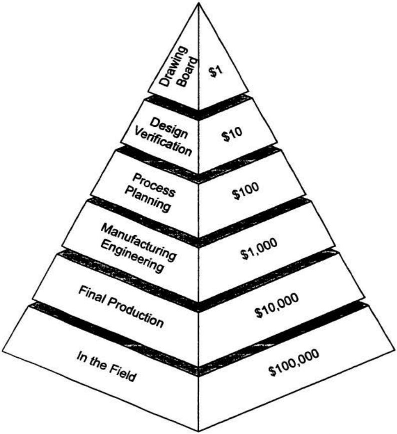


Figure 5 Cost Pyramid showing that small expenditures early in the design process can lead to enormous savings at subsequent stages.

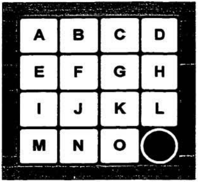


Figure 6 A child's word scramble game is analogous to the 2D Block Layout approach used by Industrial Engineers to solve Facility Layout problems.

```
GET DATA
REPEAT
    SELECT a new seed Space
    PLACE the selected Space in the layout
    FOR i = 1 to number of spaces
        SELECT a Space not yet placed
        PLACE the selected Space in the layout
    NEXT i
    SCORE layout
    IF Score < PreviousScore THEN
        PreviousScore = Score
    ENDIF
UNTIL the number of iterations is sufficient
```

Figure 7 Pseudocode for a construction algorithm. Note that *sufficient* can be a user-defined preferential value.

```
GET DATA
REPEAT
    CHOOSE a pair of activities
    ESTIMATE the effect of exchanging them
    EXCHANGE if the effect is to reduce cost
    CHECK to be sure that the new layout is better
UNTIL no more improvements are possible
```

Figure 8 Pseudocode for an improvement algorithm.

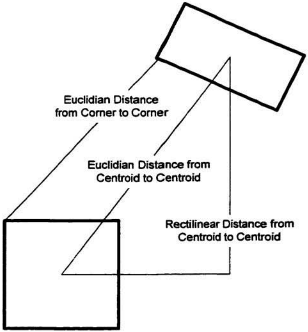


Figure 9 Examples of distance measurements.

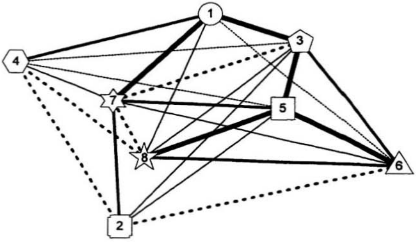


Figure 10 A graphical depiction of the creation of layouts on the basis of distance relationships between spaces. Five different weighting values (shown with five different line types) were used with an arbitrary distance unit to create this figure.

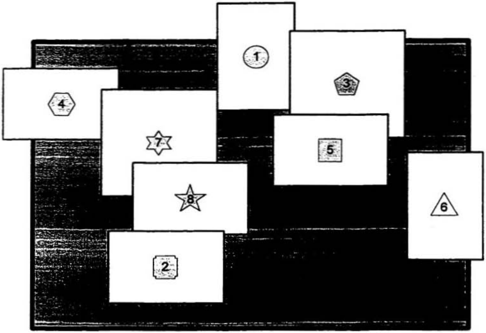


Figure 11 While layouts can be created on the basis of the positions of centroids, the addition of spatial information may make such solutions invalid. Here, not only do spaces overlap and have unnecessary void regions, but some spaces violate the exterior boundary of the design region.

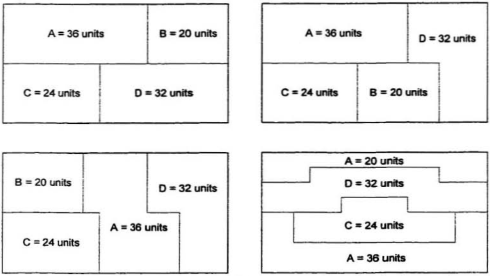
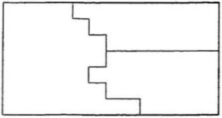


Figure 12 A series of images showing various block layout configurations for the same layout problem.



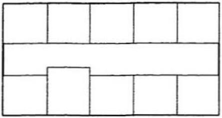
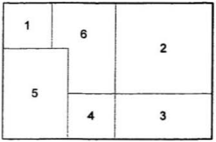


Figure 13 Simplicity and contiguity problems in block layouts. The example on the left shows the jagged edge which can result from the algorithm's desire to place a boundary through the middle of a grid unit. On the right is a corridor in which one of the spaces violates a contiguity rule and thereby ruins a clean wall line.



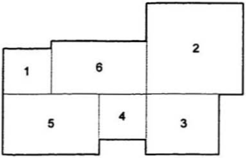


Figure 14 Bounded vs. unbounded placement. The figure to the left shows how the addition of a boundary constraint affects the shape and position of several spaces. Compare this to the same spaces in their 'natural' configuration in the unbounded example on the right.

Tables Pertaining to Chapter 1

Concurrent Engineering Benefits	
Development Time	30 - 70 % reduction
Engineering Changes	65 - 90 % reduction
Time to Market	20 - 90 % reduction
Overall Quality	200 - 600 % improvement
Productivity	20 - 110 % improvement
Dollar Sales	5 - 50 % improvement
Return on Asset	20 - 120 % improvement

Table 1 Concurrent Engineering Benefits accrue through multiple users. Software developed for Simulation-Based Design offers this potential.

Constraint	Description
Weight	*room weight is relevant for large buildings and ships*
Traffic	*frequency of people/goods entering and departing*
Vibration and Noise	*vibration or noise created in a room, or the tolerance of a room for vibration and noise*
Services	*electricity, water, sewage, etc.*
Thermal Insulation	*level of, or importance of, insulation for heat or cold from one region to another*
Construction Cost	*cost to assemble and install*
Operating Cost	*cost of maintenance and upkeep*
Access (corridors, stairwells)	*requirements for people and goods beyond the room*
Proximity to exterior	*need for external access*
Adjacency to other spaces	*need to share a wall with another room*
Proximity to other spaces	*need to be close to or far from another room*
Sharing of common spaces	*corridors, washrooms, entrances, etc.*

Table 2 Examples of distance-based layout constraints.

Constraint	Description
Size	*size of a space is not necessarily fixed*
Orientation	*orientation relative to other spaces or the boundary*
Aspect Ratio	*shape of a space is likely bounded*
Homogeneity	*a space is not divided into several pieces*
Simplicity	*few corners or jagged edges*
Contiguity	*one wall leads into another on the next space*
Consistency	*similar spaces resemble one another*
Utilization	*no voids, and adherence to fixed structures and boundaries*
Sharing	*efficiency of common spaces such as corridors, washrooms, entrances, etc.*
Accessibility	*corridors, stairwells*
Access	*location of doors, etc.*

Table 3 Examples of spatially-based layout constraints.

Units located at areas				Distances associated with unit pairs					Sum of Connectivity * Distance
1	*2*	*3*	*4*	*(a, b)*	*(a, c)*	*(a, d)*	*(b, c)*	*(c, d)*	
a	b	c	d	1	2	4	1	2	56
a	b	d	c	1	4	2	3	2	78
a	c	b	d	2	1	4	1	3	60
a	c	d	b	4	1	2	3	1	70
a	d	b	c	2	4	1	2	3	79
a	d	c	b	4	2	1	2	1	67
b	***a***	***c***	***d***	***1***	***1***	***3***	***2***	***2***	***51***
b	a	d	c	1	3	1	4	2	73
.				.	.	.	.	.	.
.				.	.	.	.	.	.
.				.	.	.	.	.	.
d	c	a	b	2	1	2	3	1	56
d	c	b	a	2	3	4	1	1	66
Connection between pairs				7	8	4	7	5	

Table 4 This example shows how the QAP formulation is used to determine an optimal solution given four units *a*, *b*, *c* and *d* located at four locations *1*, *2*, *3* and *4*. Connectivity, or weighting values, are shown in the last row of the table.

Constraint	*Description*	*Block Layout*
Size	*the size of a space is not necessarily fixed*	No variance
Orientation	*orientation relative to other spaces or the boundary*	No control
Aspect Ratio	*the shape of a space is likely bounded*	Limited variance
Homogeneity	*a space is not divided into several pieces*	Yes
Simplicity	*few corners or jagged edges*	To an extent
Contiguity	*one wall leads into another on the next space*	To an extent
Consistency	*similar spaces resemble one another*	In size but not necessarily in shape
Utilization	*no voids, and adherence to fixed structures and boundaries*	Yes
Sharing	*efficiency of common spaces such as corridors, washrooms, entrances, etc.*	No
Accessibility	*corridors, stairwells*	Can be assumed part of each space or forced as an additional space
Access	*location of doors etc.*	No

Table 5 Evaluation of the manipulation and representation characteristics of the Block Layout formulation using criteria from **Table 3**.

2

ShipArrT

Research for this project began as a master's degree investigation into computer-aided ship design, and focused in particular on the use of knowledge-based (expert) systems. Andrews among others recognized that these tools could be useful for ship design and several attempts at the development of such programs appeared in the literature as this work began**[43][44]**. During the literature search it became clear that the successful application of knowledge-based systems was easier said than done. Since knowledge-based systems are best suited for the balancing of relatively few, closely related constraints, the limited success of such systems is primarily attributable to the quantity and domain of data involved in the problem. Design Spiral solutions are particularly difficult to model because of the wide range of relatively unconnected data required.

Having expanded the search parameters, it became evident that the tools for ship design, whether knowledge-based or otherwise, focused on the derivation of principal dimensions and characteristics and that no algorithm attempted to study the problem of design from a functionality viewpoint. Thus, a designer is forced to work interactively with CAD software to manually generate a vessel's General Arrangement for each iteration of the design spiral.

Exacerbating this problem is the great number of cases in which the General Arrangement drives the external design parameters.

An algorithm which can generate a reasonable design of the interior layout of a ship would serve to ensure not only that no element of the General Arrangement is omitted but also reduce the time and effort required of the naval architect. Used in conjunction with a parametric or knowledge-based optimization method, a Facility Layout Algorithm would make possible the creation of a relatively complete preliminary design in a very short period of time. Hence, for a given period, either additional iterations of the design spiral can be completed thereby creating potential for superior designs, or, a greater number of preliminary designs can be generated. Each design can also be developed in more detail due to automation of an increasing number of design tasks, again creating the potential for superior designs.

2.1 A New Facility Layout Algorithm

A new Facility Layout Algorithm, in order to bring about superior solutions to those of its predecessors, must begin by replacing the heuristic block spatial representation. This leads to a more complex algorithm and requires the use and management of significantly greater quantities of data. The increased level of detail will make possible studies of routing and corridors previously carried out either manually or crudely modelled through the use of heuristics. Previous work in the field identifies algorithmic steps and suggests a systematic approach to the problem. In particular, scoring methods and many of the decision processes which have been well-studied over the years can be easily expanded upon and employed by a new layout algorithm. <u>**Table 5**</u> shows the steps of a new Facility Layout Algorithm, and suggests the types of computational tools and approaches required for each step. The modules are intended to operate as separate entities drawing from and contributing to a central database as shown in Figure 15.

It was decided to name the project Ship Arrangement Tool, or *ShipArrT*, so as to reflect the emphasis which has been placed on Naval Architectural problems. From the table five development tasks for *ShipArrT* become apparent:

- a *Relational Database* through which problem data can be tracked and easily manipulated
- an *Expert/Knowledge-based System* for the manipulation of conflicting constraints
- a *Solid Modeler* for spatial representation
- an algorithm for the solution of the problem of *Routing* and superposition
- an *Expert/Knowledge-based System* and/or a number of optimization algorithms (Simulated Annealing, Genetic Programming, Dynamic Programming, etc.) for updating and facilitating the algorithm's decision engine

The critical step in this list is the development of a sophisticated and robust representation for the spatial aspects of the FLP. For this reason this project has explored the development of an alternative to the block formulation called Semi-Solids. The formulation stresses the representation and automated manipulation of 3D spatial objects, and is expected to address critical shortcomings of traditional Facility Layout formulations.

2.2 *Relational Databases*

Any modern approach to Facility Layout would be expected to be far more encompassing than its traditional counterpart, thereby requiring the collection and management of a very large and varied dataset. A number of potential data fields are suggested in **Table 6**. The relationship between data elements, or referential integrity in database jargon, becomes increasingly important with the number of users and with the breadth and complexity of the problem. Large databases and Relational Database Management Systems (RDBMS) have appeared in Naval Architecture literature for many years **[45]**, with the powerful system employed for 3D Product Modelling (described briefly in Section 1.3) as an interesting demonstration of the ability and advantage of collecting a project's data in one universally accessible group. There is also some merit in making data, as opposed to software, the common element for a project, whether it is Facility Layout or broader Design Synthesis, because it leaves the user free to employ the models and software tools of their choice in the manner in which they choose to use them.

2.3 *Expert/Knowledge-based Systems*

> "A computerized expert system, as the name suggests, models the reasoning process of a human expert within a specific domain of knowledge in order to make the experience, understanding, and problem-solving capabilities of the expert available to the nonexpert for purposes of consultation, diagnosis, learning, decision support, or research. Usually an expert system is distinguished from a sophisticated lookup table (which merely maps questions to answers) by the attempt to include in the expert system some sense of an understanding of the

meaning and relevance of questions and information and an ability to draw non-trivial inferences from data[46]."

Although expert systems have been used for engineering problems for some time, they have been shown to be ill-suited for broad or complex problems[47]. Knowledge-based systems, often referred to as 'Expert Systems', are generally used where a decision is required based on incomplete or conflicting data. In this instance, a knowledge-based system would be used to derive information where data elements are missing or are contradictory. For example, the FLP formulation will require length, width and height in order to create a new space. However, it is often more practical to define a space on the basis of an area and height. The knowledge-based system would be used to determine the values which would not be specifically prescribed by the user but are required by the algorithm. In a second example, where values for length, width, and area have been prescribed but are in conflict, a knowledge-based system can be used to check and resolve the conflict. A second application of a knowledge-based system would be to manage qualitative constraints such as large, small, airy, etc.

The contents of **Table 6** also suggested that it may be possible to employ knowledge-based systems to improve the layout decision process. The expert system's ability to infer missing information and to represent the practical experience of their human counterparts makes them ideal tools for the rapid development of innovative layout solutions. As such they extend the capability of modern optimization algorithms such as Genetic Programming and Simulated Annealing[48]. Details and development for this topic have been left for future work.

2.4 *Routing Problem*

The routing problem calls for the determination and placement of efficient routes between various spaces in the layout. This includes the construction and cost minimization of corridors and services such as piping, wiring and ducting. Most importantly, the algorithm examines cost reduction through the sharing of routed services. Once solved, the results of this section will be used to contribute to the layout's overall score. Like the knowledge-based system development, this section has been left for future work.

2.5 *Traditional Facility Layout Approaches*

This category refers to documented and accepted practices for the solution of Facility Layout Problems. Solutions for the sub-problems *Creation of a Layout Plan* and *Decision of Layout Improvements* have traditionally been based on the relationships between the nodes of spaces as opposed to the spaces themselves — i.e., independent of spatial constraints. Despite the poor manner in which traditional Facility Layout Algorithms represent spatial objects, many of the steps of their algorithms are quite elegant and are worthy of further consideration and application. Scoring, placement ordering, and improvement methods have been well studied**[49][50][51][52][53]** and there is no reason why they cannot be used with an improved representation format. Because of these strengths the traditional Facility Layout Algorithms provide an excellent place to begin the development of a new model appropriate for Naval Architecture. This section has been left in the general terms above as it has already been well examined in many sources, and since it has not been implemented in this phase of the *ShipArrT* project.

2.6 *Semi-Solids Modelling*

Semi-Solids modelling refers to a method for the representation and automated manipulation of regular and irregular three dimensional objects. Developed for this project, it addresses the limitations of solid and surface models and offers a viable alternative to Spatial Enumeration methods such as that used in block layout. A description of CAD model representations has been included in Appendix 1. While computationally more demanding than the traditional block formulation, Semi-Solids is significantly more flexible and able. The term Semi-Solids was coined to reflect the similarities and differences between this formulation and a traditional Solid Model. For data storage and for the purposes of manipulation, an object created as a Semi-Solid closely resembles a solid model employing a Boundary Representation. A Boundary Representation means that the object is defined by the surfaces, lines and vertices of its exterior. Surface definitions such as colour and texture may also be included in a Boundary Representation. Appendix 1 contains a more detailed explanation of Solid Modelling. The 'semi-' designation refers to the atypical manipulation process employed by the formulation.

The majority of Solid Modellers create a single entity from the two or more primitive entities by *intersecting* the primitive objects mathematically. Referred to as Constructive Solid Geometry, the process employs Boolean Set Operators such as ***union***, ***intersection***, ***addition*** and ***subtraction*** to manipulate simple objects such as cubes and cylinders. Underlying Constructive Solid Geometry is the assumption that any object can be created from a combination of primitive objects. However, the approach is one of brute force and may require many primitives and a complex series of manipulations in order to create a shape which might have been more easily defined by its boundaries. A tetrahedron (a four-sided pyramid) is an excellent example of such a case (**Figure 16**). Shapes whose surfaces are greater than second order, such as the splined bicubic surface of a ship's hull, cannot be modelled using traditional solid modelling approaches and certainly cannot be constructed through the use of primitives. Since the ultimate goal of this project is the design of ships, the problem of manipulation is one which must be overcome in their representation. In addition, it has been found that Constructive Solid Geometry cannot cope with situations in which primitive objects do not contact or overlap one another — a situation common in Facility Layout. Finally, like almost all CAD systems, Solid Models have been constructed with the intention of interactive usage. For this reason, there has been very little success in the automation of Constructive Solid Geometry — of having the computer decide which primitives and Boolean transactions to use to create a more complex solid.

Given these limitations, Semi-Solids relies on the alteration of an object's boundary definition for changes in shape and size. The process is not trivial however. An analogy to the difficulty experienced by the computer in dealing with a layout problem would be to consider the plight of a blind person attempting to work with the Program Manager within Microsoft Windows. Even if the person is able to find an icon to activate, they will not know what program the icon invokes — the graphical icon object has no meaning because they do not use the visual medium to interpret the world. The situation is further complicated because the computer lacks the blind person's spatial understanding of the objects on the screen (e.g., the icons are adjacent to one another). Constructive Solid Geometry copes with highly complex objects because the software can refer back to the primitives from which the object was created in order to create a new surface. In Semi-Solids, the surface of a new object must reflect the surfaces of its adjoining neighbours — there is no underlying combination of objects from which a surface can be derived.

The process of developing a new surface on the basis of neighbouring objects is a four-step process in the Semi-Solids formulation, and a chapter has been devoted to each. The high computational demands of the approach are consistent with the computer's difficulty in assigning *meaning* to graphical data. Manipulation using Semi-Solids is performed facet by facet and it has been assumed that facets can only be altered in a direction perpendicular to the plane of the face. Hence, a cube requires the process to be pursued six times, one for each of its faces, plus additional iterations for any newly created patches.

The first step for the computer is to create an object and place it at a location in the design space. In the context of **Table 5**, the location of a new space is an aspect of the problem which can be developed using traditional FLP approaches. Even so, the initial location is a non-trivial problem requiring further work and will be discussed in Chapter 7. Fortunately, considerable progress has been made in previous work by Industrial Engineers. For each new object, dimensional information, including plane equations for each face, is defined by the algorithm. Also, the object is defined in the same manner as a mesh such that only one facet can adjoin the edge of another facet (**Figure 17**, **Figure 50**).

The analysis of the object begins by identifying if and which objects are in the proximity of the new object. Methods developed for computer graphics problems use projections of lines and points in order to correct the display of coincident and hidden objects. However, the process is carried out on a pixel-by-pixel basis consistent with the display of the object and is inefficient for the purpose of an automated model suitable for Facility Layout. Instead, Semi-Solids takes advantage of planar mathematics to identify objects neighbouring each plane.

Since each facet of the new object is examined independently of its neighbours, it is convenient to refer to it as the Plane of Interest or POI. The algorithm creates a prism perpendicular to the POI and by a process of evaluation and substitution identifies those objects which intersect the prism. Because plane equations are used to define each object, the equations can also be used to determine if the POI faces the interior or exterior of a neighbouring object. For the purpose of decision making, it is convenient to sort the neighbouring objects on the basis of distance from the POI. The decision process in which a choice is made between moving the POI and allowing it to remain unchanged has also been left for future work and is described in the last chapter of the project. However, for this discussion it has been assumed that the decision was to make a change.

The next chapter of the algorithm deals with altering the boundary of a Semi-Solid object and is carried out in two phases. First, the POI must be altered to resemble the surface which it will adjoin. Since this step often requires the creation of additional facets, an algorithm for doing so has been developed. For each interfering patch, it determines all 24 potential intersection points and then reduces this to a list of no more than eight vertices from which the new patch or patches will be formed. A sort is then used to determine the correct order of the vertices, and the new patch or patches are created in a process much like connect-the-dots.

Chapter 6 describes the final step of the process as one of accounting in which the adjoining object faces are updated to reflect any new patches which have been created by the previous step. It again uses sorts and a connect-the-dot process to create new patches. The model can then continue by assigning the next facet of the new object the role of POI, until all the facets, including any which are newly created, have been examined and updated. The final step involves the removal of unnecessary patches and has been left for future work as described in Chapter 7. The next four chapters explore each of these steps, and their relationships to one another are shown in the flowchart in **Figure 18**.

2.7 *Development*

The solution of Facility Layout Problems requires a great deal of iteration and the problem does not lend itself, at least initially, to interactive approaches. Because the layout is controlled by the weighting parameters used in its solution, an interactive interface with the graphical depiction of the layout is unnecessary. Images will need to be generated to illustrate the solution of the problem, but these will only be viewed once the algorithm has completed its deliberations. In this, the new program differs significantly from typical CAD software in which the manipulation of objects is almost entirely controlled by a graphical and interactive *drag-and-drop* approach. Although the drafting program AutoCAD is effectively an industry standard for the underpinnings of commercially available Facility Layout Algorithms **[54]**, the difficulty it poses as a development environment discouraged its use in the same capacity for this project. Further, it was feared that too much effort would be spent customizing the algorithm for AutoCAD rather than developing the best layout engine possible. For these reasons it was decided to focus the *ShipArrT* development effort on the manipulation of data leading to the solution of the FLP and not on its display. Hence coding was required for the importation and exportation of data and images, while their display and printing was to be handled using commercially available third-party software. The transfer format was chosen to be the *Drawing eXchange Format (*.DXF)* used by *AutoCAD* design software. The reasons for this decision include:

- *AutoCAD* is a sophisticated 3D design environment capable of the depiction and editing of solids and meshed surfaces
- DXF format lends itself to computerization and is freely available to software developers

- *AutoCAD* and DXF are common exchange formats for graphical information used by many programs in addition to AutoCAD
- an understanding and use of DXF was already required for the importation of hull surface information from the *AutoShip* software available through the Faculty of Engineering at Memorial University of Newfoundland

Having dealt with the depiction problem through translation to AutoCAD a development environment was required for the complex *ShipArrT* algorithm. A number of programming languages were considered until it was discovered that Microsoft had developed both *Visual Basic* and the Relational Database Management System (RDBMS) *Access*. The two programs are compatible to the point where they can be considered combinations of each other. *Visual Basic* makes available to software developers the same database engine used in *Access*, which effectively makes it possible to recreate *Access*. Microsoft also moved to use *Visual Basic* as its macro language in *Access*. Following development work in both environments, it was concluded that *Visual Basic* lent itself to the development of interface, but made manipulation of the database difficult. Conversely, using *Visual Basic* as a macro language within *Access* introduced minor interface limitations, but the database elements could be manipulated with far greater ease. Again recognizing that the task at hand was the Facility Layout Algorithm and not the interface, it was concluded that *Access* was the best environment for the development of *ShipArrT*.

Since this thesis has concentrated on the development of Semi-Solids, programming has also focussed on this part of the problem. As suggested in the previous section, the next three chapters deal with the Semi-Solids formulation. However, only the code for the algorithm described in Chapter 4 has been implemented because of time constraints. The algorithm for the material found in Chapters 5 and 6 has been expressed as the pseudocode found in Appendix 2 but should be more than adequate to demonstrate the workings of the Semi-Solids formulation. Even if the coding was completed for Semi-Solids, additional work will be required for a decision engine to drive the formulation. More on this can be found in Chapter 7. Both the code and pseudocode have been collected in Appendix 2.

There are considerable differences in the style of the completed code when compared to that which is only in pseudocode form because the former has, wherever possible, taken advantage of the database engine to perform many computational tasks. It was hoped that the use of the database's intrinsic functions for sorting and data manipulation would reduce the algorithm's run-time and prove less complicated to code and debug. While it was successful, it was found that the database tools were better suited to batch oriented tasks such as 'sort this list' or 'calculate X for each of this list'. Since the algorithm of Chapters 5 and 6 requires the use of multiple 'if' statements for more specific checks, it is likely that the database functions and the Structured Query Language by which many tasks are executed will be less prominent in future coding.

A broad and detailed database structure was developed in addition to the work on Semi-Solids. In part a response to the long-term needs of *ShipArrT*, the database is also the repository for the spatial data used by Semi-Solids. Its structure has been explored over the next few sections.

Finally, Chapter 8 contains suggestions for future work in expanding this database, and provides a detailed outline of the direction of further research should pursue towards completing the new General Arrangement model.

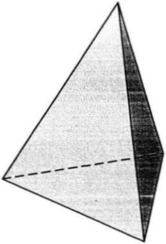


Figure 16 A tetrahedron.

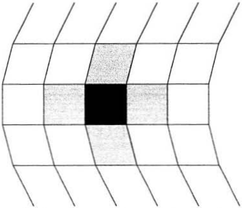


Figure 17 An example of a valid mesh element showing the four adjacent sides.

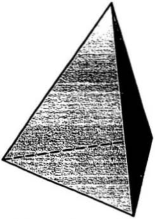


Figure 16 A tetrahedron.

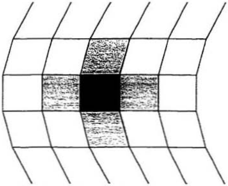


Figure 17 An example of a valid mesh element showing the four adjacent sides.

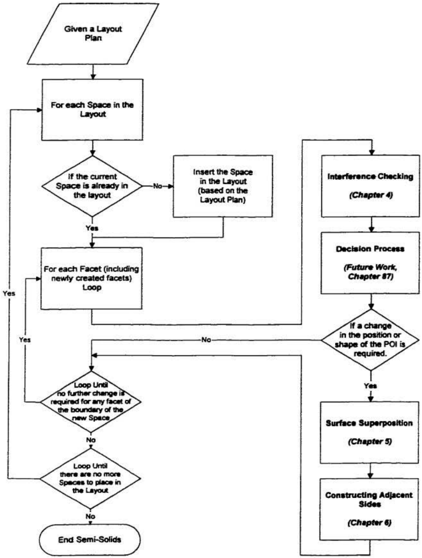


Figure 18 Semi-Solids general algorithm. The flowchart shows the relationship of the material presented in the next three chapters.

Layout Algorithm Step	Software/Approach Required
Gather & Manage Constraint Data	Relational Database
Represent Qualitative Constraints	Uncertainty Variables
Interpret and Deal with conflicting Data	Expert/Knowledge-based System
Create a Layout Plan *(deals with relationship constraints)*	Traditional Approaches, Probabilistic Methods, or Graph Theory
Generate Layout including Corridors *(deals with spatial constraints)*	Semi-Solids Modelling
Generate and Superimpose Services *(deals with most distance-based constraints)*	Routing Problem
Evaluate Layout *(evaluates score of spatial and distance-based constraints)*	Traditional Facility Layout Approaches
Decision of Layout Improvements *(what changes to make to Layout Plan or Spatial Layout)*	Traditional Facility Layout Approaches Possibly improved through the use of an Expert/Knowledge-based System
Execution of Layout Improvements	Semi-Solids Modelling
Report Generator *(data, drawings, costs, inventories, etc.)*	

Table 5 Steps required in the development of a modern Facility Layout Algorithm. In addition to the material presented in this chapter, a discussion of future research directions for Facility Layout can be found in Chapter 8.

Topic	Description
Components	doors, windows, hatches, skylights, etc.
Materials	walls, piping, wiring, floor covering, noise control, etc.
Space Definitions	dimensions, boundaries, access, etc.
Service Requirements	electricity, water, sewage, light, HVAC, etc.
Relationships	relationships and links between spaces
Regulatory Requirements	Lloyd's Register, Canadian Shipping Act, etc.
Characteristics	weights, manufacturing and assembly constraints, etc.
Solutions and Scenarios	
Documentation	
User Information	responsibility, security, preferences, etc.
Lists of Changes and Updates made to the Database	

Table 6 Steps required in the development of a modern Facility Layout Algorithm. In addition to the material presented in this chapter, a discussion of future research directions for Facility Layout can be found in Chapter 8.

3

The *ShipArrT* Database

The *ShipArrT* program takes full advantage of the capabilities of the relational database format. Related information is kept in simple tables which are linked to other tables by a variety of relationships. Continuity via relationships is controlled through the use of pointers. Pointers are pieces of information common to more than one table, and are used to direct the program to specific records. Field names including the script "_ID" refer to pointers. In structure, the database can be thought of as a series of zones as depicted in **Figure 19**.

3.1 *Zone 1: Interior Inventory*

Zone 1 can be thought of as the trunk of the tree which is the *ShipArrT* database; it contains three tables — *Ship Overall*, *Class List*, and *Space List*. The zone is depicted graphically in **Figure 20**.

ShipArrT begins the design process by allowing the user to stipulate the quantity of a particular class of spaces to be placed in the layout. The class definition does not include the specific location of each space; it merely identifies data and constraints appropriate for each space. Further, it offers the potential of customization or using room classes which have been previously defined either by the user or by means of architectural standards.

Using the example of a cruise ship, a user might require the spaces suggested in **Table 7**. In this example, 366 *Spaces* or rooms are defined using data for only seven *Classes* of spaces within the ship. The table illustrates the spatial efficiency of a relational database since the *Ship Overall* table contains only those elements which are necessary to define the ship. Its companion table *Class List* contains a long list of potential spaces for the layout, not all of which need be used. Hence the data in *Class List* is readily available to the user, but does not impede or denigrate the performance of the database or *ShipArrT* program.

The depiction of this zone in **Figure 20** includes the data type and hence the field lengths for each field. In particular, the field *Class_ID* has been defined as a byte (an integer between 0 and 254). The fields of a database have fixed lengths corresponding to the entries in **Table 8** and where fields are shared between tables so are the data formats. **Table 8** shows the data types available in Microsoft *Access*.

The *Class List* table stores a 50-character name for each class, a number for the occupancy of the room, and pointers to adjoining and related tables in which specific information about the room is stored. The reason for this choice was that a number of defining parameters might be common to several spaces. For example, architectural standards suggest a reasonable range of floor areas for bedrooms, but different classes of bedrooms may differ in their contents, or access to windows, and so on. This format minimizes the amount of repeated information in the data-set, and keeps each table small in size. Also, the *Class List* table is entirely editable, thereby providing the user the opportunity to edit the definition of a class or to add new class definitions to the table.

The pointers in this list refer to different constraints:

Constraints_ID points to a table which contains other pointers regarding the spatial constraints of the class. These may be summarized as length, width, height, area, volume and shape and will be described in greater detail in the sections dealing with Zones 5 and 6.

Relationships_ID refers to entries in an adjoining table in which the adjacency and other relationships are defined between different classes.

Boundaries_ID points to a table containing information regarding the boundary of a space. This might take two forms: that of the materials used to create the walls, floor and ceiling; and that of specific information about the existence of the walls, floor and ceiling. By way of example for the latter, the foredeck area of a ship could be defined as a room with partial walls (rails) and without a ceiling.

Entries_ID points to a table containing information regarding the access to a space. Like that of the boundaries table, this might also take two forms: that of the specification of prefabricated doorways and hatches; and that which locates those entry points (interior/exterior, hatch above or hatch below, multiple doors, doors at either end of a room, etc.).

Windows_ID points to a table containing information regarding the windows of a room. Also similar to the boundaries table, this might take two forms: that of the specification of prefabricated windows and skylights; and that which locates those components (interior/exterior, skylight above or below, multiple windows, windows on several walls of a room, etc.).

Services_ID deals with the services required for the space. These might include: potable water, hot & cold water, seawater, pneumatics, hydraulics, electricity, communications, grey water drainage, sewage drainage, etc. The capacities required for each service will also be included since this information will affect the requirements of the layout solution.

Contents_ID points to a table in which the ID values for a list of furniture, machinery and other interior contents are listed. Specific information about these elements will be found in a third table and might include the physical dimensions and weight of a particular piece. Since the *Access* database has the capability of storing graphic images as part of a database, a raster bitmap image of each object might also be included.

Subspaces_ID refers back to the class list and identifies spaces which can be treated as part of a larger space. For example, a washroom in a passenger cabin could be considered to be a sub-space of the cabin since the washroom will always be adjacent to the cabin. This can lead to an efficiency for manufacturing since the space and its sub-space are treated as a single object thereby facilitating the modularization of each space. There is also a computational gain since the algorithm can treat the two spaces as one in the construction of the layout.

Regulation_List_ID points to a table containing constraining information in the form of standards of various civilian and maritime regulatory agencies.

Comments_ID identifies relevant entries in a table containing textual information about the class. This format again takes advantage of the efficiency of the relational database since the pointers for two classes may point towards a single description, or multiple text entries can be attributed to the same pointer. Since databases use fixed formats for data storage, textual information must be limited to the 254 character limit of the text field. However, by using the same pointer value for several records or by employing the memo data type, more text information can be stored for the particular pointer.

The third table in this zone is *Space List* in which each of the spaces identified in the table *Ship Overall* are given specific identifying codes and names. These are the objects or Semi-Solids which the algorithm will arrange and manipulate. The contents of this table are generated automatically by modules of Visual Basic code in which the program creates a record for each space of each class. Returning to the example which began this section, this means that the table will contain 367 records, thereby identifying each space. Objects such as a hull form can be imported into the database; *Space List* also stores the names and identities of imported objects.

3.2 *Zone 2: Spatial Definitions*

The external boundary of each space is defined by a meshed surface. In turn, the mesh can be defined as a set of patches. In this zone the relationships which define these meshes are described (**Figure 21**).

As discussed in Section 2.4.1, the table *Space List* contains identifying information for each object in the layout. Its adjoining table *Patch List*, while simple in appearance, involves complex data relationships. For each *Space_ID* value there is a set of patches which are used to define the mesh which describes the object. In the table *Patch List*, a counter column identifies each patch element of the mesh and for each, the database stores a corresponding *Space_ID* pointer to attribute them to a particular object. Each mesh element is unique such that no patch of another object (or the first object for that matter) can be exactly the same as any other patch. This does not in any way mean that two patches cannot lie back to back. In fact, the opposite is true and for the solution of the problem of fitting, this superposition of one patch onto another will be used extensively. This process is described in detail in Chapter 5.

Consider the example of two patches which lie back to back. Presumably they are part of two different objects since otherwise they would enclose a space of zero volume. The term 'back to back' is important: while the patches appear to be the same, and the vertices which form the corners of the patches are common to both, they differ in the ordering of those vertices as either clockwise or counterclockwise. This affects the plane equations which define the patches, and also affects the adjacency pointers relative to each side of the patches (**Figure 22**). Hence, each patch, while potentially similar, is unique because of the process which has been defined for the evaluation of the terms *inside* and *outside*.

Corresponding to each patch in the *Patch List* table are entries in four other tables. *Patch Corners* is a table which contains pointers to a table of vertices. The *Patch Adjacency* table identifies which patches share edges with each mesh element, thereby ensuring the treatment of the sets of patches as an object defined by a mesh, as opposed to treating the set of patches as an object defined by a number of disjointed or unrelated facets. The table *Patch Equations* contains the four plane equation coefficients required for the patch. The final table, *Patch Hidden Edges*, contains edge visibility flags; this table is unimportant to the *ShipArrT* algorithm, but does facilitate the exportation of the solution layout meshes.

Unlike the tables in Zone 1 in which pointers referred to relatively few data elements, the tables in this zone use the Long Integer format for storing pointers. This eliminates any difficulties which might arise from large or complex models as over 4 million pointers may be assigned. Such problems were considered because of the assumption that for highly curved surfaces, the patch sizes would be reduced to more accurately represent the curves with the flat sided mesh elements.

3.3 *Zone 3: Patch Adjacency*

One of the most useful characteristics of a relational database is its ability to reflect editing changes through its related tables. Referred to as *Referential Integrity*, this property is nowhere more evident than in Zone 3 (**Figure 23**). The zone centres around the table *Patch Adjacency* and is one of four tables associated with the definition of a patch as described in the previous section. *Patch Adjacency* stores data such that for each *Patch_ID* value, ID values are stored for each of the four patches which surround and adjoin the patch.

The tables which are linked to the *Patch Adjacency* table are simply additional instances of the *Patch List* table. The reason for this lies with the need for referential integrity — the ability of related tables to update one another. Such an instance might occur when a user decides to alter the ID value for a particular patch. Not only must that ID value be altered wherever it appears, but in the *Patch Adjacency* table the pointers which direct the mesh to that patch must also be updated. By way of mechanics, the value *Patch_ID* is only stored in the *Patch List* table. Its appearance in other tables is a result of the referential information. That is, when the table *Patch Adjacency* is viewed, the *Patch_ID* values appear in the table but actually reside in the *Patch List* table. Editing of the ID values, even from within the *Patch Adjacencies* table, will alter those in the *Patch List* table and they will then reappear in altered form in the *Patch Adjacencies* table. Therefore, a referential conflict exists when the table *Patch Adjacency* is displayed because more than one *Patch_ID* value is to be displayed for a particular record. It is for this reason that the four additional instances of the *Patch List* table have been created. These circumvent the referential conflict by providing an unhindered *Patch_ID* value for each of the four pointer fields *Patch_1*, *Patch_2*, *Patch_3* and *Patch_4*. The multiple instances are effectively copies of the original *Patch List* table, and any changes which alter the *Patch List* table will also alter the data found in the four additional instances. For a 4 x 4 mesh such as that in **Figure 24** the table would take the form shown in **Table 9** of a number of pointers identifying other patches. Two special situations should be mentioned. First, there is the potential that a patch might only have three neighbours, as would be the case for a triangular patch. In this instance a null or blank field will appear in place of the absent fourth neighbour. Second, in the case of the importation of a hull form object, the object may not take the form of a closed object but may instead be just a surface. For example, *AutoShip* appears to have difficulty exporting joined objects and hence two or more objects such as a hull and a deck will be saved as separate files. When imported into *ShipArrT*, each of these objects will be given *Space_ID*s in the *Space List* table described in Section 2.4.1, and the points which define these objects will be stored in the appropriate spatial definition tables. Since the importation of unclosed objects creates a situation in which not all patches have four neighbours, tolerance has been built into the system to allow for these null neighbours to appear as nulls in the *Patch Adjacency* table.

3.4 *Zone 4: Patch Limits*

As in Zone 3 (**Figure 23**), multiple instances are used to establish the relationships associated with the vertices which define a patch. Zone 4, the contents of which are shown in **Figure 25**, uses multiple instances of the table *Vertex List* to define the corners of a new patch. These are linked by means of pointers to the table *Patch Corners* which stores pointer values for each of the four corners of the patch. The use of these pointers makes it is possible to store all of the vertices in the database in a single long table, thereby reducing unnecessarily repeated data in the database.

The table *Vertex List* stores the X, Y and Z coordinates of each patch corner. Double precision values are used for these values to ensure accuracy in the model since rounding could create disagreement between the points which comprise a patch/plane and the equation of that plane.

As discussed in the previous section, multiple instances are used in situations in which a single record contains more than one pointer to the same table. Here, *Vertex List* appears four times, once for each of the four corners of each patch record.

3.5 *Zone 5: Patch Equations*

Once again, multiple instances of a table, in this case *Equation List*, are used to represent the five equations associated with each patch. The tables and their relationships are shown in **Figure 26**.

In the *Equations List* table, each record stores the four coefficients for each plane equation. Just as for the table *Vertex List* described in the previous section, double precision values are used to ensure accuracy in the plane mathematics. For each patch noted in the *Patch List* table, there will be a corresponding record in the Patch Equations table, with the data elements being pointers to the five equations comprising the five planes associated with each patch. The first plane, or 'Face', is that of the patch itself, and the remaining four planes follow the edges of the patch and are perpendicular to it, thus forming an open box shape.

3.6 Zone 6: Constraints

This region of the database has been left for future work. While far from complete, it has been included here for the purpose of illustration.

Zone 6, shown in **Figure 27**, contains the now familiar multiple instances of tables containing similar information, with pointers to each instance. The *Constraints* table contains data in two forms: bit flags which indicate whether the user has specified a particular dimensional constraint; and ID pointers all referring to either the *Dimensions* table or the *Constraints Shape* table. The multiple instances of the *Dimensions* table is a result of a recognition that the five spatial constraints — height, length, width, area and volume — share the same data storage format. Both storage space and model complexity are saved by this move. Only the *Constraints Shape* table differs in the fields required for its contents.

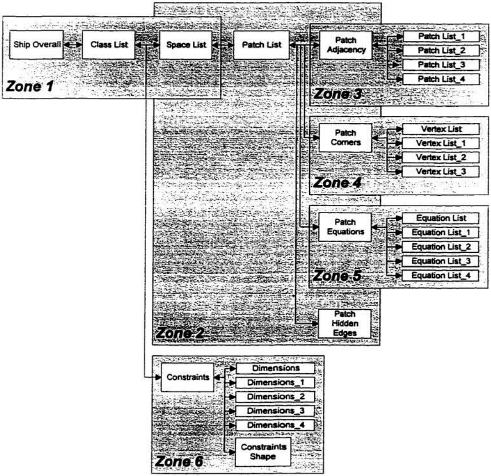


Figure 19 Complete database for *ShipArrT* showing data relationships and zone divisions. The zones divide the database into related topics and will be used to facilitate the explanation of the database later in the chapter.

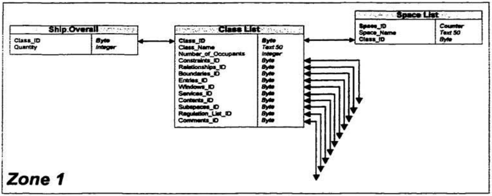


Figure 20 The tables of Zone 1. This zone contains the ship's overall description and links Spaces to their constraints.

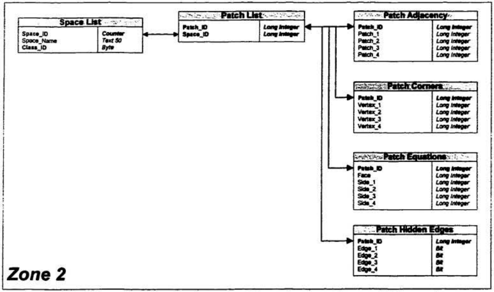


Figure 21 Table elements comprising Zone 2. These elements relate spatial data such as vertices to each Space / room in the layout.

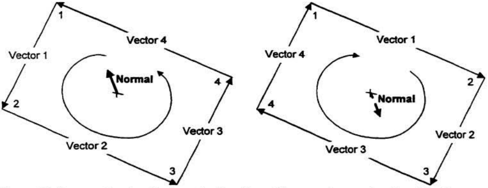


Figure 22 Two patches showing how the direction of the normal vector is affected by the relative numbering of its vertices.

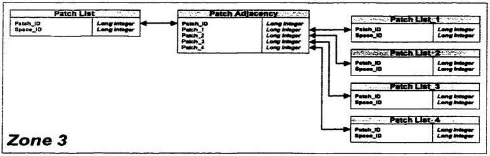


Figure 23 Depiction of tables and relationships for Zone 3. The zone represents neighbourhood data for each surface patch by identifying the adjoining patches.

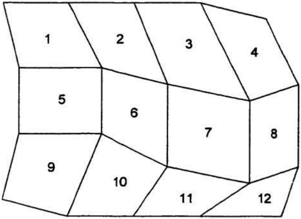


Figure 24 An example of a typical 4 x 4 surface patch.

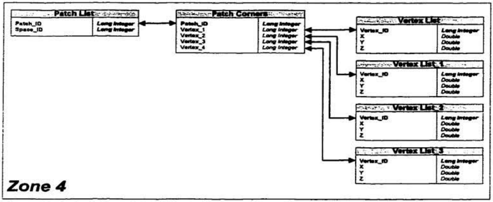


Figure 25 A depiction of the tables and relationships of Zone 4. The zone involves the vertex information of the corners of each patch.

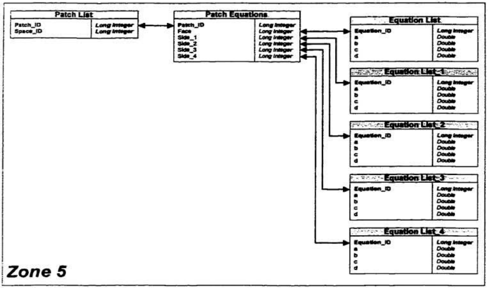


Figure 26 A depiction of the tables and relationships of Zone 5. The zone deals with the mathematical definition of each plane and its coincident orthogonal surfaces.

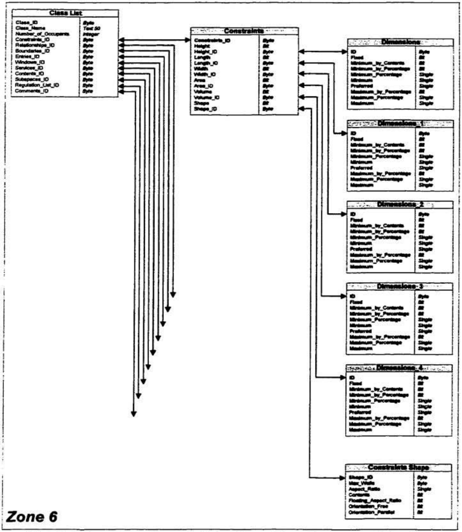


Figure 27 A depiction of the tables and relationships of Zone 6. The zone deals with the constraints associated with each Space / room in the layout. In particular, it demonstrates how pointers can be used to attribute a large quantity of information to a single Space_ID.

Tables Pertaining to Chapter 3

Quantity	Class
100	First Class Cabins
200	Second Class Cabins
50	Two person Crew Cabins
10	Officer Cabins
1	Engine Room
4	Machinery Rooms for HVAC
1	Galley
	etc...

Table 7 Examples of the spatial requirements for a cruise ship. The list shows how many of the areas of the ship can be treated as quantities of a relatively few number of space Classes.

Data Type	Description
Text	Alphanumeric characters up to 255 bytes (1 byte per character).
Memo	Alphanumeric characters (usually several sentences or paragraphs) up to 64,000 bytes.
Date/Time	Dates and Times, always occupies 8 bytes.
Currency	Monetary Values, always occupies 8 bytes.
Counter	A numeric value that Microsoft Access automatically increments for each record you add. Occupies 4 bytes.
OLE Object	OLE objects, graphics, or other binary data. Occupies up to 1 gigabyte (limited by disk space).
Number	Numeric values (integers or fractional values). Occupies 1, 2, 4, or 8 bytes.
• Bit	A Boolean data element. True/False, Yes/No, On/Off, etc.
• Byte	A combination of 8 bits. It can be used to represent a single alphanumeric character or can also be used to store numbers from 0 to 255.
• Integer	Stores numbers from -32,768 to 32,767 (no fractions). It occupies 2 bytes.
• Long Integer	Stores numbers from -2,147,483,648 to 2,147,483,647 (no fractions). It occupies 4 bytes.
• Single	Stores numbers with 6 digits of precision, from -3.402823E38 to 3.402823E38. It occupies 4 bytes.
• Double	Stores numbers with 10 digits of precision, from -1.79769313486232E308 to 1.79769313486232E308. It occupies 8 bytes.

Table 8 Database field data types.

Patch Adjacencies				
Patch_ID	Patch_1	Patch_2	Patch_3	Patch_4
1		2	5	
2		3	6	1
3		4	7	2
4			8	3
5	1	6	9	
6	2	7	10	5
7	3	8	11	6
8	4		12	7
9	5	10	13	
10	6	11	14	9
11	7	12	15	10
12	8		16	11
13	9	14		
14	10	15	13	
15	11	16	14	
16	12		15	

Table 9 Typical contents of the *Patch Adjacencies* table. For the purpose of example, the contents are consistent with the surface patch in **Figure 24**.

4

Interference Checking

This chapter and the two which follow explain the mechanics of spatial manipulation using Semi-Solids. The chapter has been divided into subsections which first introduce a basic object and then discuss the process of locating that object relative to others in an Facility Layout Problem (FLP) layout. A flowchart which illustrates the steps of this algorithm is shown in **Figure 28**. The chapter which follows describes the method by which situations involving spatial conflict are resolved.

At the time of writing, *ShipArrT* is in no way a complete program. For the purpose of illustration, several steps of a Facility Layout algorithm have been passed over, and a number of assumptions have been made. The manipulation of the dimensional variables of length, width, height, area, volume, shape, etc., is not addressed in this phase of the *ShipArrT* development. In addition, the process by which such an object is located at a particular location within the layout has been left for future work. More information regarding these topics is presented in the Chapter 7.

In this section it has been assumed that a user has created a spatial object. Although almost any faceted 3D object can be defined and manipulated using the representational tools developed for this project, a rectangular polyhedron-shaped room will be used for the purpose of illustration. Therefore values exist within the database for the shape, size and initial location of the object. The algorithm has defined the room's boundary by creating a six-sided figure, the sides of which appear as single patches and the whole forming a continuously meshed entity. This is placed in the layout where it joins objects which act as a hull boundary (**Figure 29**).

4.1 Interference Approaches

The computer is much like a blind human at this stage. As in **Figure 30**, the computer is aware that there is an object, and by virtue of coordinate systems, the object is in the vicinity of others such as the hull form. The relative locations of objects in the layout remain unknown but are required in order to complete the layout efficiently, optimally, and without overlap or undue void space.

Void regions and overlap would have been avoided in traditional block layout approaches by verifying the contents of each square of the layout grid. However, *ShipArrT*'s Semi-Solid representation makes this method of verification impossible. A Solid Modeller generally checks interference by attempting to perform a mathematical transaction involving two or more objects. Where there is a change in the net volume of the two objects following such a transaction, an interference exists. Because Solid Modellers are constructed around Boolean transactions, they require manual intervention to correct the dimensions of the original objects to eliminate the overlap. Further, where no overlap exists, the modellers are unable to find a reference by which to determine if an object is in the vicinity of another. To determine adjacency and interference, the Semi-Solids formulation takes advantage of the plane equation information and the sophisticated search engines of the database software.

4.2 *The POI Prism*

Semi-Solids takes advantage of the mathematical properties of the patches to evaluate their relative proximities and orientations. By way of example consider again the object within a hull boundary as suggested by **Figure 29**. Taking the first of its six patches, the task is to determine which other objects lie in the vicinity of that patch. The patch will hereafter be referred to as the POI or *Patch of Interest*. To facilitate this step, the database stores plane equations for the POI and planes which adjoin and are perpendicular to the POI.

The equations are derived from vectors formed by each of the four edges of the POI. The algorithm solves for the cross product of the normal of the POI and each of these vectors. Because of the perpendicular characteristics of the cross products, the resulting four vectors are the normals to planes which are perpendicular to the POI *and* pass through the sides of the patch. Back substitution of the vertices and the new normal vectors yields the remaining $\boldsymbol{d}$ coefficients thereby completing the definitions of these four planes. The calculations necessary are shown in **Equation 2**, **Equation 3** and **Equation 4**.

$$\begin{aligned} Vector_1 &= V_2 - V_1 \\ &= [\ x_2 - x_1,\ y_2 - y_1,\ z_2 - z_1\ ] \\ Normal_{POI} &= [\ a_{POI},\ b_{POI},\ c_{POI}\ ] \\ New\ Normal &= Vector_1\ X\ Normal_{POI} \end{aligned}$$

Equation 2 Calculation of the normal of a surface of the POI Prism. $Vector_1$ is a vector formed by the difference between the vertex coordinates of two adjacent vertices of the POI. $Normal_{POI}$ refers to the normal vector of the POI. The cross product of these two vectors yields a third vector called New Normal. This new normal vector is that of one side of the POI prism.

$$A = b_{POI}(z_2 - z_1) - c_{POI}(y_2 - y_1)$$
$$a = \|A\| = \frac{A}{\sqrt{A^2 + B^2 + C^2}}$$

Equation 3 The A, B and C components of the New Normal vector are normalized in the manner above to magnitudes between -1 and 1 to control the magnitudes of subsequent solutions.

$$V_1 = [\, x_1,\ y_1,\ z_1 \,]$$
$$New\ Normal = [\, a,\ b,\ c \,]$$
$$0 = a*X + b*Y + c*Z + d$$
$$\therefore D = -1*(\, a*x_1 + b*y_1 + c*z_1 \,)$$

Equation 4 Using the coordinates of one of the POI vertices used in the calculation shown in **Equation 2**, this calculation determines the ***d*** coefficient necessary for the equation of the prism side. New Normal refers to the vector calculated in **Equation 2** and V_1 is a vertex of the POI. Back substitution into the plane equation formula yields the *D* coefficient which is then normalized to be consistent with the New Normal coefficients shown in **Equation 3**.

Since the planes are perpendicular to the POI, they never converge (**Figure 31**). Visually, the region of intersection of the four planes can be thought of as a rectangular prism (**Figure 32**). By strictly adhering the vertex order conventions established in the database, the new normals all point into the interior of the region of intersection. Working under the assumption that any change made by a patch will take place only along a line perpendicular to the patch, the prism defines the region in which interference can occur. Metaphorically, the POI is much like an elevator, with its shaft formed by the prism. And just as in the case of the elevator, all is well so long as no other object violates the prism, sharing the shaft with the elevator.

4.3 *Vertex Substitution*

A Dynaset is a database object which appears to be a table, but contains no data. It provides a means of temporarily combining the data elements of several tables in a single table for display and evaluation. Dynasets are editable and the changes are written directly to the tables from which the dynaset was derived. It is similar to the multiple table instances discussed in Section 2.3.3. Tabular dynasets are created to collect relevant data for each of the algorithm steps described in this Chapter.

The next step in the Semi-Solids algorithm is carried out by the substitution of every vertex of all the objects of the database into the four prism equations and the equation of the POI. The results are collected in a dynaset of a form similar to that of **Table 10**. These values are the equation solutions of the POI and its Prism for each vertex in the database.

This is perhaps the least efficient of all the steps of the algorithm because every single vertex in the database is substituted into the five plane equations defined in the previous step, therefore requiring significant mathematical evaluation. However, compared to the traditional mesh evaluation using projections and lines, the batch nature of the process removes the evaluative and primitive development steps from each iteration and thereby reduces the time required.

4.4 *Relate Vertices to Patches*

This step creates a large table called *Solutions for Patches*. The data it contains will be used by not only the other queries in this section but also those of the next chapter in which the POI is altered. There, not only will information regarding the patches be required, but so will references to the parent objects of each patch.

The previous step collected a series of information relating the individual vertices of the objects in the database to the POI and its prism. In this step, this data is compiled and expressed in terms of the patches and objects in the database. Because four vertices comprise each patch, there is repetition of many of the fields, notably those which store the solutions for each of the vertices as found in the *Solutions for All Vertices* table described in the previous section. The new table contains the fields depicted in **Table 11**. The table entries begin with reference ID values through which additional information can be determined where necessary. For each vertex in the patch, there are solution values corresponding to the substitutions made in the POI prism. A field referred to as *InOrOut* completes the table and contains the output of a dot product calculation (**Equation 5**) which determines if the POI prism faces the inside or outside of a patch. This distinction is important because it relates the direction faced by the POI to that of the patches in the dataset.

$$\textit{POI Normal} = [\, a_1,\ b_1,\ c_1 \,]$$

$$\textit{Patch Normal} = [\, a_2,\ b_2,\ c_2 \,]$$

$$\textit{POI Normal} \cdot \textit{Patch Normal} = \|\textit{POI Normal}\|\,\|\textit{Patch Normal}\| \cos\theta$$

Equation 5 Given the normal vectors of the POI and a neighbouring patch, a Dot Product is calculated which establishes the relative orientations of the two patches. The solution *InOrOut* which replaces $\cos\theta$ gives a value from -1 to 0 for patches facing the same direction and a value from 0 to 1 for patches facing one another. Where *InOrOut* = 0 the patches are perpendicular to one another.

4.5 Remove Wholly Excluded Patches

The next step is a delete query in which all the patches found by the query criteria are removed from the *Solutions for Patches* table. This does not mean that the patch information is removed from the database; instead, the items which meet the criteria will no longer appear in the dynaset developed in the previous step. The query evaluates the data stored in the *Solutions for Patches* table to remove patches in which all four vertices lie in the negative or 'out' side of any one of the four planes which define the POI prism. By way of example in **Figure 33**, if the values stored in the fields V1_Plane2, V2_Plane2, V3_Plane2 and V4_Plane2 are all less than zero, then their patch lies wholly outside the first plane of the POI prism. The vast majority of patches in the *Solutions for Patches* table will meet this and similar criteria and will be removed from the table, thus radically reducing the set size requiring subsequent manipulation and evaluation.

4.6 *Perpendicular Patches*

A special case of patches which interfere with the POI prism are those which are perpendicular to it (**Figure 34**). Identification of these patches is carried out by means of a Dot Product calculation between the normal vectors of the POI and each of the patches in the *Solutions for Patches* table. This calculation has already been performed (**Equation 5**) and the results are stored in the ***InOrOut*** field. Where ***InOrOut*** is exactly equal to 0 the patch is perpendicular to the POI.

Testing for interference is carried out by substituting each of the four vertices of the POI into the equation of the Patch. Where any one or more vertices lies on opposite sides of the plane of the Patch, that patch violates the POI prism. Vertices lie on opposite sides when the solution of the Patch Plane equation yields one or more positive or negative values relative to the other three solutions. The patch does not violate the POI prism when all four Patch equation solutions share the same sign. Such patches can then be removed from the *Solutions for Patches* table.

4.7 *The Patch Prism*

At this stage the algorithm has failed to exclude all the patches which lie outside the POI prism. The remaining patches are removed by creating an interference prism for each patch remaining in the *Solutions for Patches* table (**Figure 35**). Unlike the prism developed for the POI, the new prisms will be perpendicular not to their patches but to the POI. This is done by taking the cross product of their border vectors and the normal vector of the POI. The process is virtually identical to that described in Section 4.2 with the exception that the prism mathematics are determined only for those patches in the *Solutions for Patches* table. As a result, the quantity of calculations required are significantly reduced relative to that which would be required for the complete set of patches found in the *Patch List* table. The prisms created in this operation will subsequently be referred to as *Parallel Patch Prisms*. **Equation 6** shows the formulation of the mathematics of Parallel Patch Prisms. Using the same elevator metaphor which was introduced in Section 4.2, the Parallel Patch Prism is an elevator shaft in which the floor of its elevator lies at a slant to its direction of motion.

$$\begin{aligned}
Vector_1 &= V_2 - V_1 \\
&= [\,x_2 - x_1,\, y_2 - y_1,\, z_2 - z_1\,] \\
Normal_{POI} &= [\; a_N,\; b_N,\; c_N \;] \\
\text{Normal of Patch Prism Side} &= Vector_1 \times Normal_{POI}
\end{aligned}$$

Equation 6 In this case V_2 and V_1 lie on the patch and not on the POI. This differentiates between the POI prism and the Parallel Patch Prism.

4.8 POI Vertex Substitution

The next step in the process of identifying patches which neighbour the POI is carried out by substitution of the four vertices of the POI into the Parallel Patch Prism equations determined in the previous step. The dynaset into which the results are stored differs in format from that used in the evaluation of the POI prism in Section 4.3 because in this case four vertices are substituted into many equations rather than many vertices into five equations. The dynaset has been named *Solutions for POI Vertices*, and its fields are shown in **Table 12**. The table contains fewer fields primarily because solutions specific to the POI have already been determined and stored in the *Solutions for Patches* table. Hence solutions are only found for the equations of the prism sides and not for the plane formed by the patch itself. Also, the table is not keyed to the Vertex_ID values as in the *Solutions for Vertices* table but instead to the Patch_ID values, thereby eliminating the need for a compilation step similar to that described in Section 4.4.

4.9 Evaluate External Prisms

Once more a delete query is used to remove irrelevant patches from the *Solutions for Patches* table. In almost exactly the same process described in Section 4.5, the query accesses the information stored in the *Solutions for POI Vertices* dynaset. The patches which are to be deleted are those in which all four POI vertices lie in the negative or 'out' side of any one of the four planes which define each Patch Prism (**Figure 36**). By way of example, if the values stored in the fields V1_Plane1, V2_Plane1, V3_Plane1 and V4_Plane1 are all less than zero, then the patch lies wholly outside the POI prism, and can be discarded. Patches are discarded through their deletion from the *Solutions for Patches* dynaset.

The vast majority of patches in the *Solutions for Patches* table were removed when the planes of the POI prism were evaluated. In the three steps which have followed, the manipulation was carried out only on the patches which remained in the table, and hence only a few patches will be removed by this step in the algorithm. The computation required for the evaluation of the external patches is significantly reduced by working with the smaller dataset.

4.10 Conclusion

The goal of Semi-Solids modelling is to identify the relative positions of objects and to enable the computer to quickly fit objects against other objects, regardless of shape, thereby performing the same function as the blocks in block layout. Since shape is derived from the relationship of flat surfaces, it follows that the more oblique patches which define an object, the more complex its shape.

The Semi-Solids algorithm has assumed that interference between objects can be evaluated on a plane-by-plane or patch-by-patch basis. Further, it has also been assumed that a patch can only be altered in its position along its normal vector. That is, for each POI, the POI can only be moved in a direction perpendicular to its surface plane as suggested by **Figure 37**. This alters the patches which are adjacent to the POI.

The material presented in this chapter creates a list of patches which intersected the POI prism. The POI Prism is a construct used to determine interference and adjacency. Three characteristics were used to determine the position of objects relative to the POI. First, for each vertex of each patch, the *Solutions for Patches* table contains solutions from their substitutions into the plane equation of the POI. Through this technique it is possible to determine the position of the patch relative to the POI. Second, the InOrOut field in the *Solutions for Patches* table is used to indicate whether it is the inside or outside of a patch which faces the POI. A negative sign in this field indicates a patch which faces towards the POI, and a positive sign indicates a patch which faces away from the POI. In cases where the patches face the POI, the POI faces the outside of a neighbouring object. Conversely, patches facing away from the POI, effectively in the same direction as the POI, expose the interior of a neighbouring object. Third, the Space_ID field has been included to ensure that if a patch of one object is considered a boundary of the POI, then all the patches of that object may be considered. Use of these ID's and the information found in the *Patch Adjacency* table can be used exclusively to ensure that the surface is applied only to the near side of an object which crosses the POI prism. Ideally, the InOrOut field could be used to make the same determination but the criteria yields a false result for cases in which an indentation in the surface exists which would present an interior view of an exterior patch. **Figure 38** shows such a case.

Figure 39 shows a potential outcome of the identification process described in the chapter. While the POI in this example points aft, it could just have easily been oriented to coincide with any of the six surfaces of the original object in **Figure 29**.

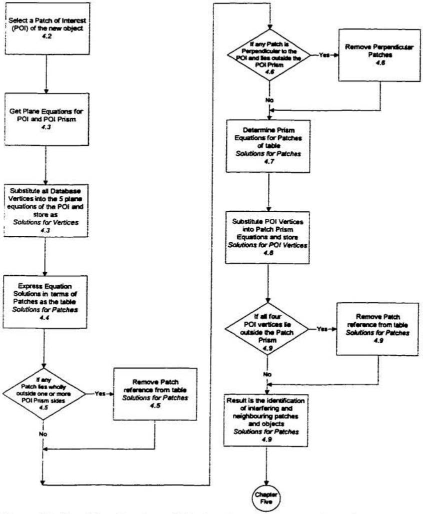


Figure 28 Algorithm flowchart which describes the process of interference checking.

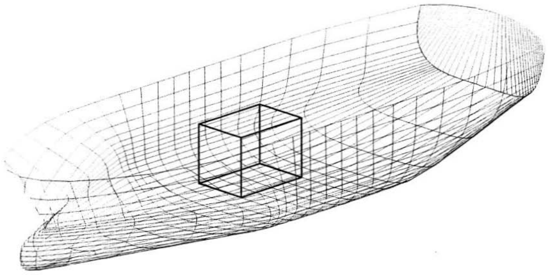


Figure 29 A six-sided meshed object within the boundary of a more complex meshed object.

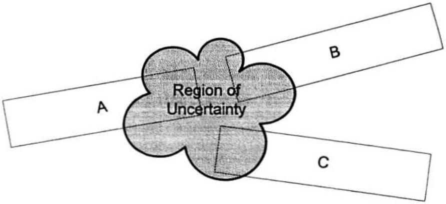


Figure 30 An example of several objects which neighbour each other but do not intersect. The figure suggests the difficulty of identifying the relative positions of non-contacting objects, particularly when the identity of the neighbouring object is unknown.

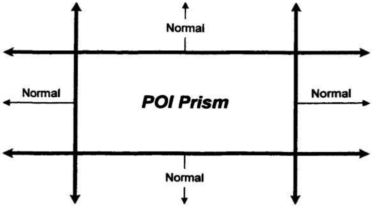


Figure 31 A cross-section of the POI prism showing the normal vectors of the planes which form the prism.

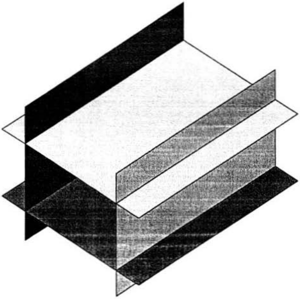


Figure 32 A section of a POI prism showing the planes which define the region. The POI is a patch which is perpendicular to the prism and whose dimensions are the same as those of the interior of the prism. The normal vectors of each plane forming the prism point outwards away from the bounded region.

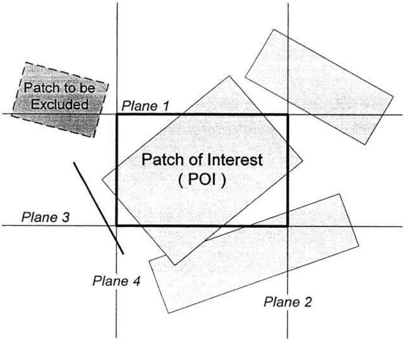


Figure 33 Figure showing five potential cases in which patches may be missed by the first exclusion process. The patch which will be removed from the list of interfering patches lies wholly outside a single plane of the POI prism.

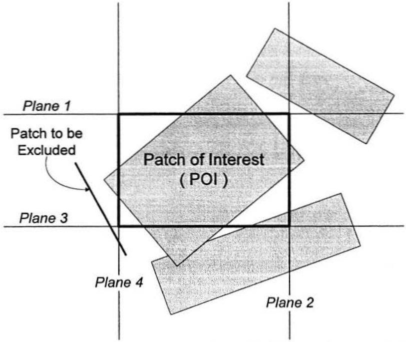


Figure 34 The POI prism showing a perpendicular patch which requires removal from the *Solutions for Patches* table.

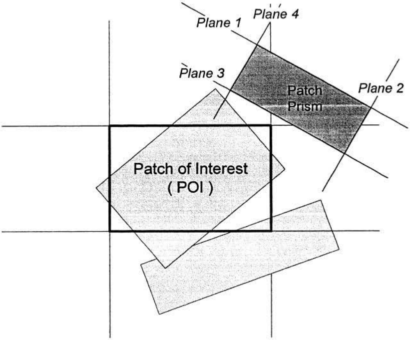


Figure 35 The POI Prism showing a neighbouring Patch Prism.

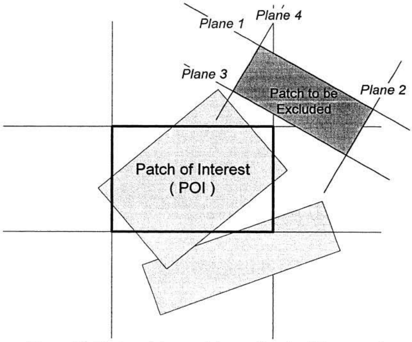


Figure 36 The last of the remaining patches slated for removal.

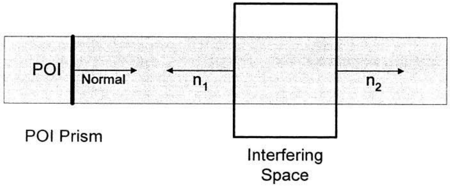


Figure 37 A view of the POI Prism in which a space violates the prism. The normals of the two sides of the interfering space which lie inside the POI point in opposite directions, distinguishing between *inside* and *outside.* The Dot Product of these normal vectors and that of the POI constitute the contents of the InOrOut field of the *Solutions for Patches* table.

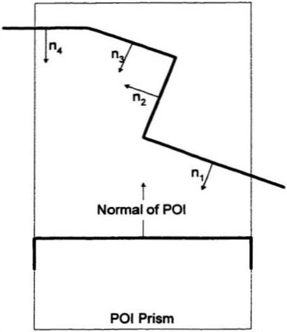


Figure 38 In this view of the POI prism, the object which interferes also presents a negative normal vector to the POI. However, unlike the situation shown in the previous figure, the offending patch is one to which it is intended to mould the POI projection. Hence, it is a case in which the InOrOut field of the *Solutions for Patches* table cannot distinguish between patches to ignore and those to address. The information found in the *Patch Adjacency* table for the particular object can be used to provide additional information.

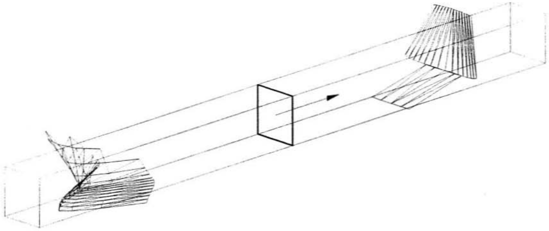


Figure 39 POI Prism showing how the prism is used to identify neighbouring patches and objects.

Solutions for All Vertices					
Vertex ID	POI	Plane1	Plane2	Plane3	Plane4
2	44.455556	89.5446444	-5.5465465	1.2165465	24.546465
4	5.477977654	-54.4879	-4.54646644	6.165465	0.4646
14	0.546746546	6.879465464	78.46876	-66.854644	9.994425
23	65.879878	892.432412	-3.354654	5.500	-6.4654654
...	...	...	...	...	...

Table 10 Typical entries in the *Solutions for All Vertices* temporary table. Each column contains the solutions for the plane equations of the POI prism, with one record for each vertex of the database.

Space_ID
Patch_ID
POI Patch_ID

Vertex1
V1_POI
V1_Plane1
V1_Plane2
V1_Plane3
V1_Plane4

Vertex2
V2_POI
V2_Plane1
V2_Plane2
V2_Plane3
V2_Plane4

Vertex3
V3_POI
V3_Plane1
V3_Plane2
V3_Plane3
V3_Plane4

Vertex4
V4_POI
V4_Plane1
V4_Plane2
V4_Plane3
V4_Plane4

InOrOut

Table 11 The field headings for the *Solutions for Patches* table. It reduces the contents of the *Solutions for All Patches* table from a representation based on individual vertices to one which is based on patches. This shift is required for subsequent analysis of the patches.

Patch_ID
V1_Plane1
V1_Plane2
V1_Plane3
V1_Plane4
V2_Plane1
V2_Plane2
V2_Plane3
V2_Plane4
V3_Plane1
V3_Plane2
V3_Plane3
V3_Plane4
V4_Plane1
V4_Plane2
V4_Plane3
V4_Plane4

Table 12 Field headings for the *Solutions for POI Vertices* table. Because this table is the result of the substitution of POI vertices into the other patch equations of the layout, it is already compiled on the basis of Patch_IDs.

5

Surface Superposition

The previous chapter determined which patches lie in the path of the Patch of Interest (POI) and provided relationship information from which the relative positions of these patches could be studied. A decision engine will interpret this data and decide if a change in the shape or position of the POI is necessary. Generally a change in the POI will be executed so that it can be superimposed against a neighbouring surface. In such cases the decision engine will reduce the list of patches in the *Solutions for Patches* table (**Table 13**) to just those against which the POI should be superimposed. The decision engine has been left for future work but for the purpose of illustration it has been assumed that the POI is to be altered and that a list of adjacent patches has been created.

The algorithm described in this chapter resembles that of Chapter 4 in that the steps of the superposition process will be employed on a patch-by-patch basis. Just as the search algorithm of Chapter 4 examined the patches of the new object one at a time, so will the new patches of the superimposed POI be formed one at a time. Beginning with the list of coincident surfaces identified by the hypothetical decision engine, the work of this chapter is carried out for each of its records. A flowchart of the algorithm presented in this chapter is shown in **Figure 40**.

As suggested in **Figure 41**, each patch in the list of coincident surfaces falls into one of two categories: those wholly contained within the POI prism, and those which are only partially contained within the POI prism. Vertices and planes are used in the evaluation of both cases. A patch is wholly contained within the POI prism when all four of its vertices lie inside the prism. For those patches which only partially cross the POI prism, a sub-patch is required which is comprised of the region of the patch within the prism.

5.1 *Remove Contained Patches*

In this step the algorithm takes those patches wholly contained within the POI prism and copies them as part of the replacement of the POI. Such patches are those in which all four vertices are contained within the POI prism.

This step is literally a copying process. The new patches will use the same vertices as that of the coincident patch. The difference will be in the relative ordering of those vertices because of the impact this has on the direction of the normal of the new patch's plane equation. Similarly, equation and possibly some adjacency information can also be reused to reflect the direction faced by the new patch. Any missing adjacency information such as that required for patches still to be created will be added as it becomes available.

5.2 *Finding Potential Vertices*

The patches which remain in the list are those which are not wholly contained within the POI prisms. For these patches it will be necessary to derive new patches from appropriate vertices. The steps presented from here are applied to each patch in the list individually. Such a patch will be identified through the use of a capitalized name 'Patch'. Each patch in the

ShipArrT database has been stored with an equation for its own plane as well as the equations of four planes perpendicular to this plane. Essentially this is the equivalent of the POI prism introduced in the last chapter and is referred to by the name Patch Prism. It differs from the Parallel Patch Prism developed in Chapter 4 because its sides are perpendicular to its corresponding Patch and not to the POI. The difference makes possible the generic application of the methods described in this section for any potential configuration of neighbouring patches. Were the Parallel Patch Prism used in this section, additional steps would be required to deal with the case of a Patch orthogonal to the POI. Therefore, any reference to a Patch Prism in this Chapter refers to the prism formed by the planes orthogonal to the Patch and not the POI.

New patches are derived from vertices which are found from the intersection points of planes which potentially define a patch. Combinations of intersections of the Patch Plane, a POI Prism side, and a Patch prism side constitute 16 of the 24 possible vertices (**Equation 7**).

$$\begin{array}{ll} \textit{Patch Plane:} & a_1 * X + b_1 * Y + c_1 * Z + d_1 = 0 \\ \textit{POI Prism Side:} & a_2 * X + b_2 * Y + c_2 * Z + d_2 = 0 \\ \textit{Patch Prism Side:} & a_3 * X + b_3 * Y + c_3 * Z + d_3 = 0 \end{array}$$

Equation 7 Equations used to determine 16 of the 24 potential vertices resulting from the intersection of the POI Prism and the Patch Prism.

Four of the remaining eight points are taken from the four vertices of the Patch, and the last four are found from the intersection of the Patch Plane and two adjacent POI Prism sides (**Equation 8**).

$$
\begin{array}{ll}
\textit{Patch Plane}: & a_1 * X + b_1 * Y + c_1 * Z + d_1 = 0 \\
\textit{POI Prism Side } (i): & a_2 * X + b_2 * Y + c_2 * Z + d_2 = 0 \\
\textit{POI Prism Side } (i+1): & a_3 * X + b_3 * Y + c_3 * Z + d_3 = 0
\end{array}
$$

Equation 8 Equations contributing to an additional four potential vertices. The solution of this system of equations effectively projects the four vertices of the POI onto the neighbouring Patch.

Graphically, the 24 vertices might take a form such as that shown in **Figure 42**.

Because the goal of the work presented in this chapter is to fit one object against the boundary of another, the Patch Plane equation is used in the calculation of all intersections because it is against this plane which the newly created patches will be located. As a result, all of the vertices will be coplanar to the patch plane. An error function is used to flag unsolvable vertices. A vertex may be unsolvable when two or more of the intersecting planes are parallel, or if the patch or prism only has three sides. Pseudocode which finds these 24 points is shown in Appendix 2.

5.3 *Verification of Vertices*

This section describes how the number of points found in the previous step is reduced to a maximum of eight potential vertices for new patches. The reduction is performed by the substitution of each vertex into the four prism plane equations of the POI and the four prism plane equations of the Patch. A vertex is valid where it is wholly contained within all eight planes (i.e., where the solution of each vertex in each plane is greater than or equal to 0). Pseudocode which performs this decision is shown in Appendix 2.

The vertices which are selected in this section have been found in no particular order. Interestingly, they also form a convex hull — a region defined by a set of points where all the points lie on the exterior boundary. A property of the intersection of two four-sided patches is that the vertices which define the intersection region always define the exterior boundary of the convex hull. Thus, no concave regions will be formed between vertices, so long as they are taken in the appropriate order. While the convex hull region is obvious when viewed, its development and evaluation is much more involved for the 'blind' computer. The sorting and formation of the convex hull region will be described in detail in subsequent sections.

5.4 *Counting the Vertices*

Next it is necessary to tally the vertices which form the patches or patches of the superimposed surface. The vertex count affects the shape and number of new patches. The pseudocode in Appendix 2 indicates how this count is performed.

A characteristic of this problem is that there can only be a maximum of eight valid vertices created by the intersection of two four-sided patches. The portion of code which creates the patches follows a connect-the-dot methodology. For this reason, the order of the patch vertices becomes important. Where only three vertices are present in the list, the sort routine described in Sections 5.6 and 5.7 can be skipped.

5.5 *Establishing a Vertex Sort Key*

Now that a list of valid vertices has been created, it is necessary to determine the order by which they will be evaluated for the creation of patches. In order to avoid the creation of overlapping or twisted patches, an ordering for the vertices must be established such as that suggested in **Figure 43**. The only exception is the case in which only three vertices are contained in the list because the points are always in the correct order. Where fewer than three vertices exist in the *Vertex List*, it is impossible to create a new patch.

The sort is conducted in two phases. First by determining a baseline reference plane and then by measuring the positions of the vertices relative to this plane. A cutting plane is drawn between points 1 and 2 in the tempVertexList as shown in **Figure 44**. The plane is formed by means of the cross product of the normal vector of the Patch against which the POI is to be superimposed and the vector formed by linking the two vertices. The plane is drawn perpendicular to the Patch and not the POI because the solution vertices all lie on the Patch and not the POI. Therefore it is important that the evaluation of the points be performed in the context of the plane on which all the points lie. Having determined a normal vector *[a, b, c]* for the reference plane through **Equation 9**, back substitution of one of the vertices can be used to find the *[d]* value required for the plane equation (**Equation 10**).

Once the equation of a reference plane has been determined, it is then necessary to substitute each of the remaining vertices in the list into the new plane equations. The result of this substitution will be a list of Reference Plane Equation solutions of positive and negative values. The sign of the solutions refer to which side of the plane each vertex lies. The goal of the development of the reference plane is to create a situation in which all the solution vertices lie on one side of the reference plane (**Figure 45**). By doing so, Dot Products can the be used to determine the relative positions of the solution vertices. Where one or more negative values are found in the list of Reference Plane Equation solutions, that with the greatest magnitude is selected for use in the formation of a new reference plane. This process continues until no more negative vertex solutions are determined.

$$\begin{aligned} Vector_1 &= V_2 - V_1 \\ &= [x_2-x_1, y_2-y_1, z_2-z_1] \end{aligned}$$

$$Normal_{Patch} = [\ a_N,\ b_N,\ c_N\ ]$$

$$Sort\ Plane\ Normal = Vector_1 \times Normal_{Patch}$$

Equation 9 Derivation of the Cross Product calculation which determines the normal vector of a reference plane used in the sorting of the vertices in the *Vertex List*.

$$V_1 = [x_1, y_1, z_1]$$

$$Sort\ Plane\ Normal = [\ a_{SPN},\ b_{SPN},\ c_{SPN}\ ]$$

$$0 = a \cdot x + b \cdot y + c \cdot z + d$$

$$\therefore d = -a_{SPN} \cdot x_1 - b_{SPN} \cdot y_1 - c_{SPN} \cdot z_1 - d_{SPN}$$

Equation 10 Calculation of the final coefficient required for the plane equation of the new Reference Plane.

5.6 *Sorting the Vertices*

Having now developed a Reference Vector, the next step in the development of patches from vertices is to arrange the vertices in order. This is done by means of Dot Products as shown by **Equation 11**.

$$\textit{Reference Plane Normal} = \left[\, a_{RPN},\ b_{RPN},\ c_{RPN} \,\right]$$

$$\begin{aligned} \textit{VertexVector} &= V_2 - V_1 \\ &= \left[\, x_2 - x_1,\ y_2 - y_1,\ z_2 - z_1 \,\right] \end{aligned}$$

$$\textit{Reference Plane Normal} \cdot \textit{Vertex Vector} = \left\| \textit{Reference Plane Normal} \right\| \left\| \textit{Vertex Vector} \right\| \cos\theta$$

Equation 11 Dot Product calculation in which the angles are determined between the reference plane and vectors formed of the vertices to be sorted. This calculation is the mathematical aspect of the model shown in **Figure 46**.

The dot product of two vectors results in an angle from 0 to 180 degrees (or 0 to π radians). The determination of a reference plane described in the previous section was implemented because the angle between vectors cannot be determined through 360 degrees — hence it was impossible to distinguish between angles on one side and the other of the Baseline Reference Vector. From the angles between each of the vectors as shown in **Figure 46**, the vertices can be sorted into an order acceptable for the creation of patches.

5.7 *Creating Patches*

Having now ordered the vertices which comprise the new patches, the creation of the patches is now a simple process of connect the dots (**Figure 47**). The algorithm works from the newly-ordered list of vertices and begins assigning these points to the vertices of patches. Pseudocode for the patch creation process is shown in Appendix 2.

Patches are restricted to a maximum of four vertices. Therefore, every four vertices the algorithm assigns, the current patch is completed and a new patch is begun, building on the last edge of the previous patch. Because there can only be at most eight valid vertices, no more than two patches will ever be created by this subroutine.

5.8 *Check Patch Orientation*

The last step in the creation of the new patch(es) is a verification of its orientation. For this, a Dot Product such as that in **Equation 12** is calculated between the normal vectors of the new patch and that superimposed by the new patch. The value of the solution indicates whether the patch faces inward or outward relative to the object of space being created.

$$\textit{Patch Normal} = [\, a_1,\ b_1,\ c_1 \,]$$

$$\textit{New Patch Normal} = [\, a_2,\ b_2,\ c_2 \,]$$

$$\textit{Patch Normal} \cdot \textit{New Patch Normal} = \|\textit{Patch Normal}\| \|\textit{New Patch Normal}\| \cos\theta$$

$$\therefore \cos\theta = \frac{a_1 a_2 + b_1 b_2 + c_1 c_2}{\sqrt{a_1^2 + b_1^2 + c_1^2}\ \sqrt{a_2^2 + b_2^2 + c_2^2}}$$

$$\textit{where}\ -1 \le \cos\theta \le 1$$

Equation 12 Dot Product calculation to determine the orientation of the new patch relative to the POI Prism side.

5.9 *Finish the Patch List*

Two tasks remain following the completion of the new patches for this particular intersection. The first task involves the repetition of the algorithm described in this chapter until all the patches in the *Solutions for Patches* table have been evaluated and superimposed. Typical output for the example which was introduced in Chapter 4 is shown in **Figure 48**.

The second is one of housekeeping in which the tables dealing with adjacency and plane equations are updated to reflect the new patches. However, this step cannot be completed until the patches on the adjoining faces have been created as will be described in the next chapter.

Figures Pertaining to Chapter 5

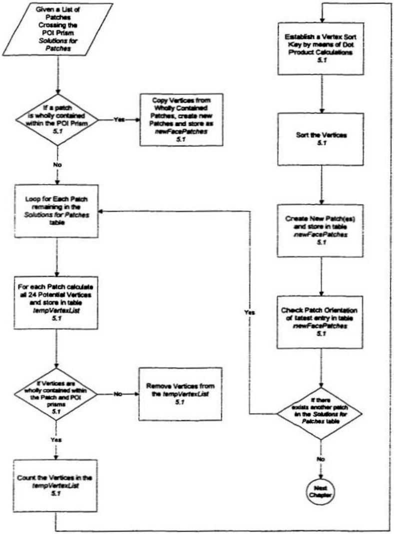


Figure 40 Flowchart of the algorithm which superimposes one surface on another.

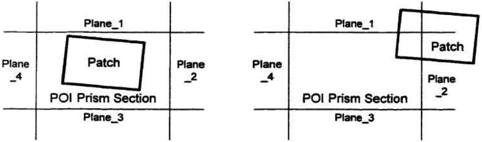


Figure 41 Examples of patches which are wholly contained and partially contained within the POI Prism.

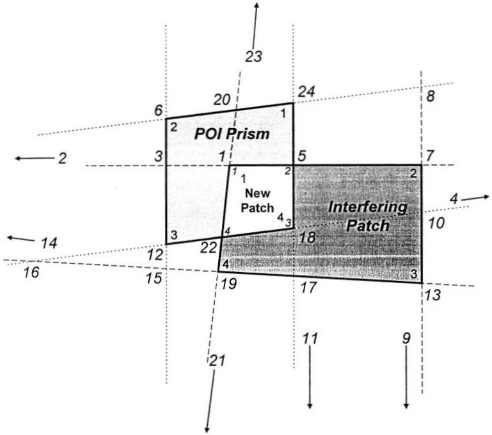


Figure 42 A depiction of two overlapping patches. The planes which form the patches are shown in dashed lines with each of the 24 potential vertices. The four vertices which form the new patch are distinguished from the remaining 20 because only these are wholly contained within both the Patch Prism and the POI Prism.

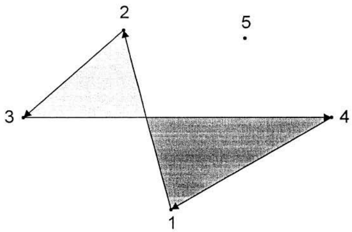


Figure 43 Given a random set of patches, it is often difficult to determine the best way to construct new patches. The Bow Tie-shaped patch shown in this figure is an example of a patch which might result when the order and orientation of the vertices are not taken into account when developing a new patch.

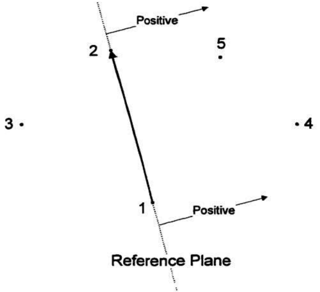


Figure 44 A list of vertices can be sorted by use of a reference plane and vertex substitution. The vertices are coplanar and lie on the Patch Plane. The reference plane is formed by the cross product of the equation of the Patch Plane and the vector formed between the first two vertices in the list. Since Vertex 3 in this figure lies on the negative side of the reference plane, it will be necessary to construct a new reference plane.

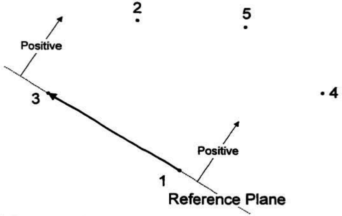


Figure 45 This figure shows the reference plane moved so that it now passes through Vertex 3. By doing so, all of the vertices in the list now lie either on or on the positive side of the reference plane.

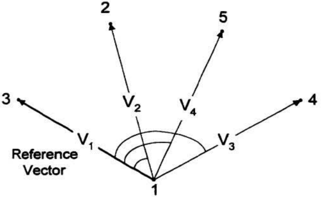


Figure 46 Once the reference plane has been determined, Dot Products can be used to sort the vertices. The Dot Product is taken between the vector formed by the reference plane and similar vectors formed from the contents of the *tempVertexList* table.

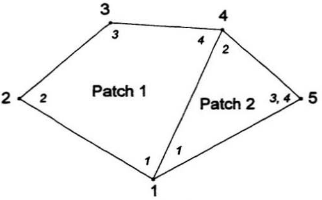


Figure 47 Once the vertices have been sorted, it is a simple process of connecting the dots to properly create the new patches.

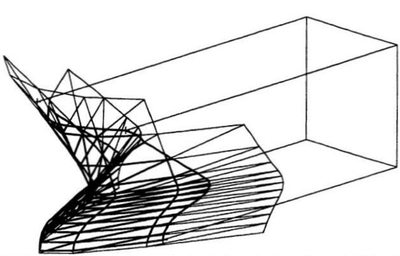


Figure 48 This image builds on the two ship images introduced in Chapter 4. Using the process described in this chapter, the model has projected new patches onto the hull boundary. Both the boundary and the new patches are shown and can be differentiated by the line formed by the POI Prism.

Table Pertaining to Chapter 5

Space_ID
Patch_ID
POI Patch_ID

Vertex1
V1_POI
V1_Plane1
V1_Plane2
V1_Plane3
V1_Plane4

Vertex2
V2_POI
V2_Plane1
V2_Plane2
V2_Plane3
V2_Plane4

Vertex3
V3_POI
V3_Plane1
V3_Plane2
V3_Plane3
V3_Plane4

Vertex4
V4_POI
V4_Plane1
V4_Plane2
V4_Plane3
V4_Plane4

InOrOut

Table 13 Field headings for the *Solutions for Patches* table generated in the previous chapter.

6

Constructing Adjacent Sides

The previous two chapters have dealt with the superposition of spaces. Chapter 4 described a process in which the patches of a neighbouring object can be identified and isolated from the rest of the dataset. Chapter 5 built on this by describing a means by which the Patch of Interest (POI) could be superimposed on these patches.

Following the same progression, this chapter describes the means by which these new patches are tied into the mesh of the new Object. This process involves the identification and sorting of vertices which lie on the boundary of the POI prism, and the creation of a mesh which lies against the sides of the POI prism. The need for the creation of this side mesh is based on the principle that objects created in Semi-Solids are formed by closed meshed surfaces. In order for these meshes to be valid, for each patch edge there can only be one adjoining patch edge. The difference between invalid and valid patches is illustrated by **Figure 17**, **Figure 50** and **Figure 49** respectively.

The process begins by finding the vertices which lie on the plane in question. These vertices are then sorted using the adjacency information of the patches created by the algorithm described in Chapters 4 and 5. New side patches are developed by means of rays or vectors which extend from an anchor point to each of these vertices. Steps are taken to encourage reasonably shaped patches, and to deal with situations in which the rays overlap boundaries. A flowchart depicting this algorithm is shown in (**Figure 51**) through (**Figure 62**) inclusive.

6.1 *Determining the Vertices*

Of the steps in the construction of side patches, this step is the most simple. Given the list of new patches and vertices created by the algorithm described in Chapter 5, it merely collects those vertices which, when substituted into the equation of a side of the POI Prism, yield a solution equal to zero. That is, it finds only those vertices which lie exactly on the plane.

Pseudocode for this section can be found in Appendix 2. It performs the substitution of vertex coordinates into the prism plane equations for each of the four prism sides.

6.2 *Creating an Ordered Vertex List*

The previous step gives the algorithm a means of distinguishing the vertices coincident with one side of the POI Prism from those coincident with another. The creation of new patches requires additional vertex sorting before the algorithm can consistently create valid patches. This involves sorting the vertices in the list into the order these vertices would be encountered were one to move from one prism edge to the other (Surface A to Surface B in **Figure 63**). This is analogous to a child's connect-the-dot puzzle. The previous step has identified the dots, this step numbers them.

The sort involves three steps. First the vertices of the POI Patch must be removed from the list of vertices identified in the previous step. In so doing the algorithm effectively 'drops' the POI patch definition in favour of the new patch created for the adjacent surface. At the same time, the two remaining vertices of the original POI Prism side are renumbered to become the first and last points of the *Vertex List*. **Figure 63** and **Figure 64** show the deletion of the POI Patch in favour of the new patches, and the renumbering of the vertices from 1 and 4 to 1 and 6.

The next step is to establish a sort key by linking the five remaining patches of the new object to the three new patches as shown in **Figure 64**. Linking patches A and C requires identifying the shared vertex by substitution of all the vertices into the equation of plane A (the top of the POI prism). The result of this substitution is the identification of Vertex 2 as shown in **Figure 64**.

Although uncommon, it is possible that more than one vertex exists which meets this criterion. This would require the adjacent surface to contain a switch-back or hollow such as that shown in **Figure 65**. Where more than one vertex is found which meets this criterion, distance from Vertex 1 is used to select the appropriate point. This distance can be easily determined from the coordinates of two vertices using the formula in **Equation 13**.

$$distance = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}$$

Equation 13 A common distance formula suitable for determining the distance between two three-dimensional points.

Having joined the original patches in **Figure 64** to the three new ones, the adjacency properties of the patches can be used to order the remaining vertices. By checking each of the new patches for that which contains Vertex 2, patch C can be identified. By comparing the vertices of patch C to those in the vertex list, the third vertex can be established. Similarly Vertices 4 and 5 can be found by the same process of the identification of patches and shared vertices. Pseudocode for this step is shown in Appendix 2. The principle of the algorithm is to find a patch and use its properties to find the next patch and its vertices. For the purpose of efficiency, each identified patch can then be removed from the patch list so that fewer patches need be searched in the next iteration.

For example, in **Figure 66** Vertex 1 is taken as one of the vertices of the new object. Vertex 2 is found at the junction of two adjacent POI prism sides (Surface A and the surface facing the reader). Since no other vertex lies on this line, use of a distance criterion is unnecessary to establish a point as Vertex 2 in the context of the sort. The list of new patches (Patch 1, Patch 2 and Patch 3) is then searched for that which contains Vertex 2 (in this case Patch 1). The remaining vertices in the list are then checked against the vertices of Patch 1 to determine Vertex 3. The algorithm then removes Patch 1 from the search list and checks for Vertex 3 among the remainders (Patches 2 and 3). The process continues until there are no new patches and the sorted list is completed by the addition of the other vertex of the new object (Vertex 6).

6.3 *Calculating Angles*

Unlike a child's connect-the-dot game, the goal here is to create a valid mesh through each of the vertices. Neither crossed patches nor concavities are acceptable in valid patches. The angles measured between the vectors formed by adjacent vertices can be used to identify potentially invalid patches prior to their creation (**Figure 67**). Angles are determined by means of the Dot Product formula. Unfortunately, the Dot Product yields an angle between 0 and 180 degrees where an angle on a 360 degree basis is required. The reason for this requirement will become apparent in the next section.

To distinguish between Dot Product results which are less than 180 degrees and those greater than 180 degrees, a reference plane is used similar to that described in Chapter 5. Figure 68 shows three vertices to which the algorithm applies a Dot Product to calculate the interior angle. The plane shown is created by means of the Cross Product between the normal vector of the plane on which all three vertices lie, and the vector formed by Vertices 1 and 2. The normal of this new plane points *into* the new patch. Substitution of Vertex 3 into the equation of the new plane will give a result which suggests that it is either above or below the new plane. Where Vertex 3 lies below the new plane, the angle should be considered to be exterior and hence is calculated by subtracting the angle found by the Dot Product from 360 degrees. Similarly, where Vertex 3 lies inside the new plane, the angle should be considered to be interior and can be taken directly from the Dot Product calculation.

Pseudocode for this section is in Appendix 2. Computation time can be reduced by retaining the patch equations instead of recalculating them for each iteration.

6.4 *Creating Patches*

Having now created an ordered list of vertices and determined the angles between the vertices, the algorithm can begin to construct the new patches. All new patches are created from a specific anchor point. In the attempted construction of the first patch, the first vertex in the sorted list is used as anchor. The new patch is created by *walking* around the vertices which form the new boundary. Thus, Vertex 1 of the new patch is the first vertex in the *Vertex List*. The vertex which will be assigned to Vertex 2 of the new patch will be the second vertex in the *Vertex List*. At this point, a decision must be made about the potential validity of the new patch. In general, a valid patch will be formed if its vertices form a Convex Hull. A Convex Hull is a theoretical boundary passing through each member of the enclosed set using only convex curves. If there is an internal angle within a four-sided patch which is greater than 180 degrees then a concavity exists in that patch and the Convex Hull property is violated. **Figure 69** shows such a concavity and is contrasted by the valid mesh element in **Figure 70**. Steps in the algorithm can be saved by checking these angles as the algorithm creates the new patch and it was for this reason the angles were determined in the previous step of the algorithm.

If the angle at Vertex 2 is less than 180 degrees then the new patch can continue. In instances where the angle is greater than or equal to 180 degrees as in **Figure 71**, there is a potential for a concavity in the new patch — a situation which is considered invalid.

Where an invalid situation is found, the algorithm discards this patch and notes that the anchor position must be changed in order to create a new patch (**Figure 72**). In the case in which the angle at Vertex 2 is less than 180 degrees the algorithm continues to walk through the ordered *Vertex List* seeking the third vertex of the new patch.

Continuing to the next vertex in the new patch, a decision must again be made based on the angle found at the current vertex. This time, instead of questioning the potential for the creation of a patch, the algorithm decides if the patch will contain three or four sides. If the angle at Vertex 3 is less than 180 degrees then the new patch can be attempted with four sides. Where the angle at Vertex 3 is greater than or equal to 180 degrees, the new patch will be invalid because of a concavity.

In cases where a four-sided patch is created, a final angle is calculated between the vectors formed between Vertices 1 and 2 and Vertices 4 and 1. Such a case appears in **Figure 73**. Should this angle prove to be greater than or equal to 180 degrees, the fourth vertex is dropped and a three-sided patch is attempted.

6.5 *Interference Checking*

Having created a four-sided patch, the next step is to ensure that the new patch does not interfere with any other patches. To this end, a plane equation is determined from the vector between Vertices 1 and 4 of the new patch and the normal vector of the current side of the POI Prism. Using this equation, the remaining vertices in the *Vertex List* are checked to ensure that they do not lie inside this plane.

If any vertex lies inside the plane then the algorithm assumes that a four-sided patch is invalid. Taking **Figure 74** as an example, Vertex 4 of the new patch is set equal to Vertex 3, and interference checking is performed again. Interference checking takes place in exactly the same manner as before — create a plane using a vector between points Vertex 1 and 4 and check the remaining vertices in the *Vertex List*.

Although the number of sides of the patch in **Figure 74** was reduced to three, **Figure 75** still shows an interference. A· a result, this patch cannot be completed. It is instead discarded, and a flag is set to indicate that the anchor must be moved. Where no interferences are found, the new patch can be considered to be complete and can be stored.

Once a new patch has been completed, the *Vertex List* is updated by removing vertices. In the case of a four-sided patch Vertices 2 and 3 would be trapped by the new patch such that they could not be used in any additional patch construction. For this reason, these entrained Vertices would be removed from the *Vertex List*. Similarly, in the case of a three-sided patch, Vertex 2 would be removed. This step makes continued walking through the *Vertex List* possible, greatly facilitating the creation of the remaining patches.

6.6 *Anchor Points*

As already outlined, patches are created using a connect-the-dots approach in which the algorithm walks through an ordered *Vertex List*. The first vertex of each new patch is considered an anchor point and is shared by more than one patch whenever possible. In Semi-Solids, two anchor points are used which correspond to the beginning and end of the *Vertex List*. To distinguish between them, nautical definitions can be used such that 'anchor' refers to the first vertex in the *Vertex List*, and 'kedge' refers to the last vertex in the *Vertex List*. Once a patch has been created from the anchor, the algorithm shifts its focus to the kedge point and attempts another patch by walking backwards through the *Vertex List*. The use of anchor and kedge points has been made to encourage more regular patch shapes instead of slivers as might be created in the example in **Figure 76** and **Figure 77**.

If a patch cannot be created from a particular anchor point, the anchor point is moved to the next vertex in the list. For example, if no patch can be created using Vertex 1 as an anchor point, the algorithm then assigns Vertex 2 to be the anchor. Movement of anchor points is considered to be a full move, and therefore the algorithm changes sides again, in the hope that this will encourage new patches to originate from the original patches.

6.7 *Meeting The Other End*

The algorithm tracks which vertices have been reached from either end of the list. When the two ends meet, the algorithm assesses the number of vertices in the list and stops when only two vertices remain. Where more than two vertices remain in the list, the algorithm returns to the first vertex in the list and begins the process again, this time working with the remaining vertices in the list.

6.8 *Checking Normals*

Since the algorithm creates patches from either end of the list, the order of points will be inconsistent for the new patches. The normal vector of these patches should all be the same and be oriented towards the exterior of the object. Just as described in the previous chapter, correction of the patches can be made by simply exchanging Vertices 2 and 4 of each incorrect patch (**Figure 78** and **Figure 79**). The verification process is carried out by determining the two vectors, performing a cross product, and comparing the result to the intended surface normal using a Dot Product calculation.

6.9 *Examples*

Figure 80 shows a typical output for the algorithm described in this chapter. The effect of the anchor points on the shape of the patches is evident.

Unlike steps of the Semi-Solids algorithm, the material presented in this chapter forms a series of nested loops and has not followed a linear pattern either in execution or in description. For the purpose of clarity a robust example has been solved step by step in the hopes that this might provide the reader with a more clear understanding of the algorithm. Found in Appendix 3, the example assumes that the Vertex List has already been updated and sorted. As suggested by the ship example in **Figure 80**, situations as complex as the one shown in the example are unlikely in the majority of ship problems. The algorithm described in this chapter derives surface meshes for each of the four sides of the prism. The ship example is shown with the POI and the four prism sides completed in **Figure 81**.

6.10 *Potential Improvements*

The formation of large regular patches is a desirable goal of this algorithm and the method of alternating anchor points described in Section 6.7 is one means by which this can be encouraged. A second means might be to form a patch using the start and end points of the vertex list. However, where the adjacent surface is relatively flat this encourages small sliver-like patches. Perhaps some sort of optimization could be added to minimize the number of patches to form regular patches through the evaluation of the interior angles of each patch, and to encourage the development of similarly sized patches. Unfortunately there will be instances in which the vertices fail to form a convex hull, thereby making less predictable the number of new patches.

An alternative approach might be to try different initial anchor points. One could also attempt to create triangular patches for the initial patch and/or additional patches. Unfortunately, where there are many vertices such as in the case of the example in Appendix 3, this evaluation may be time consuming as the algorithm explores the many potential patch configurations.

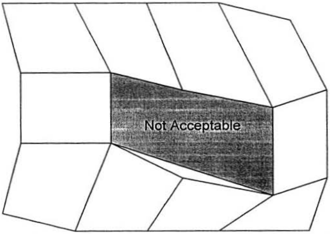


Figure 49 A depiction of an invalid mesh element. The element violates meshing rules because it has four sides while adjoining five other patches.

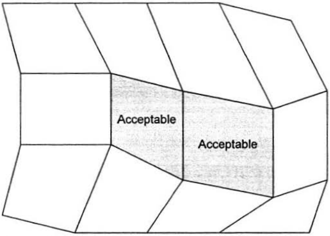


Figure 50 A depiction of the same mesh region, this time validly defined by the use of two new mesh elements.

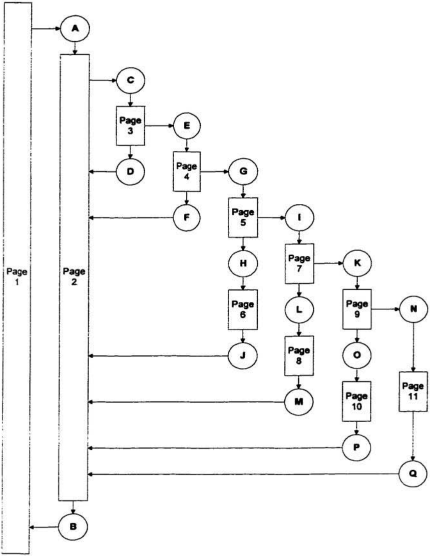


Figure 51 This sheet is a key which shows the relationship of the flowchart pages shown in the series of figures which follows.

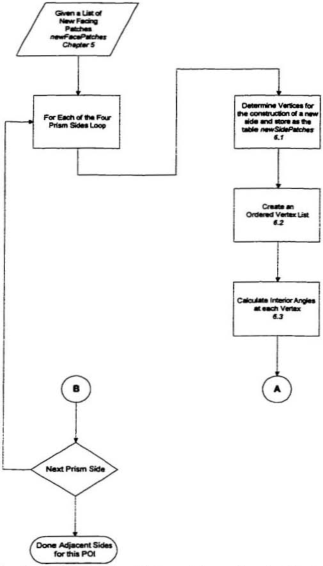


Figure 52 Algorithm for the Construction of Adjacent Sides — Page 1. The characters in the connector symbols refer to parts of the algorithm on other pages.

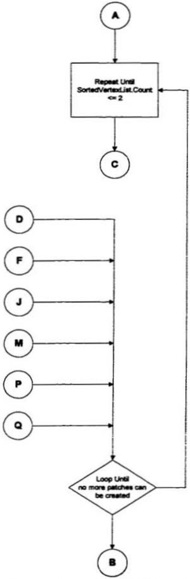


Figure 53 Algorithm for the Construction of Adjacent Sides — Page 2.

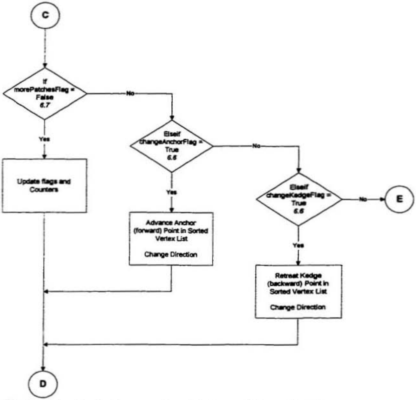


Figure 54 Algorithm for the Construction of Adjacent Sides — Page 3.

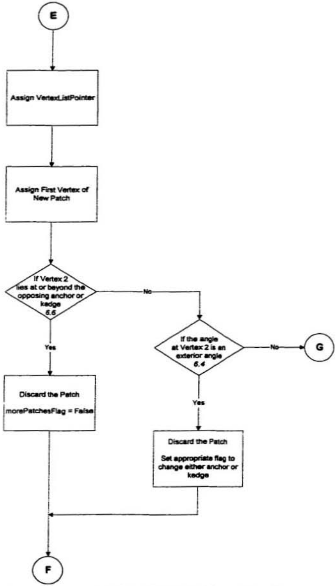


Figure 55 Algorithm for the construction of adjacent sides — Page 4.

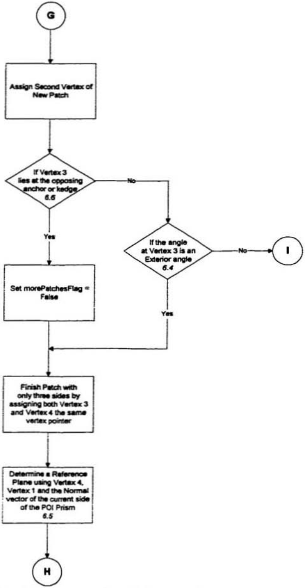


Figure 56 Algorithm for the construction of adjacent sides — Page 5.

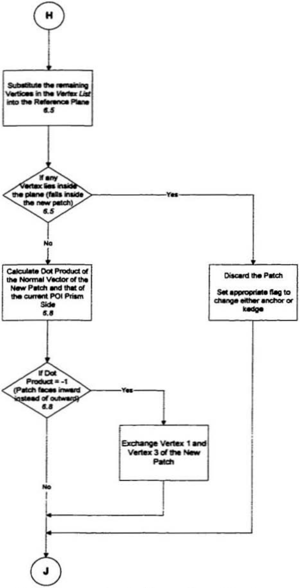


Figure 57 Algorithm for the construction of adjacent sides — Page 6.

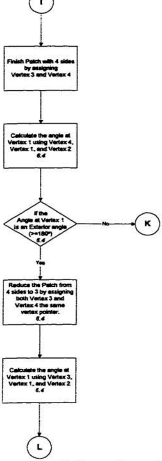


Figure 58 Algorithm for the construction of adjacent sides — Page 7.

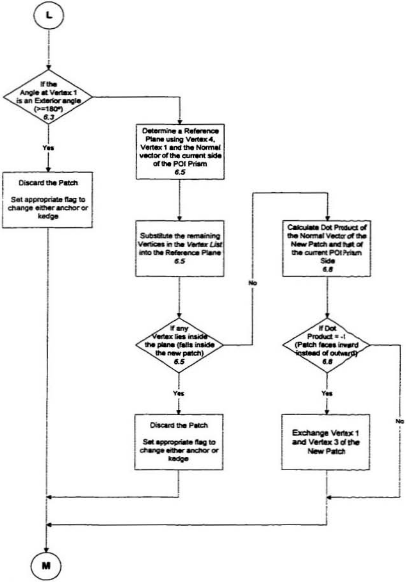


Figure 59 Algorithm for the construction of adjacent sides — **Page 8.**

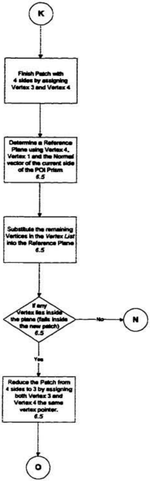


Figure 60 Algorithm for the construction of adjacent sides — Page 9.

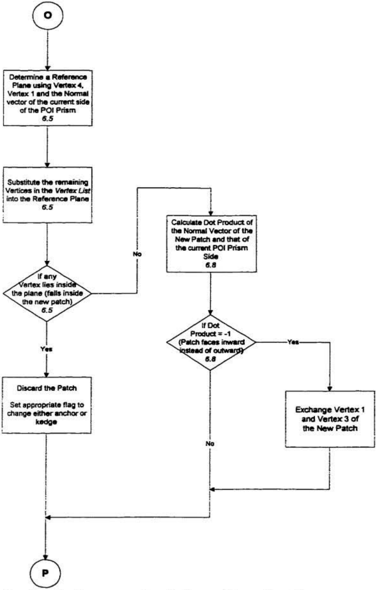


Figure 61 Algorithm for the construction of adjacent sides — Page 10.

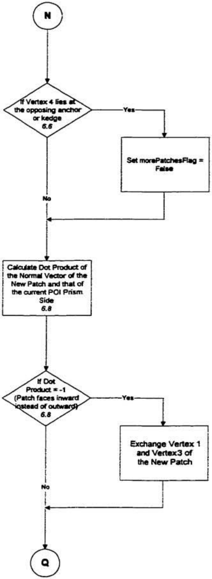


Figure 62 Algorithm for the construction of adjacent sides — Page 11.

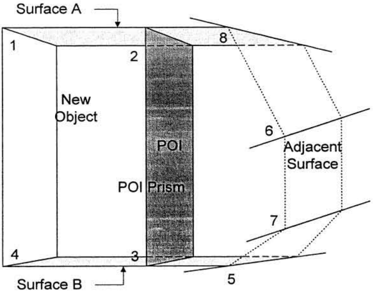


Figure 63 An example of the problem of vertices which define surfaces which adjoin those which were created in the code of the previous chapter.

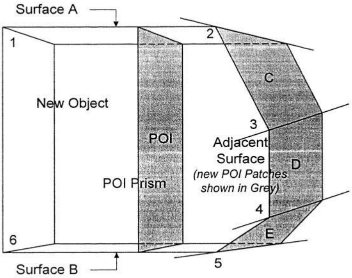


Figure 64 This figure is identical to the previous one except that Vertices 2 and 3 have been dropped from the potential list of vertices for the surface. The POI remains in the figure as a reminder that the vertices have only been removed relative to the surface which faces the reader.

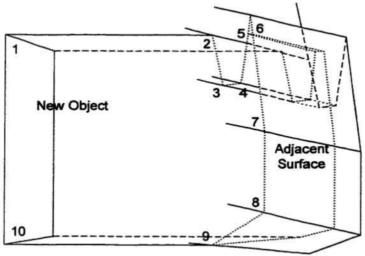


Figure 65 This figure shows a case in which the sort key described in the next section might fail because more than one vertex lies on the same line (passing through Vertices 1, 2, 5 and 6). The distance from the POI can be used to address this unusual case.

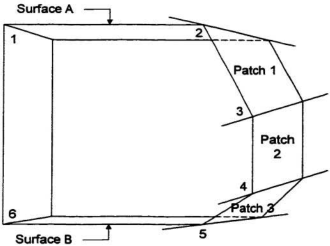


Figure 66 The basic figure showing the vertices of the surface without the presence of the POI.

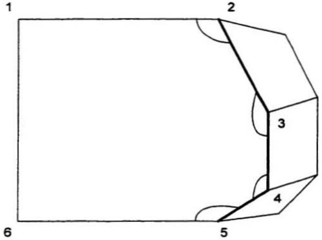


Figure 67 Figure showing the interior angles found between edges formed by the vertices of this surface.

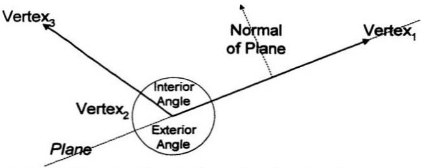


Figure 68 Detail of the previous figure showing interior and exterior angles at a vertex.

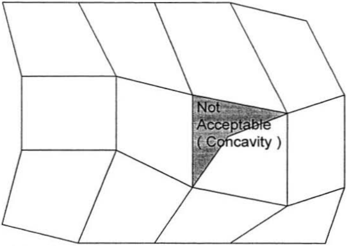


Figure 69 A depiction of an invalid mesh element. The vertices of each element should form a convex hull. This is not true in this case and is evidenced by the concavity shown in the figure.

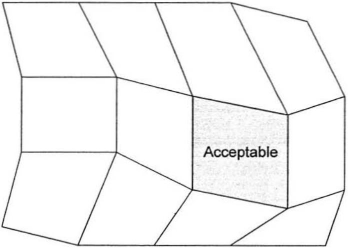


Figure 70 This figure shows the same mesh as in **Figure 69** but with valid mesh element highlighted for contrast. The element is valid because its vertices form a Convex Hull. A property of the Convex Hull is that none of its exterior angles exceed 180°.

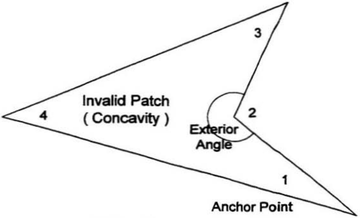


Figure 71 The development of this invalid patch could have been prevented by noting the exterior angle at Vertex 2.

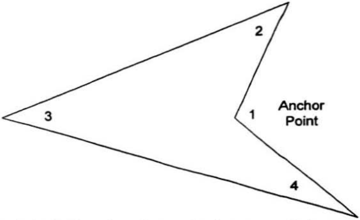


Figure 72 Patch showing the anchor point moved to the next vertex in the potential new patch. Although the four-sided patch is still invalid, a valid three-sided patch is now possible.

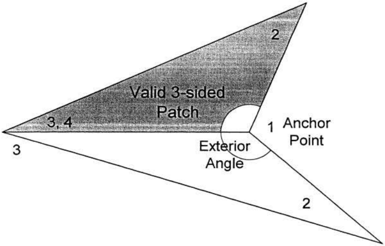


Figure 73 In this case the patch contains an exterior angle at Vertex 3. A decision can be made at this point to limit the patch to a valid three-sided shape.

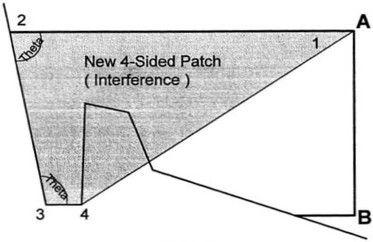


Figure 74 The newly-created patch shown in this figure is invalid because it crosses a boundary formed by the vertices of the *Vertex List.*

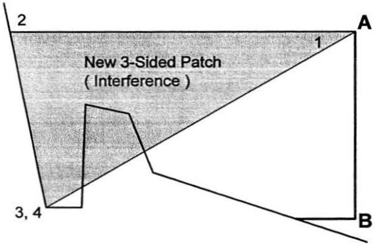


Figure 75 Building on the previous figure, the algorithm attempts to create a valid patch by dropping one of the four vertices thereby forming a three-sided patch. Once again, the patch is invalid because one of its sides violates the valid region defined by the *Vertex List.*

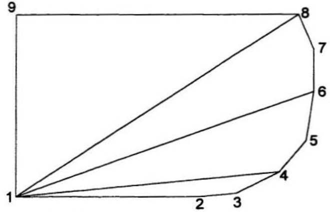


Figure 76 An example of patches which radiate from a single point. The figure is intended to demonstrate the sliver-like form of the newly-created patches.

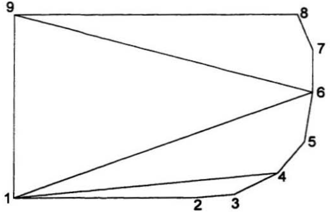


Figure 77 Similar to the previous figure, this figure shows that by alternating patch creation origins (Anchor points), patches which are more regular or *square* can be created.

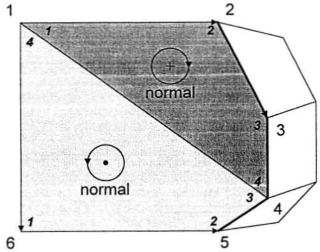


Figure 78 Newly-created patches in which one patch faces outward instead of inward.

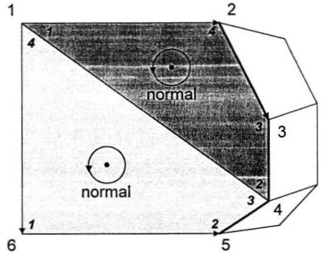


Figure 79 A depiction of the same patches, but with Vertices 2 and 4 exchanged on the invalid patch. The exchange makes it valid because it faces in a direction consistent with its neighbours.

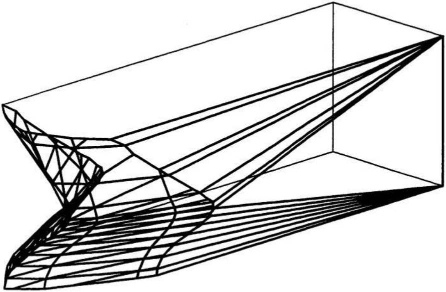


Figure 80 Building again on the ship example introduced in Chapter 4, this figure shows the construction of patches linking the back surface of the new object and the projected surface which replaced the POI.

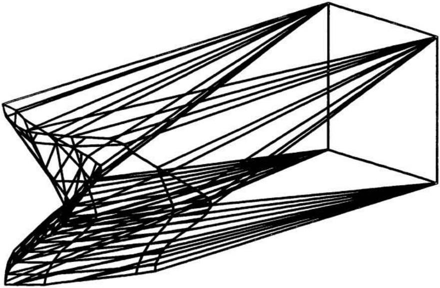


Figure 81 The same image as in **Figure 80**, showing new patches on all four of the POI prism surfaces.

7

Representation Conclusions and Future Work

The introduction of a three-dimensional representation format for Facility Layout problems was prompted specifically by the needs of Naval Architects who require layouts to reflect the compound curvatures of their hull forms. By moving to a 3D representation, the sophistication of models can increase significantly and models will be more adaptable to ship hull forms. Summarized in **Table 14**, the Semi-Solids formulation introduced in this study differs in many respects from the 2D Block representation traditionally used for Facility Layout problems.

Although Semi-Solids has been described in this thesis by means of detailed pseudocode, the algorithm has only partially been implemented. In an effort to save time, it was recognized that the concepts of Semi-Solids, expressed in detail, would be sufficient for the requirements of this project. A true implementation would employ modern programming environments and specialists capable of achieving their full potential for a fast and accurate execution.

Chapters 4, 5 and 6 have described the mechanics of the automated manipulation of objects through the Semi-Solids formulation. The process was examined in detail for a single face of an initially six-sided object and can be generalized as three distinct steps:

- identification of neighbouring objects
- update of surfaces which lie flush to neighbouring surfaces
- update of surfaces adjacent to the updated surfaces

The process is completed by examining the remaining faces of the object until every patch, new or old, has been updated to reflect its surroundings.

While Semi-Solids is well able to address the problem of *fit*, the algorithm has the potential to create unnecessary patches. Unfortunately, the simplification of the meshed surface of an object is difficult to resolve because of the adjacency rules. The impact of this problem is impossible to evaluate without Semi-Solids being operational, but is likely proportional to the number of patches against which an object is being placed. Particularly for interior objects, boundaries will tend to be square and simple, reducing the impact of this problem. **Figure 82** builds on **Figure 83** and shows the extra patch which might result.

This project has yet to examine the mechanics by which Semi-Solids can be applied, other than to suggest that knowledge-based systems and fuzzy-set variables can be used to encourage reasonable solutions in situations in which infeasible solutions may be found. The material presented in Chapter 8 should begin to remedy this omission. However, without Semi-Solids or some other similar representation for the spatial data one cannot begin to build an acceptable, much less effective, Facility Layout Algorithm. The remainder of this chapter discusses areas for future consideration and effort towards what amount to the bricks and mortar of a new Facility Layout Algorithm.

7.1 Literature Review of IEEE Materials

The Semi-Solids formulation proposed in this project was created as a response to inadequacies in the Block Layout approaches currently employed by Industrial Engineers. However, advances in computer graphic models suggest that it may not be the only representation format which could be used for this problem. For this reason it is recommended that any future work include an extensive search in the literature of the Institute of Electrical and Electronic Engineers (IEEE). Although almost 200 references were reviewed over the course of this project, the emphasis was placed on marine-related topics. Because the IEEE publications were only briefly surveyed, it is possible that the Semi-Solids formulation has already been developed. However, on the surface, it appears that Electrical Engineering tends to approach network problems using a 2D format and is therefore fundamentally different from the 3D model described here. It is also possible that a superior representation has been developed as texts dealing with interactive computer graphics show the depth and rapid evolution of this field **[55] [56]**. However, the continued emphasis on interactive models suggests that references to automated representations may be few and far between. In addition, the Electrical Engineering problem of Very Large Scale Integration (VLSI) shares many attributes with the problem of Facility Layout and ideas and solutions for *ShipArrT* may be found in the publications on this topic. For example, the corridor and services routing problem is similar to the power and data lines within a integrated circuit.

7.2 *Complete Coding for Semi-Solids*

Future work must pursue the completion of the code for Semi-Solids as well as the optimization of the algorithm to reduce computation time. In the form presented in Chapters 4, 5 and 6, coding should be a relatively quick task, but there is considerable room for taking advantage of aspects of the database environment to reduce computation time. To this end, Structured Query Language (SQL) and the query functions of Microsoft *Access* should be employed wherever possible. For still superior performance, implementation in languages such as C or Assembler could potentially reduce run times although poorly written routines may impact the algorithm's performance as much as the efficiency of the basic algorithms.

Once coded, performance testing should take place to determine if Semi-Solids can be reasonably applied to Facility Layout Problems. Because of the high number of calculations it is expected that the algorithm will appear to be slow in execution. However, the model is significantly more complex than its predecessors, and it is expected that by the time a complete Facility Layout Algorithm has been developed, the speed of computers will have advanced to the point where the additional computation will be unnoticed. This evolution is similar to the evolution of Graphical User Interfaces (GUI), such as Microsoft *Windows*, which have supplanted their text-based predecessors as the processing power of the personal computer has improved.

7.3 *Acquire and/or Code an Octree Model*

Another possible representation for Facility Layout would be the Octree model described in Appendix 1. The advantages of Octrees suggest that the formulation would provide a useful benchmark against which the performance of Semi-Solids can be evaluated. For this reason it is strongly recommended that an Octree model be developed in parallel to that of Semi-Solids.

Octrees are related to Block Layout in that both are a form of Spatial Enumeration. However, Octrees differ significantly because they are able to subdivide large blocks into smaller blocks to model unusual shapes. Each cube is divided into eight smaller cubes and the process of division can continue until any desired resolution is achieved. Unlike Semi-Solids which exactly models a faceted approximation of a surface, Octrees approximate exact surfaces to a predefined significant figure using a stepped approximation. In both models the complexity of a solution is proportional to the complexity of the boundary of the design space and the shapes being created. An example of the representation of a curve using facets and spatial enumeration is depicted in **Figure 84** and **Figure 85**.

Octrees offer a significant reduction in the complexity of manipulation, but may require a large number of divisions to achieve a resolution which is acceptable to the user. That is, the Octree algorithm may be simple but highly repetitive in contrast to the more complex but less repetitive Semi-Solids formulation. For the purpose of Facility Layout, the capacity of Octrees to model objects with any level of resolution holds considerable appeal. In the course of generating a layout it may become apparent that the current layout will not be an improvement over its predecessor at a stage when the layout model is still quite coarse. The ability to eliminate many potential layouts using a coarse model would significantly reduce the run time of the Facility Layout process.

Like the issues related to computation, there is also a potential for Octrees to be demanding for data storage. For example, an accurate model of a ship hull may require the definition of an enormous number of cubes. However, the actual data element corresponding to each cube is numeric so as to designate which room in the layout of which the cube is a part. This is in contrast to Semi-Solids in which regions are defined by what may be quite few faceted objects, but the definition of those facets requires many bytes of data to be stored.

A potential problem for the use of an Octree formulation is the difficulty of importing and exporting models. While it is relatively easy to use Octrees to model faceted or curved surfaces, it is very difficult to create a meshed surface from an Octree model. Since most output programs such as those for mathematical modelling, rendered graphics and virtual reality require meshed surfaces as input, this will be a critical problem to overcome. The minimum addresses the importation problem because hull forms are currently imported as faceted 3D meshes or as line plans.

7.4 Adapt Semi-Solids for Bicubic Surfaces

A third representation possibility would be to develop a formulation called Bicubic-Solids which builds on the Semi-Solids formulation but replaces the 2D facets of the representation with bicubic surfaces. Intuitively, this means that the definition of each facet will require 16 mathematical coefficients and the 3D coordinates of 16 vertices. This is in contrast to the planar definition of four vertices and four equation coefficients for a comparable four-sided mesh element. Not only does this increase the data storage requirements of a particular model, but also radically increases the computation requirements because of the complexity of bicubic surfaces. Despite these two obvious reasons to discard a Bicubic-Solids formulation, there is potential for a reduction in the number of surface patches required to define a model such as a ship's hull-form. The bicubic definition is able to represent a large region of curvature with a single patch such that the entire hull of a ship might be modelled by few patches, thereby reducing both data and computation demands. Therefore, the trade-off lies between numerous relatively simple calculations versus few highly complex calculations. It is likely that the optimal approach is problem specific.

An additional advantage of moving to a Bicubic-Solids formulation is that the model represents the desired surface exactly, thereby making this formulation the most robust of the three. However, it is also common to experience agreement problems where two bicubic patches intersect, particularly along the intersecting edges of two separate objects.

The mechanics of manipulation of Bicubic-Solids may prove to create more problems than it solves. For example, the projection of prisms described in Chapter 4 becomes virtually impossible where the prism sides are bicubics. It is hoped that future work will quickly determine the feasibility of Bicubic-Solids, but for the purpose of this chapter it is assumed that such a representation can be developed.

7.5 Compare Semi-Solids, Octrees and Bicubic-Solids

Perhaps the greatest difficulty in comparing the performance of the three representations just proposed stems from the fact that the intent is their application in a Facility Layout Algorithm. Although many components of a Facility Layout Algorithm have been discussed in the thesis, an algorithm which links and controls these components does not yet exist. Hence a simple model must be developed such that each representation can display its strengths and weaknesses while the detailed algorithm is being developed.

One such model might take the appearance of the narrow hull of a catamaran similar to that shown in **Figure 86**. For simplicity, the shape of the hull does not change with depth into the page. The blocks shown may either be thought of as initial Octree blocks or as spaces / rooms in the other two formulations. The shape is relatively simple when compared to a complex ship form and, more importantly, its narrow width eliminates the need to be concerned with the relative orientations of the spaces or cubes. It therefore reduces the problem to one of *fit* as opposed to layout and provides a fair basis for the comparison of the different formulations described in this chapter. **Table 15** outlines a set of criteria for comparing the different formulations.

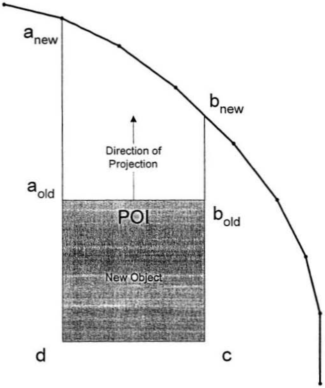


Figure 82 Top view of the process of fitting one object against another. The view shows how the vertex pointers at *a* and *b* are moved to reflect the new vertex positions.

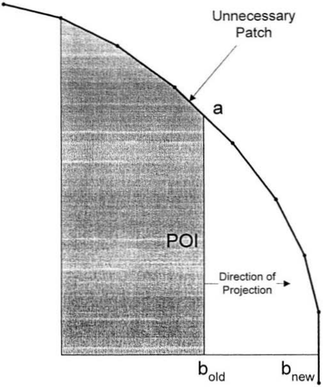


Figure 83 Top view showing how the next projection plane completes the fitting process. The figure also shows how the construction algorithm creates an unnecessary patch. The problem can be much more significant where the bounding mesh is considered in three dimensions.

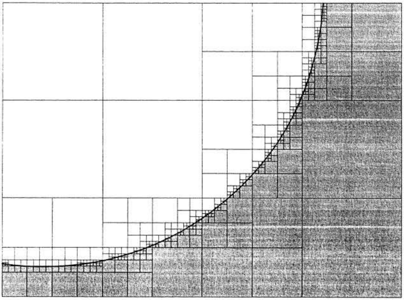


Figure 84 An example of modelling a curve using Quadtrees. Quadtrees are the two-dimensional equivalent of Octrees. There is a rapid increase in the number of squares required to accurately model the curve. Also, while it is simple to approximate a curve by spatial enumeration, it is difficult to create a curve from a series of blocks.

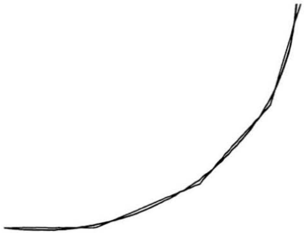


Figure 85 The same curve which was modelled in the previous figure can be described by means of a series of straight lines. The lines correspond to facets in the Semi-Solids formulation. For simple curves such as this, relatively few line segments are required to approximate the curve to the level of accuracy shown.

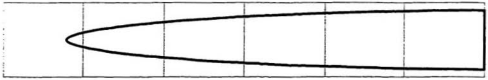


Figure 86 A possible model against which the three potential representation formulations can be applied during the evaluation of their performance. The simple shape extends into the page to provide a boundary for the third dimension.

	Block Layout	Semi-Solids
Definitions	Spatial enumeration of a 2D design space through the assignment of 2D blocks to corresponding grid structure in the design space.	A 3D representation capable of modelling 3D faceted shapes by using planar mathematics to define an enclosed region. The object so defined can be moved without changes in shape, or the shape of the region can be altered plane-by-plane (facet by facet) independently or to reflect adjacent objects.
Advantages	A well-established modelling format which tends to not be computationally demanding. A well-developed class of mathematical algorithms called Quadratic Assignment is the basis of the representation defined by this format.	Both representation and manipulation algorithm are very robust and can form any faceted shape. Makes possible the inclusion of surface attributes such as the position of doors and windows, and such as the weight of wall materials. Three dimensional. Each room is defined as a single object.
Limitations	Two dimensional. Rooms may require many blocks to define. Model accuracy is limited to the resolution (size) of the blocks. Block sizes cannot be varied within the same layout. Blocks cannot accurately represent curved surfaces. Difficult to assign surface attributes such as doors and windows to Blocks.	The complex manipulation algorithm is computationally demanding. Each model requires an enormous quantity of mathematical data. The time required to complete a layout may prove to be unacceptable. May create unnecessary patches which are very difficult to remove.

Table 14 Table comparing the Block Layout representation commonly used for Facility Layout Problems, and the new Semi-Solids formulation which has been proposed to replace it.

Performance	• Speed • Efficiency of Data Storage • Impact of increasing model complexity • Sensitivity to Initial Octree cube size
Accuracy	• Accuracy of Representation • Sensitivity of Accuracy on Performance (increasing the number of facets for Semi-Solids, increasing the resolution for Octrees)
Facility Layout	• Ease by which code can create a shape (try creating the same shapes without the aid of the boundary) • Ease by which a new space can be added to the layout • Ease of Manipulation
Miscellaneous	• Ease by/means of including boundary information - surfaces & materials - windows - doors - services (electrical outlets, etc.) • Ease of visual display of model • Import / Export constraints — especially for Octrees which require a surface mesh to be created from their models • Ease of checking consistency

Table 15 Ideas for evaluation criteria to compare the model representations Semi-Solids, Octrees and Bicubic-Solids.

8

ShipArrT Conclusions and Future Work

This project began as an investigation into computer-aided ship design and, for this reason, requirements specific to the design of ships have been included wherever possible. The software developed for the project has been named *Ship Arrangement Tool* (ShipArrT) in reference to the General Arrangement of a ship. Since a ship's General Arrangement is closely analogous to the land-based problem of Facility Layout, the *ShipArrT* algorithm should be equally effective ashore and afloat.

The decision to modernize the software used for Facility Layout stems from two sources. First, the lack of success of traditional algorithms for Facility Layout can be attributed to the crude manner in which they manipulate spatial constraints, particularly because of the almost universal Block representation. Second, enormous advances have been made in the performance of computers, thereby making possible the use of more complex and sophisticated models.

Just as Semi-Solids has only been partially coded at the time of writing, so does *ShipArrT* remain incomplete. The remainder of this chapter contains suggestions and directions for future work in research and development which should aid in bringing *ShipArrT* to a reality.

8.1 The Representation of Quantitative Data

The goal for any data-oriented problem is to maximize the information available while minimizing its storage requirements. Chapter 3 described a database structure by which quantitative data could be stored and quickly accessed. It was also proposed that a relational database is a suitable environment because it facilitates data manipulation by linking dissimilar data elements to one another, such as a door description and a room dimension. In addition, the relational database greatly facilitates future expansion of the same dataset since the appending of new tables allows the existing records and data structures to remain intact.

Since the size of a dataset is always of concern, particularly for a data-oriented problem such as Facility Layout, the relational database makes possible the sharing of common data elements. For example, if the model of a hotel contains 1000 identical rooms, a relational database makes it possible to store a single room definition. Therefore, each room record in the database need only contain data specific to that particular room (e.g., its location in the hotel) with common data such as the room's contents accessed by means of a pointer to the shared room definition.

While still in the developmental phase, Microsoft's *Access* and *Visual Basic* appear to offer a simple, yet sophisticated, developmental environment. The *Access* database also offers programmers access to many of the program's internal functions such as sorts and queries. Given that there is a large quantity of data associated with a Facility Layout problem and that the data must be accessed many times during the development of a layout solution, the speed by which data can be stored and manipulated is critical.

8.2 *The Representation of Qualitative and Indefinite Data*

Briefly mentioned in Chapter 2 was the potential for using a variation of fuzzy sets to represent qualitative data and ranges of quantitative data. A fuzzy value is usually defined by a magnitude and a *membership function*. The *membership function* is usually, although not necessarily, linear and ranges from 0 to 1. It is intended to provide a measure of the degree to which the value actually is a member of the set of values. Sometimes fuzzy membership functions are described as measures of the degree of possibility so as to distinguish such functions from their statistical counterpart, although in many cases it is difficult to distinguish between the two. In fact, "fuzzy measures are defined by weaker axioms, thus subsuming probability measures as a special type of fuzzy measure**[57]**." "One immediately apparent difference is that the summation of probabilities on a finite universal set must equal 1, while there is no such requirement for membership grades**[58]**."

The variation which was introduced in this project involves the use of a *membership function* to interpret a range of qualitative values. For example, under a fuzzy measure the floor area of a room can be defined by a range of numbers — a minimum, a preferred and a maximum value. The range forms a set of valid potential values for the particular variable, area in this case. This introduces the possibility that a quantitative value can differ from its preferred value so long as it lies in the predefined range. However, it is also desirable that solutions be as close to their preferred values as possible. To this end the membership function can be used to create a penalty which appears in the score of the layout (**Figure 87**). Thus, in terms of the area example, a layout solution in which the area of a room is the room's preferred value receives no penalty. However, a layout solution in which the area of the room is close to its minimum value receives a penalty which increases the score of the layout. Since the goal of the Facility Layout algorithm is to find the layout with the minimum score, the penalty acts to discourage (but not prevent) the algorithm from finding the second layout to be the optimum.

8.3 *Difficulties Associated with Constraints and Data*

As suggested in Chapter 2, traditional Facility Layout algorithms employ a single constraint for the purposes of scoring. However, there are a multitude of constraints associated with Facility Layout and it is desirable to model as many as possible. Therefore, one step which is necessary in a new Facility Layout algorithm is a means by which multiple constraints can be represented and applied to the layout model. The contents of **Table 16** and **Table 17** which were introduced in Chapter 1 show a number of such criteria. In implementation, factors such as services will involve additional variables for calculation and will therefore become more complex.

The use of many constraints introduces three problems in the development of the new Layout algorithm. First, the greater the number of constraints the greater the computational demands of the model — hence the algorithm requires more time to determine a solution. However, as previously noted, increasing computation time is not necessarily critical given that the speed of computers increases daily. Second, there has been little research into the relative significance of various constraints, so it is quite likely that some may be over- or under-valued, thereby affecting the solution layout. The answer to this is to get a new Facility Layout algorithm operational and then perform sensitivity analysis on each of the constraints for a number of different layouts. The results can then be confirmed by experts in the manual solution of such problems. Third, there are often instances in which variables or constraints are in conflict. For example, if a user defines a room in the layout by its floor area and volume using the Fuzzy Sets described in the previous section, it may be that the solution will call for an area which cannot be achieved for a valid volume. The solution to this problem might be best addressed through the use of a knowledge-based/expert system. By developing such a system, the problem of constraints becomes one of defining a set of rules by which preference can be given to particular variables in the event of conflicts.

The problem of constraints can be solved by initially developing a model for a handful of constraints. The model should be similar to the database structure discussed in the previous section such that new constraints can be easily added as the algorithm develops. This will greatly facilitate the addition of constraints such as those related to multi-story layouts. In the future, as the success of multi-criteria algorithms becomes better established, the addition of constraints normally associated with building codes and the rules of regulatory bodies can also be added.

8.4 *Balloon Modelling*

An interesting metaphor for spatial constraints in Facility Layout Problems is a box of balloons[59]. If each balloon represents a space, then simultaneous inflation of the balloons leads to a situation not unlike the layout process. Each balloon would be injected with a quantity of air appropriate for the size of the space it represents. The balloons would experience some changes in relative positions as some became larger than their neighbours. Further, they would also experience a change in volume consistent with the forces applied by the surrounding balloons. Once inflated, all of the balloons would contain air at the same pressure, with some of their numbers larger and others smaller as appropriate for the surroundings and the quantity of air they contain. The equality of the air pressure within all of the balloons is analogous to a system solution. Further, because of the influence of their neighbours, balloons which were intended to be equal in shape will likely vary. And yet, with the system at steady state, the physical dimensions of the balloons will be optimal. The balloon model is therefore a very reasonable representation of how neighbouring spaces can impact on each other in a layout. This balloon model does not address the relative configuration of the balloons, but instead finds only an equilibrium for the spatial interaction.

This balloon concept introduces a interesting approach to the problem of improving the score of a layout. For example, consider the exchange of two dissimilarly sized balloons. Inflating the balloons will lead to a situation in which the large balloon crammed into a small volume will have a high internal pressure and the small balloon in the large hole will have a relatively low internal pressure. The pressures effectively act as a force which push upon and alter the positions and shapes of neighbouring spaces until a new steady state is achieved. It should be possible to determine a measure of this force, and to evaluate/predict its effect on the spaces relative to other constraints, especially their boundaries. The evaluation of the pressures is not dissimilar to a topographic style isobaric map in which the high pressure region appears as a mountain, the low as a valley, and the steady state/optimum is achieved when the map is uniformly level. Weather models or perhaps Finite Element Modelling (FEM) might provide quite interesting ways of evaluating this. If such a pressure-based evaluation can be made, this method will avoid the need for rearranging the whole layout for each improvement attempt. Further, it should be possible to make multiple exchanges (i.e., five or more instead of two or three) thereby greatly improving upon traditional Improvement algorithms. Lastly, a pressure model such as this would make possible interactive manipulation of the layout, since it provides a means by which the layout can be appropriately updated to reflect manual/interactive changes in the position of a space.

8.5 *Problems Associated with Superposition*

Another challenging problem associated with Facility Layout is the need for sharing of space and resources. The difficulties associated with the traditional approach to distance constraints such as pipe networks was introduced in Chapter 2. However, the problem does not just affect services but also spatial constraints in the form of walled and unwalled corridors. The next four subsections discuss problems related to superposition and routing, and suggest ideas and approaches which might contribute to their solution.

8.5.1 Arrangement of Furnishings for Each Room

Just as the layout boundary affects and is affected by its contents, so are the shapes and dimensions of individual spaces impacted by their contents. For this reason, a subproblem would be the valid layout of the contents of each space. Machinery Arrangement has already been a published topic of research, and the problem is the same for any objects including furnishings and cargo.

Using a bedroom as an example, one approach would be to establish a zone of open area around each piece of furniture, much like a corridor. For example, consider the furniture one might find in a bedroom: a bed, desk, chair, wardrobe, and end table. The problem of layout within a bedroom becomes one of maximizing open areas for the room's preferred floor area while maintaining access. However, there are instances in which the objects can share corridor space, or corridor space can simply be neglected. The bed and the end table are one such example of pieces of furniture which do not require an open region between the two of them, nor between themselves and a wall. Further, they can share the open region in front of the table and to one side of the bed. Building on the use of fuzzy sets previously described, it should be possible to define furnishings so that a range of interference percentages can be tolerated by the arrangement algorithm. It should also be possible to increase the significance of the open areas around a piece of furniture in a manner inversely proportional to the unencumbered area remaining. Hence the more sides which are impinged upon by walls and other pieces of furniture, the greater becomes the importance of the dimensions of the corridor leading to the piece of furniture. Ideas such as these find direct application in superposition problems such as corridors and services.

8.5.2 Design of Corridors

Traditional Facility Layout algorithms assume that the area required for corridors has been included in the area definition of each room, and therefore the problem of corridors can be neglected. In practice, corridors present the architect with a superposition problem, one which is largely related to traffic flow. For example, a single room requires a corridor of cross-sectional dimensions appropriate for what will be entering and leaving the room, whether it is humans or five-tonne trucks. When a second room is created adjacent to the first, it is intuitively obvious that the creation of a separate corridor is an inefficient use of space in the layout. Instead, the two rooms should share the single corridor. The next question is, should the dimensions of the corridor be altered to suit the increase in traffic? If the two rooms have equal traffic requirements then should the corridor be doubled in width? How does a change in corridor dimension affect the shape and location of the rooms and their neighbours? The problem is similar to that of furniture arrangement described in the previous subsection and it is likely that many aspects of the solution algorithm can be shared.

8.5.3 Servicing Spaces with Utilities

If one extends the analogy of corridors to the services associated with rooms in the layout, then the problem of corridors is similar to that of a large duct. From the solution of the problem it should be possible to generate a list of the components required for the duct such as dampers (doors), T-intersections and tubing of the appropriate diameter. Further, the same logic can be applied to other services such as potable water, sewage, electricity, etc. The inventory of the components required for services is quite useful for detailed design and because real costs can be attributed to each pipe or wire, thereby leading to highly accurate cost estimates. Also, recall that the layout solution is given a numerical score which could be expressed in terms of cost. Therefore, the true cost of servicing a room can be incorporated into the measures of merit of the layout solution, providing valuable information to the designer.

8.5.4 Routing Problems for Services and Corridors

The routing of services, including corridors can dramatically affect the efficiency of a layout solution because of the potential for wasted space and high service costs. A routing algorithm working in concert with the superposition methods suggested in the previous sections will make a significant contribution in the determination of an optimal layout.

> "Going back to configuration design, and especially the layout of compartments, because I felt it was most neglected, and it was the area of simple design for production which I was working in about four years ago which led me to the idea of space in the compartments of a ship in the comprehensive way (in fact, one stage beyond what the author has done), where the simplifications of arrangement design and hence the reduction in shipbuilders' costs, and the improvement in functionality of all the operational systems onboard — are all improved by simple compartment layouts. The most important being the routing of pipes, cables and trunkings. The reduced number of bends on pipes, to take a simple example, reduces the pressure drop, so that the same functionality and better performance is achieved for less power: or you get better functionality for the same power, whichever you prefer**[60]**."

Fortunately, Mechanical Engineers have made progress towards automated algorithms for piping and ducting of large systems. However, the key remains that of the whole Facility Layout problem — how does one give the computer the means to freely add, delete, size, locate and check the interference of the components of such systems without a means of perceiving or modelling their spatial characteristics.

8.6 Optimization and Facility Layout

There has been an enormous effort applied to the problem of optimization of complex problems. Mathematical programming and search techniques are now well established and have been complimented by non-traditional algorithms such as Simulated Annealing and Genetic algorithms. Combinations of these algorithms guided by expert systems are also becoming prevalent**[61]**. Once the representational problems have been addressed and implemented as suggested in Chapter 8, there is no reason why these modern mathematical approaches cannot also be applied to the problem of Facility Layout. Further, it may be also appropriate to apply some of these methods to sub-problems of the Layout, such as the routing problem described in the previous section.

8.7 Communication of Results

Although not previously discussed in this report a translation routine was developed for the importation and exportation of meshed objects. The importation algorithm reads files written in AutoCAD's Drawing Exchange Format (DXF) and assigns their contents to the appropriate tables and fields in the database. DXF is a common translation medium for many CAD programs. In particular, Autoship Systems' *AutoShip* surface modelling program, which has been made available through the Faculty's computing centre, exports its data as faceted 3D meshes using DXF.

When exporting files from *ShipArrT*, a similar translation program reads the database and creates a file in *.DXF which can then be viewed and edited using AutoCAD. Since the Facility Layout algorithm is intended to be automated, the display of layouts was considered to be unnecessary and superfluous. Instead, layout solutions are exported and viewed from AutoCAD or a similar CAD package.

Each mesh which *ShipArrT* imports is treated as a single object in the database. Thus, the transom of a ship, as a different mesh, appears as a separate object within the database. Similarly, when exporting solutions, the translation program creates a mesh for each Space/Room, which facilitates any viewing, rendering, colouring, etc.

Ideally the user would be able to watch the layout develop, but not only is display computationally inefficient it is also unnecessary in the determination of a solution. The more practical alternative would be to export solution layouts to a third-party Virtual Reality package and offer the architect the ability to *walk* through the layout. The ability to export models for mathematical analysis such as that of Finite Elements would also be valuable. The use of such third-party software for analysis and display makes the program more flexible for users and greatly reduces the programming effort for the project.

8.8 *Criticisms Associated with ShipArrT and Semi-Solids*

8.8.1 Too Much Detail

The concern that *ShipArrT* requires too detailed an analysis for preliminary or conceptual design work can be challenged in two ways. First, the model is unhampered by a lack of information / constraint information. So long as a hull shape can be provided to provide a hull boundary, the model can be run on the basis of simple assumptions regarding the number and classes of spaces require. Second, this criticism is valid if a distinction can be made between preliminary design, conceptual design, detailed design, and production design. However, most new software for Computer Aided Ship Design, particularly those systems used commercially, is increasingly structured to facilitate a beginning-to-end approach to design.

8.8.2 *ShipArrT* Data Sources

In addition to the option of interactive editing of dimensional data during the modelling process, dimensional data can also be predefined by the user or taken from published architectural standards. If the user is content with using such standards then the creation of a new ship need only involve identifying the number of required spaces for each class (e.g., 15 single bedrooms). In addition, it may also be possible to simplify the process even further by using ship types to define interior regions — thus similarly sized ships of a particular type will always have the same number of berths, cabins and recreational spaces.

8.8.3 Consistency of Analysis

In Chapter 1 it was suggested that design analyses should be executed to a consistent depth. For preliminary or conceptual design the General Arrangement model proposed here will be more sophisticated than the other models. However, if the layout algorithm is automated, the impact of its detail will be of little importance other than run-time. As suggested in Chapter 1, a design analysis is only as accurate / optimal as its components — in this instance the overall design will be limited by calculations other than that of the *ShipArrT* algorithm.

8.9 *Summation and Conclusions*

There are years of work to be done towards the creation of a new algorithm for Facility Layout. However, the problem can be divided into a number of steps, each of which can be solved with a high probability of success given modern experience with computer modelling. The premise which underlies any such development lies in distinguishing between busy work and the designer's true thought processes. So long as Naval Architects choose to ignore this difference the evolution and integration of computer-aided design will stagnate. In practice this means moving beyond an interactive approach to design, and instead towards an automated process under the expert direction of the Naval Architect in a manner similar to that suggested by this thesis. Consistency and depth of analysis are critical characteristics of this process, and, as suggested in Chapter 1, the layout problem examined herein is an area in which the profession of Naval Architecture is very weak.

Systems engineering has a two-way relationship with architecture. Firstly, a system will adapt to suit its surroundings in the same way that one adapts to a house lacking a front-hall closet. The system of living in the house alters so that another closet might be used as a replacement or that an additional piece of furniture such as a hall-tree might be introduced. Secondly, for someone designing a new house, the system drives many aspects of the design. For example, if there are three children living in the house then the architecture might include bedrooms for each child as well as a family room. In their own ways, the implementation of each of these relationships are forms of busy work. The design problem, and the true intellectual challenge, lies in the analysis and evaluation of the system itself. There is only so much time in the day and the less busy work the better.

Figure Pertaining to Chapter 8

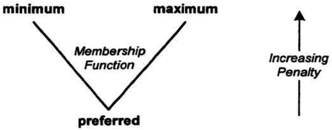


Figure 87 A variation of a fuzzy set in which the membership function takes the shape of a *V* and is used as a penalty function. Applied in the scoring of a layout, the penalty function acts to discourage solutions whose quantitative values differ from the preferred amount.

Constraint	Description
Weight	*room weight is relevant for large buildings and ships*
Traffic	*frequency of people/goods entering and departing*
Vibration and Noise	*vibration or noise created in a room, or the tolerance of a room for vibration and noise*
Services	*electricity, water, sewage, etc.*
Thermal Insulation	*level of, or importance of, insulation for heat or cold from one region to another*
Construction Cost	*cost to assemble and install*
Operating Cost	*cost of maintenance and upkeep*
Access (corridors, stairwells)	*requirements for people and goods beyond the room*
Proximity to exterior	*need for external access*
Adjacency to other spaces	*need to share a wall with another room*
Proximity to other spaces	*need to be close to or far from another room*
Sharing of common spaces	*corridors, washrooms, entrances, etc.*

Table 16 Examples of distance-based layout constraints.

Constraint	*Description*
Size	*size of a space is not necessarily fixed*
Orientation	*orientation relative to other spaces or the boundary*
Aspect Ratio	*shape of a space is likely bounded*
Homogeneity	*a space is not divided into several pieces*
Simplicity	*few corners or jagged edges*
Contiguity	*one wall leads into another on the next space*
Consistency	*similar spaces resemble one another*
Utilization	*no voids, and adherence to fixed structures and boundaries*
Sharing	*efficiency of common spaces such as corridors, washrooms, entrances, etc.*
Accessibility	*corridors, stairwells*
Access	*location of doors, etc.*

Table 17 Examples of spatially-based layout constraints.

References

[1] 1991. "*Le Corbusier*: A Ferry Proposal for the Future", *The Naval Architect*, (The Royal Institution of Naval Architects, London, United Kingdom), November 1991. Page E525.

[2] 1991. "*Le Corbusier*: A Ferry Proposal for the Future", *The Naval Architect*, (The Royal Institution of Naval Architects, London, United Kingdom), November 1991. Page E525.

[3] D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Pages 448-449.

[4] D.K. Brown, 1993. "Naval Architecture", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), January 1993. Pages 43-45.

[5] D.K. Brown, 1993. "Naval Architecture", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), January 1993. Page 44.

[6] D.K. Brown, 1993. "Naval Architecture", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), January 1993. Page 44.

[7] J.P. Hope, 1981. "The Process of Naval Ship General Arrangement Design and Analysis", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), August 1981. Page 34.

[8] J.P. Hope, 1981. "The Process of Naval Ship General Arrangement Design and Analysis", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), August 1981. Page 34.

[9] S. Erichsen, cited in the discussion of:

D.J. Andrews, 1985. "An Integrated Approach to Ship Synthesis", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 464.

[10] D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom).

[11] D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 447.

[12] D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 447.

[13] C.J. Jones. Design Methods, (Wiley Interscience). Cited in:

D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 452.

[14] D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 452.

[15] D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 447.

[16] D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 447.

[17] D.J. Andrews, 1985. "An Integrated Approach to Ship Synthesis", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 99.

[18] G.H. Fuller, cited in the discussion of:

D.J. Andrews, 1985. "An Integrated Approach to Ship Synthesis", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 93.

[19] G.H. Fuller, cited in the discussion of:

D.J. Andrews, 1985. "An Integrated Approach to Ship Synthesis", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 93.

[20] L.J. Rydill, cited in the discussion of:

D.J. Andrews, 1985. "An Integrated Approach to Ship Synthesis", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 91.

[21] Sir R. Baker, cited in the discussion of :

D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 460.

[22] Sir Rowland Baker, cited in the discussion of :

D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 460.

[23] D.K. Brown, 1993. "Naval Architecture", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), January 1993. Page 44.

[24] M.A. Polini, D.J. Wooley, and J.D. Butler, 1997. "Impact of Simulation-Based Design on Today's Shipbuilders", *Marine Technology*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 34, number 1, January 1997.

[25] S.J. Baum and R. Ramakrishnan, 1997. "Applying 3D Product Modelling Technology to Shipbuilding", *Marine Technology*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 34, number 1, January 1997.

[26] S.J. Baum and R. Ramakrishnan, 1997. "Applying 3D Product Modelling Technology to Shipbuilding", *Marine Technology*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 34, number 1, January 1997. Page 56.

[27] S.J. Baum and R. Ramakrishnan, 1997. "Applying 3D Product Modelling Technology to Shipbuilding", *Marine Technology*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 34, number 1, January 1997. Page 62.

[28] S.J. Baum and R. Ramakrishnan, 1997. "Applying 3D Product Modelling Technology to Shipbuilding", *Marine Technology*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 34, number 1, January 1997. Page 56.

[29] D.J. Andrews, 1985. "An Integrated Approach to Ship Synthesis", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 90.

[30] L.J. Rydill, cited in the discussion of:

D.J. Andrews, 1985. "An Integrated Approach to Ship Synthesis", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 92.

[31] Reference to W.G. Holmes, 1930. Plant Location, (McGraw-Hill, New York, New York), as found in

James A. Tompkins, John A. White, 1984. Facilities Planning, (John Wiley & Sons, Inc., New York, New York).

[32] A. Cort and W. Hills, 1987. "Space Layout Design Using Computer Assisted Methods", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), May 1987.

[33] P.B. Mirchandani and R.L. Francis, eds., 1990. Discrete Location Theory (John Wiley & Sons, Inc. New York, New York). Page 27.

[34] R.L. Francis, L.F. McGinnis, Jr. and J.A. White, 1992. Facility Layout and Location: An Analytical Approach, (Prentice-Hall Canada Inc., Toronto, 2nd Edition). Page 165.

[35] H.L. Hales, 1984. Computer-Aided Facilities Planning, (Marcel Dekker, Inc., New York, New York). Page 50.

[36] R.L. Francis and J.A. White, 1974. Facility Layout and Location: An Analytical Approach (Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1st Edition). Page 129-133.

[37] R.L. Francis, L.F. McGinnis, Jr. and J.A. White, 1992. Facility Layout and Location: An Analytical Approach (Prentice-Hall Canada Inc., Toronto, 2nd Edition). Page 149.

[38] D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom).

[39] D.J. Andrews, 1985. "An Integrated Approach to Ship Synthesis", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom).

[40] D.J. Andrews, 1985. "An Integrated Approach to Ship Synthesis", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 73.

[41] J.P. Hope, 1981. "The Process of Naval Ship General Arrangement Design and Analysis", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), August 1981.

[42] A. Cort and W. Hills, 1987. "Space Layout Design Using Computer Assisted Methods", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), May 1987.

[43] M.Th. van Hees, 1995. "Towards Practical Knowledge-based Design Modelling", *Proceedings of the 6th International Symposium on Practical Design of Ships and Mobile Units, Seoul Korea, September 17-22, 1995*, (The Society of Naval Architects of Korea, Seoul, Korea), volume 2.

[44] M. Welsh, I.L. Buxton and W. Hills, 1990. "The Application of an Expert System to Ship Concept Design Investigations", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom).

[45] B. Johnson, N. Glinos, N. Anderson, D. McCallum, W. Beaver and P. Fitzsimmons, 1990. "Database Systems for Hull Form Design", *SNAME Transactions*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 98.

[46] G.J. Klir and T.A. Folger, 1988. Fuzzy Sets, Uncertainty, and Information. (Prentice-Hall, Inc., Englewood Cliffs, New Jersey). Page 265.

[47] E. Turban, 1988. "Review of Expert Systems Technology", *IEEE Transactions on Engineering Management*, (The Institute of Electrical and Electronic Engineers, New York, New York), volume 35, number 2, May 1988.

[48] S. Ashley, 1992. "Engineous Explores the Design Space", *Mechanical Engineering*, (The American Society of Mechanical Engineers, New York, New York), February 1992.

[49] R.L. Francis and J.A. White, 1974. Facility Layout and Location: An Analytical Approach, (Prentice-Hall Inc., Englewood Cliffs, New Jersey).

[50] R.L. Francis, 1974. "Computerized Layout Planning", Facility Layout and Location: An Analytical Approach, (Prentice-Hall Inc., Englewood Cliffs, New Jersey).

[51] H.L. Hales, 1984. Computer-Aided Facilities Planning, (Marcel Dekker, Inc., New York, New York), November 1984.

[52] P.B. Mirchandani and R.L. Francis, 1990. Discrete Location Theory, (John Wiley & Sons, Inc., Toronto, Canada).

[53] R.W. James and P.A. Alcorn, 1991. "Layout of Facilities", A Guide to Facilities Planning, (Prentice-Hall Inc., Englewood Cliffs, New Jersey).

[54] D.P. Sly, E. Grajo and B. Montreuil, 1996. "Layout Design and Analysis Software", *IIE Solutions*, (The Institute of Industrial Engineers, Norcross, Georgia), August 1996.

[55] J.D. Foley and A. van Dam, 1984. Fundamentals of Interactive Computer Graphics, (Addison-Wesley Publishing Company, Reading, Massachusetts).

[56] J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, 1995. Computer Graphics: Principles and Practice, (Addison-Wesley Publishing Company, Reading, Massachusetts). March 1995.

[57] G.J. Klir and T.A. Folger, 1988. Fuzzy Sets, Uncertainty, and Information, (Prentice-Hall, Englewood Cliffs, New Jersey). Pages 108-109.

[58] G.J. Klir and T.A. Folger, 1988. Fuzzy Sets, Uncertainty, and Information, (Prentice-Hall, Englewood Cliffs, New Jersey). Page 11.

[59] The idea of representing this problem using balloons came up briefly in a conversation with M. Fuglem of Memorial University sometime in the summer of 1995. At the time I was describing how Semi-Solids superimposes objects on their surroundings. I have expanded the concept considerably for its inclusion in this document.

[60] G.R. Snaith, cited in the discussion of:

D.J. Andrews, 1985. "An Integrated Approach to Ship Synthesis", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom). Page 92.

[61] S. Ashley, 1992. "Engineous Explores the Design Space", *Mechanical Engineering*, (The American Society of Mechanical Engineers, New York, New York), February 1992.

Selected Bibliography

1988. "General Arrangement Drawing Format", *Technical & Research Bulletin 7-2, Panel SD-4 (General Arrangements) of the Ship Design Committee*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 7, number 2, May 1988.

1988. "General Arrangement Drawing Details", *Technical & Research Bulletin 7-3, Panel SD-4 (General Arrangements) of the Ship Design Committee*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 7, number 3, May 1988.

1990. "General Arrangement Design Criteria and Constraints", *Technical & Research Bulletin 7-4, Panel SD-4 (General Arrangements) of the Ship Design Committee*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 7, number 4, May 1990.

1994. Microsoft Access ver. 2.0: Building Applications, (Microsoft Corporation), July 1994.

1994. Microsoft Access ver. 2.0: User's Guide, (Microsoft Corporation), July 1994.

A. Akinturk, M. Atlar and S.M. Calisal, 1995. "Preliminary Design of Multi-hull Fishing Vessels using an Expert System Environment", *Proceedings of the 6th International Symposium on Practical Design of Ships and Mobile Units, Seoul Korea, September 17-22, 1995*, (The Society of Naval Architects of Korea, Seoul, Korea), volume 2.

S. Ammar, 1989. "Determining the 'Best' Decision in the Presence of Imprecise Information", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 29.

D.J. Andrews, 1981. "Creative Ship Design", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom).

D.J. Andrews, 1985. "An Integrated Approach to Ship Synthesis", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom).

S. Ashley, 1992. "Engineous Explores the Design Space", *Mechanical Engineering*, (The American Society of Mechanical Engineers, New York, New York), February 1992.

S.J. Baum and R. Ramakrishnan, 1997. "Applying 3D Product Modeling Technology to Shipbuilding", *Marine Technology*, (The Society of Naval Architects and Marine Engineers, The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 34, number 1, January 1997.

M.P. Biswal, 1992. "Fuzzy Programming Techniques to Solve Multi-Objective Geometric Programming Problems", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 51.

K.S. Brower and K.W. Walker, 1986. "Ship Design Computer Programs - An Interpolative Technique", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), May 1986.

D.K. Brown, 1993. "Naval Architecture", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), January 1993.

R. Byrnes and H.S. Marcus, 1990. "A Systematic Approach to Producibility and Lessons Learned for Naval Shipbuilding", *Journal of Ship Production*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 6, number 4, November 1990.

D.E. Calkins, 1983. "An Interactive Computer-Aided Design Synthesis Program for Recreational Powerboats", *SNAME Transactions*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 91.

D.E. Calkins, V.E. Theodoracatos, G.D. Aguilar and D.M. Bryant, 1989. "Small Craft Hull Form Surface Definition in a High-Level Computer Graphics Design Environment", *SNAME Transactions*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 97.

C.M. Carlson and H. Fireman, 1987. "General Arrangement Design Computer System and Methodology", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), May 1987.

D. Catley and T. Koch, 1995. "The Impact of New Technologies on Computer-Aided Ship Design", *Proceedings of the 6th International Symposium on Practical Design of Ships and Mobile Units, Seoul Korea, September 17-22, 1995*, (The Society of Naval Architects of Korea, Seoul, Korea), volume 2.

S. Chanas, 1989. "Fuzzy Programming in Multiobjective Linear Programming — A Parametric Approach", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 29.

S.C. Chapra and R.P. Canale, 1988. Numerical Methods for Engineers, 2nd Edition, (McGraw-Hill Publishing Company, New York, New York).

Y. Chou and C.O. Benjamin, 1992. "An AI-based Decision Support System for Naval Ship Design", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), May 1992.

L.M. Collier, 1983. "Use of the Computer In Facilities Planning — Yes", *Industrial Engineering*, (The Institute of Industrial Engineers, Norcross, Georgia), March 1983.

A. Cort and W. Hills, 1987. "Space Layout Design Using Computer Assisted Methods", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), May 1987.

A.M. D'Arcangelo, 1969. "Relationship Between Spaces and Access", Ship Design and Construction, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey).

G. D'Souza and B.B. Mohanty, 1986. "An Interactive Multilevel, Multicriteria Dynamic Approach to Facility Layout Analysis", *Fall Industrial Engineering Conference*, (The Institute of Industrial Engineering, Norcross, Georgia).

D. Dutta, R.N. Tiwari and J.R. Rao, 1992. "Multiobjective Linear Fractional Programming — A Fuzzy Set Theoretic Approach", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 52.

D.J. Eyres, 1988. Ship Construction, (Heinemann Professional Publishing).

R.D. Filley, 1985. "Three Emerging Computer Technologies Boost Value of, Respect For Facilities Function", *Industrial Engineering*, (The Institute of Industrial Engineers, Norcross, Georgia), May 1985.

J.D. Foley and A. van Dam, 1984. Fundamentals of Interactive Computer Graphics, (Addison-Wesley Publishing Company, Reading, Massachusetts).

J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, 1995. Computer Graphics: Principles and Practice, (Addison-Wesley Publishing Company, Reading, Massachusetts), March 1995.

R.B. Footlik, 1983. "Use of the Computer In Facilities Planning — No", *Industrial Engineering*, (The Institute of Industrial Engineers, Norcross, Georgia), April 1983.

R.L. Francis and J.A. White, 1974. Facility Layout and Location: An Analytical Approach, (Prentice-Hall Inc., Englewood Cliffs, New Jersey).

R.L. Francis, 1974. "Computerized Layout Planning", Facility Layout and Location: An Analytical Approach, (Prentice-Hall Inc., Englewood Cliffs, New Jersey).

G.K. Gaston, 1984. "Facility Layout Optimizes Space, Minimizes Costs", *Industrial Engineering*, (The Institute of Industrial Engineers, Norcross, Georgia), May 1984.

H.L. Hales, 1984. Computer-Aided Facilities Planning, (Marcel Dekker, Inc., New York, New York), November 1984.

H.L. Hales, 1984. "Computerized Facilities Planning and Design: Sorting Out The Options Available Now", *Industrial Engineering*, (The Institute of Industrial Engineers, Norcross, Georgia), May 1984.

R.L. Harrington, 1992. Marine Engineering, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey).

P.J. Hartman, 1988. "Practical Applications of Artificial Intelligence in Naval Engineering", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), November 1988.

J.P. Hope, 1981. "The Process of Naval Ship General Arrangement Design and Analysis", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), August 1981.

Y.A. Hosni and G.E. Whitehouse, "Layout Evaluation", (The Institute of Industrial Engineers, Norcross, Georgia).

Y.A. Hosni, G.E. Whitehouse and T.S. Atkins, "Optimum Facility Location", (The Institute of Industrial Engineers, Norcross, Georgia).

Y.A. Hosni, G.E. Whitehouse and T.S. Atkins, "Micro-CRAFT", (The Institute of Industrial Engineers, Norcross, Georgia).

Y.A. Hosni, G.E. Whitehouse and T.S. Atkins, "From/To Chart Generator", (The Institute of Industrial Engineers, Norcross, Georgia).

Y.A. Hosni and T.S. Atkins, 1982. "Facilities Planning Using Microcomputers", *Annual Industrial Engineering Conference*.

K.M. Hyde and D.J. Andrews, 1992. "CONDES — A Preliminary Warship Design Tool to Aid Customer Decision Making", *Proceedings of the 5th International Symposium on Practical Design of Ships and Mobile Units, Newcastle UK, May 17-22, 1992*, (Elsevier Science Publishers B.V. (North Holland), London, United Kingdom), volume 2.

F.R. Jacobs, J.W. Bradford and L.P. Ritzman, 1980. "Computerized Layout: An Integrated Approach to Special Planning and Communications Requirements", *Industrial Engineering*, (The Institute of Industrial Engineers, Norcross, Georgia), July 1980.

R.W. James and P.A. Alcorn, 1991. "Layout of Facilities", A Guide to Facilities Planning, (Prentice-Hall Inc., Englewood Cliffs, New Jersey).

B. Johnson, N. Glinos, N. Anderson, D. McCallum, W. Beaver and P. Fitzsimmons, 1990. "Database Systems for Hull Form Design", *SNAME Transactions*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 98.

K. Khaopravetch and R. Nanda, 1990. "Assessing Solution Efficiency for Quadratic Assignment Problems", *Industrial Engineering*, (The Institute of Industrial Engineers, Norcross, Georgia), April 1990.

S. Khator and C. Moodie, 1983. "A Microcomputer Program to Assist in Plant Layout", *Industrial Engineering*, (The Institute of Industrial Engineers, Norcross, Georgia), March 1983.

E.T. Kinney and D.F. Funkhouser, 1987. "A Disciplined Approach to Machinery Arrangements in Ship Design", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), May 1987.

G.J. Klir and T.A. Folger, 1988. Fuzzy Sets, Uncertainty, and Information, (Prentice-Hall, Englewood Cliffs, New Jersey).

Y. Lai and C. Hwang, 1992. "A New Approach to Some Possibilistic Linear Programming Problems", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 49.

Y. Lai and C. Hwang, 1992. "Interactive Fuzzy Linear Programming", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 45.

J.F. Leahy III and J.C. Ryan, 1987. "CAD/CAM Directions for the Navy", *Journal of Ship Production*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 3, number 1, February 1987.

K.Y. Lee, S.W. Suh and S. Han, 1992. "On the Development of Computer Integrated Basic Ship Design and Performance Analysis System", *Proceedings of the 5th International Symposium on Practical Design of Ships and Mobile Units, Newcastle UK, May 17-22, 1992*, (Elsevier Science Publishers B.V. (North Holland), London, United Kingdom), volume 2.

J. Lee, K. Lee, N. Park, J. Kim, Y. Jang, J. Bae and H. Shim, 1995. "Knowledge-based Design System for the Machinery Arrangement of Ship Engine Room", *Proceedings of the 6th International Symposium on Practical Design of Ships and Mobile Units, Seoul Korea, September 17-22, 1995*, (The Society of Naval Architects of Korea, Seoul, Korea), volume 2.

K.Y. Lee, S.W. Suh, D. Shin, D.K. Lee, W. Kang and Y. Kim, 1995. "Development of a Computerized Ship Design System", *The 6th International Symposium on Practical Design of Ships and Mobile Units, Seoul Korea, September 17-22, 1995*, (The Society of Naval Architects of Korea, Seoul, Korea), volume 2.

G.F. Luger and W.A. Stubblefield, 1989. Artificial Intelligence and the Design of Expert Systems, (Benjamin/Cummings Publishing Company Inc., Redwood City, California).

M.K. Luhandjula, 1989. "Fuzzy Optimization: An Appraisal", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 30.

T.D. Lyon and F. Mistree, 1985. "A Computer-Based Method for the Preliminary Design of Ships", *Journal of Ship Research*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 29, number 4, December 1985.

M.J. McCormick, 1985. "A Step Beyond Computer-Aided Layout", *Industrial Engineering*, (The Institute of Industrial Engineers, Norcross, Georgia), May 1985.

J.C. McNeal, H.G. Nilsen and J.J. Matthews, 1985. "CAD/CAM Applications to Mass Properties", *Journal of Ship Production*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 1, number 2, May 1985.

P.B. Mirchandani and R.L. Francis, 1990. Discrete Location Theory, (John Wiley & Sons, Inc., Toronto, Canada).

F. Mistree, W.F. Smith, B.A. Bras, J.K. Allen and D. Muster, 1990. "Decision-Based Design: A Contemporary Paradigm for Ship Design", *SNAME Transactions*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 98.

J.M. Moore, 1980. "Computer Methods In Facilities Layout", *Industrial Engineering*, (The Institute of Industrial Engineers, Norcross, Georgia), September 1980.

C.A. Mota Soares, 1986. "Computer Aided Optimal Design: Structural and Mechanical Systems", *Proceedings of the NATO Advanced Study Institute on Computer Aided Optimal Design: Structural and Mechanical Systems held in Troia Portugal, June 29 - July 11, 1986*, (Springer-Verlag, New York, New York), April 1986.

R. Nelson, 1993. Running Visual Basic, (Microsoft Press, Redmond, Washington), November 1993.

K. Niwa, 1990. "Toward Successful Implementation of Knowledge-Based Systems: Expert Systems vs. Knowledge Sharing Systems", *IEEE Transactions on Engineering Management*, (The Institute of Electrical and Electronic Engineers, New York, New York), volume 37, number 4, November 1990.

C. Olsen, 1993. "Helper & Hinderance — Optimization and Fuzzy Sets", *Engineering Economic Analysis Course Term Paper*, July 1993.

M.A. Polini, D.J. Wooley and J.D. Butler, 1997. "Impact of Simulation-Based Design on Today's Shipbuilders", *Marine Technology*, (The Society of Naval Architects and Marine Engineers, The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 34, number 1, January 1997.

A.A.G. Requicha, 1980. "Representations for Rigid Solids: Theory, Methods, and Systems", *ACM Computing Surveys*, (Association of Computing Machinery), volume 12, number 4.

M.E. Resner, S.H. Klomparens and J.P. Lynch, 1981. "Machinery Arrangement Design — A Perspective", *Naval Engineers Journal*, (The American Society of Naval Engineers, Alexandria, Virginia), June 1981.

H. Rommelfanger, R. Hanuscheck and J. Wolf, 1989. "Linear Programming with Fuzzy Objectives", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 29.

M. Sakawa and H. Yano, 1989. "An Interactive Fuzzy Sacrificing Method for Multiobjective Nonlinear Programming Problems with Fuzzy Parameters", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 30.

M. Sakawa and H. Yano, 1989. "Interactive Decision Making for Multiobjective Nonlinear Programming Problems with Fuzzy Parameters", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 29.

V. Sankar, 1986. "INLAPS: An Integrated Layout Planning System — Some Algorithms for the FLP using Quadratic Programming and Statistical Analysis", (M.Eng. Thesis, Faculty of Engineering and Applied Science, Memorial University of Newfoundland, St. John's, Newfoundland), July 1986.

D.A. Savic and W. Pedrycz, 1991. "Evaluation of Fuzzy Linear Regression Models", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 39.

R. Sedgewick, 1983. Algorithms, (Addison-Wesley Publishing Company, Reading, Massachusetts).

P. Sen and J. Yang, 1995. "An Investigation into the Influence of Preference Modelling in Ship Design with Multiple Objectives", *Proceedings of the 6th International Symposium on Practical Design of Ships and Mobile Units, Seoul Korea, September 17-22, 1995*, (The Society of Naval Architects of Korea, Seoul, Korea), volume 2.

D.P. Sly, 1995. "Computerized Facilities Design and Management", *IIE Solutions*, (The Institute of Industrial Engineers, Norcross, Georgia), August 1995.

D.P. Sly, E. Grajo and B. Montreuil, 1996. "Layout Design and Analysis Software", *IIE Solutions*, (The Institute of Industrial Engineers, Norcross, Georgia), August 1996.

D.P. Sly, 1996. "Using CAD for Space Planning and Asset Management", *IIE Solutions*, (The Institute of Industrial Engineers, Norcross, Georgia), November 1996.

D.P. Sly, 1996. "Issues and Techniques for Using CAD to Draw Factory Layouts", *IIE Solutions*, (The Institute of Industrial Engineers, Norcross, Georgia), August 1996.

J. MacGregor Smith and R.S. Pelosi, 1982. "Interactive Modeling of Wicked Design Problems", *Annual Industrial Engineering Conference*.

W.F. Smith, S. Kamal and F. Mistree, 1987. "The Influence of Hierarchical Decisions on Ship Design", *Marine Technology*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 24, number 2, April 1987.

J.A. Tompkins, 1984. "Successful Facilities Planner Must Fulfill Role of Integrator In the Automated Environment", *Industrial Engineering*, (The Institute of Industrial Engineers, Norcross, Georgia), May 1984.

E. Turban, 1988. "Review of Expert Systems Technology", *IEEE Transactions on Engineering Management*, (The Institute of Electrical and Electronic Engineers, New York, New York), volume 35, number 2, May 1988.

I.B. Turksen, D. Ulguray and Q. Wang, 1992. "Hierarchical Scheduling Based on Approximate Reasoning — A Comparison with ISIS", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 46.

M. Th. van Hees, 1992. "QUAESTOR: A Knowledge-Based System for Computations in Preliminary Ship Design", *Proceedings of the 5th International Symposium on Practical Design of Ships and Mobile Units, Newcastle UK, May 17-22, 1992*, (Elsevier Science Publishers B.V. (North Holland), London, United Kingdom), volume 2.

M. Th. van Hees, 1995. "Towards Practical Knowledge-based Design Modelling", *Proceedings of the 6th International Symposium on Practical Design of Ships and Mobile Units, Seoul Korea, September 17-22, 1995*, (The Society of Naval Architects of Korea, Seoul, Korea), volume 2.

R.K. Verma, 1990. "Fuzzy Geometric Programming with Several Objective Functions", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 35.

J.R. Walters and N.R. Nielsen, 1988. Crafting Knowledge-Based Systems: Expert Systems Made Easy / Realistic, (John Wiley & Sons, Inc., New York, New York), July 1988.

D.G.M. Watson and A.W. Gilfillan, 1976. "Some Ship Design Methods", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom).

M. Welsh, I.L. Buxton and W. Hills, 1990. "The Application of an Expert System to Ship Concept Design Investigations", *RINA Transactions*, (The Royal Institution of Naval Architects, London, United Kingdom).

J. Wollert, M. Lehne and B.E. Hirsch, 1992. "Modeling for Ship Design and Production", *Journal of Ship Production*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 8, number 1, February 1992.

D.F. Wong, H.W. Leong and C.L. Liu, 1988. Simulated Annealing for VLSI Design, (Kluwer Academic Publishers, Boston, Massachusetts), May 1988.

K.L. Wood, K.N. Otto and E.K. Antonsson, 1992. "Engineering Design Calculations with Fuzzy Parameters", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 52.

D.J. Wooley and M.L. Manix, 1987. "Development of an Initial Graphics Exchange Specification Capability", *Journal of Ship Production*, (The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey), volume 3, number 4, November 1987.

T. Yang, J.P. Ignizio and H. Kim, 1991. "Fuzzy Programming with Non-Linear Membership Functions: Piecewise Linear Approximation", *Fuzzy Sets and Systems*, (Elsevier Science Publishers B.V. (North Holland), New York, New York), volume 41.

A1

Appendix 1: CAD, Solid Modelling and Semi-Solids

This appendix reviews the representation formats currently used in solid modelling to provide a basis for comparison for the new formulation. Semi-Solids is similar to *Surface & Boundary Representations*, but differs in its manipulation. The often conflicted processes of depiction and modelling are discussed and are used to introduce the unique methods used in manipulation of objects constructed as *Semi-Solids*.

A1.1 Raster Representations

The fundamental building block of computer technology has been binary codes. Monitors and printers control depictions by means of pixels, or tiny dots, covering their entire surfaces. Raster images are formed by combinations of these dots. Used very commonly for graphical interfaces and photographic reproductions, there are a number of significant drawbacks associated with *raster representations*:

- the storage required for even a simple *raster image* is large because the status of every pixel must be saved
- image resolution is limited to that in which it was created such that 'zooming' closer to the image does not give a more detailed view
- editing images requires slow manual manipulation on a dot-by-dot basis
- ill suited for 3D models because of the enormous increase in the number of dots required
- ill suited for problems requiring some sort of mathematical calculation since the representation is not based on values
- reduced resolution in diagonal and other non-rectilinear shapes

A1.2 Vector Representations

Vector representations are software dependant as opposed to image dependent as is the case in *raster representations*. That is, vector images require a software interpretation of their data in order to generate an image. Recall that in the raster format, images are stored by denoting a pixel location and attributing that pixel with a colour or shade value. In the vector representation of a line, all that would be stored would be the Euclidian coordinates of the line's two end points, and a note indicating that this object is a line. Any display of the line requires computer software to generate the appropriate screen pixels. Other attributes can be associated with the line but the format is the same — one or more location coordinates, plus appropriate attributes and identifiers so that the software can distinguish between different objects. As a result, models are often simple compared to the computational effort of their display.

Vector representations can be divided into three broad modelling subgroups: lines, surfaces, and solids. While the more complex model forms appear to merely employ their relatives as primitives, each representation has its own peculiarities and applications. Three dimensional models are easily and efficiently constructed using vector representations because all that is required is the establishment of their Euclidean coordinates. Software manipulations which control viewpoints and limit display areas are then used for the image's depiction.

Software used for vector model construction is almost invariably interactive in format. While perhaps the most versatile for single depictions, for scenario evaluation or animation interfaces must be far more batch oriented with their associated drawbacks. Unfortunately, while their data is similar, batch CAE software does not lend itself to the simple drawing manipulation and reproduction that is found with the interactive CAD packages.

A1.3 Lines

By far the most common of the Vector Representations, simple linear objects are the mainstay of a great number of CAD and Desk Top Publishing software products. Other 2D objects such as circles and arcs also fall into this category. Using these primitives, it is possible to construct complex objects much as one would using a pencil and paper. However, the representation is poor when it comes to colouring or giving a 'surface' to objects created from lines. This is because the creation of surfaces requires the identification of a 'region' and then a means of filling that region. Hatching is the most common manner in which linear objects are given surfaces, and employs a continuous boundary for the filled region with a simple fill pattern constructed from additional lines.

A1.4 Surfaces

Model developers interested in filled or rendered images found that not only was hatching inadequate but the line representation was difficult to manipulate into surfaces. Instead they added additional subroutines to the software such that by creating a grid of coordinates or vertices, the software would not only connect adjacent points with lines, but would also fill the regions between the points with a surface. Thus, just as line representation requires software to create objects for a series of coordinates, surfaces require the software not only to 'draw' the lines between the coordinates, but also to apply colour or shading to the circumscribed regions.

The representation of surfaces need not be only in two dimensions. The smooth rendering found in the depictions of many modern software packages is a reflection of this. That is, not only are meshes created in 3D, but the lines and surfaces which connect the mesh vertices can be of higher mathematical orders. Hence a mesh whose coordinates might suggest a great number of flat facets can actually be drawn by the software as a smooth and continuous bicubic surface. The ease with which surfaces can be applied to complex shapes is directly related to the size and shape of the facets of their meshes. Three- and four-sided mesh elements are the most common.

Surface modelling by means of meshes was an important advance in computer aided drafting since mesh representations could be used for more than just linking many lines together in the form of a single entity. One of the first applications of meshes was as input for rendering software in which a meshed surface is displayed as a solid, whole surface. The process of surface rendering has been the focus of many texts dealing with computer graphics[1] and is a

---

1 J.D. Foley and A. van Dam, 1984. Fundamentals of Interactive Computer Graphics, (Addison-Wesley Publishing Company, Reading, Massachusetts).

sophisticated problem of light, colour, texture, surface development, projection, and computation.

A second important application of surface meshes has been in the area of numerical modelling. Software for the study of hydrodynamics and for finite elements generally use meshed surfaces as part of their inputs. The reason meshes have been popular for modelling is that they give the software an adjacency relationship between individual objects or elements. For example, **Figure 88** shows a simple four-sided mesh element.[2] While the element may actually be a part of a curved surface, without additional control points, its curvature cannot be calculated, represented, or utilized.

Surface representations are difficult to create and edit since they require a large array of vertex coordinates for their creation. Further, rendered surfaces are extremely demanding computationally such that full rendering is rarely used for anything other than final output. The depiction of surfaces is often different from the modelling of surfaces because it is relatively difficult for software to intersect surfaces in areas other than on their underlying mesh structure. This problem in particular leads to solid modelling.

---

2 J.D. Foley and A. van Dam, 1984. Fundamentals of Interactive Computer Graphics, (Addison-Wesley Publishing Company, Reading, Massachusetts). Page 529.

A1.5 Solids

Solid modelling "is the representation of volumes completely surrounded by surfaces, such as a cube, an airplane, or a building."[3] Although many designers have switched to Solid modelling, the transition has been very slow in part because the computational horsepower has only recently become available. As a result there tends to be a lack of familiarity with the new representation and its methods of manipulation. Also, there are a number of shapes which cannot be easily constructed using the most common solid representations. An obvious example would be the compound bicubic spline curvature of a hull surface.

Generally solid models are manipulated by means of transactions referred to as *Boolean Set Operations* (**Figure 89**[4]) — a set of convenient tools for users since they remove the tedious task of editing the locations of various surface vertices or volume primitives. However, this does not mean that there is independence from such determinations; instead, the Boolean Set Operations are coded into the underlying software and as a result, what appears to be a simple transaction from the point of view of the user may be quite complex for the software. While not unacceptable when used interactively, the time required for the computation of solid models may yet prove to be a stumbling block for the automation of models.

Where solids come into their own is in their ability to evaluate interference. Solid modelling has grown in popularity for this reason and has proven itself very useful in areas such as the routing of piping and HVAC services. Interference checks are generally performed by means of

---

3 J.D. Foley and A. van Dam, 1984. Fundamentals of Interactive Computer Graphics, (Addison-Wesley Publishing Company, Reading, Massachusetts). Page 529.

4 Adapted from:

J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, 1996. Computer Graphics: Principles and Practice, 2nd Edition, (Addison-Wesley Publishing Company, Reading, Massachusetts). Page 535.

Boolean transactions such that where a subtraction is applied, and a volume change for two objects is registered, an interference exists. However, just as for all vector-based representations, solid modelling offers no user-accessible means for evaluating relative positions of objects (**Figure 90**). This is the strength of interactive approaches since the onus is on the user to make the interpretation of the relationships between objects.

An area which the solid modeller has also proven itself is in cases in which additional information is to be attributed to a particular object. For example, software is available which will allow the user to attribute a mass or density to particular objects and thereby perform weight calculations for the computer model.

Solid model representations can be divided into six distinct groups: *Primitive Instancing, Sweeps, Surface & Boundary Representations, Spatial Partitioning, Spatial-Occupancy Enumeration*, and *Constructive Solid Geometry*. In several cases, additional subheading have been used to discuss particular subsets of these six groups. While there are many representations, these are both the most common and the most distinctive. Semi-Solids falls under the class of *Surface & Boundary Representations*, and for this reason greater emphasis has been placed on this section. Section A1.12 shows a detailed comparison of these the solid models discussed, including the Semi-Solids formulation.

A1.6 Primitive Instancing

Primitive instancing is a solids representation which is often used for the representation of relatively complex objects such as gears or bolts. The objects tend to be those which commonly appear in a model but whose construction from primitive shapes through Boolean transactions might be either tedious or impossible. Analogous to the CAD construct *group* or *block*, the objects lack the facility for alteration or combination with other objects. Objects defined by primitive instancing are generally defined by means of programmed code rather than by any direct definition of vertices and surfaces. For example, **Figure 91**[5] shows a gear created through the specification of numerical constraints particular to their shapes.

A1.7 Sweep Representations

A simple means of defining a 3D entity is by means of a sweep. Such objects are created by defining a closed 2D shape and then either rotating it about an axis or translating it linearly or along a curve. In translation, a sweep resembles an extrusion as one might find in plastics or metal fabrication. For the rotational case, swept objects have an appearance similar to that of a material which has been turned on a lathe. **Figure 92** shows a 2D shape used as a template for 3D solids through this approach. Because of the potential complexity in their definition, sweeps tend to be difficult to combine with other objects without reverting either manually or algorithmically to a more malleable representation such as surfaces or lines.

A1.8 Surface and Boundary Representations

Surfaces and Boundaries are both the most robust and the most complex of the representations described here. They are robust in the sense that their capacity to represent objects is virtually unlimited, but complex in that the format requires accurate and consistent

---

5 Adapted from:

J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, 1996. Computer Graphics: Principles and Practice, 2nd Edition, (Addison-Wesley Publishing Company, Reading, Massachusetts). Page 539.

management of a number of lists of data. The problem of data representation will become more apparent in the following descriptions.

A1.8.1 Explicit Polygons

Intuitively, the most obvious way to represent a flat surface is to define an *n*-sided patch using *n* coplanar lines. However, where many facets are to be defined, the manipulation of lines becomes cumbersome. This has in turn encouraged the use of standard 3- or 4-sided 3D Face primitive (**Equation 14**) in which sets of (x, y, z) coordinates refer to the corners of the patch.

$$3D\ Face\big((x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3), \ldots, (x_n, y_n, z_n)\big)$$

Equation 14 Typical format of a 3D Face graphical object. The face element is derived from the corners by which it is defined. 3D Faces are commonly four-sided. Where only a three-sided figure is desired, it is common for the coordinate of the fourth corner in the structure to be set equal to the values of the third corner.

In terms of manipulation, this formulation is efficient for small numbers of faces. However, when used for the creation of a mesh, the duplication of shared coordinates becomes costly from the point of view of storage. Further, in terms of display or output, that shared edges are not explicitly defined by the representation leads to the computationally wasteful duplication of the lines which join them.[6] That there is no reference as to which patches are adjacent is a related problem and one which entails a long and computationally expensive search to surmount. Because of these problems, shared lines are displayed twice, and changes to vertices are slow because each vertex must be sought several times through the sifting of the entire list of patches.

---

6 J.D. Foley and A. van Dam, 1984. Fundamentals of Interactive Computer Graphics, (Addison-Wesley Publishing Company, Reading, Massachusetts). Page 508.

A1.8.2 Polygon Meshes

To address the efficiency problem raised in the previous section two tactics can be pursued. First, where polygons are assembled to form a mesh, the coordinates of the corners of the polygons are stored in a long list. Each vertex coordinate appears only once, and a polygon is defined by means of pointers to this list. For example, a polygon might be defined as P = (3, 4, 2, 7) where each of 3, 4, 2, and 7 refer to a particular coordinate in the vertex list.

As shown in **Figure 93**,[7] this representation elegantly solves the problem of repeated vertex points. It also addresses the editing problem noted in the previous section because a change in the position of a vertex is immediately reflected in all the patches whose pointers address that vertex.

It is important to distinguish between the needs of modelling software relative to those of display. For the purpose of modelling, defining patches by pointers to a list of vertices is sufficient to reduce storage requirements and facilitate editing changes. However, display problems tend to be more common and hence an additional change in the data representation is required. Each time software displays an object on the screen, it must first represent its model of that object as a 2D view. As a result, the screen locations of every vertex, line, and surface must be determined. Since the majority of mesh elements share edges with other elements, computational time can be reduced by almost one-half just by preventing the computer from

---

7 Adapted from:

J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, 1996. Computer Graphics: Principles and Practice, 2nd Edition, (Addison-Wesley Publishing Company, Reading, Massachusetts). Page 474.

creating all four lines for each mesh element[8]. Thus, for the purpose of display, many polygon meshes are described by a list of the edges which form each mesh element. **Figure 94**[9] shows a polygon defined as pointers to a list of edges. In turn, the edges in the list point to vertices in the vertex list to complete the definition of the patch. Hence, display architecture requires only that all the edges be displayed, and since the edges only appear once in the edge list, no longer is there the case of two lines being drawn on top of one another during the display of the mesh.

Since the Semi-Solids representation draws on the structures described in this section two points should be emphasized: the first is the use of pointers to refer to data elements shared by several objects which is similar to the mechanics of relational databases; second, it is often convenient to establish a representation format which facilitates manipulation, just as for display purposes it is computationally efficient to represent the mesh as edges.

A1.8.3 Quadric Surfaces

Instead of being defined by points and vectors, Quadric Surfaces are defined by mathematical functions of the model's coordinate system. Generally of the form shown in **Equation 15**, such expressions are functions of the coordinate system of the model space.

---

8 Given a rectangular $m \times n$ mesh, the following lines are required to be drawn:

$(n-1) \times (m-1)$	elements of the form	┌
$1 \times (m-1)$	elements of the form	⊏
$(n-1) \times 1$	elements of the form	∩
1×1	elements of the form	□

Summing this list suggests that the number of edges required is $2nm + m + n$ Since the complete delineation of every facet requires the drawing of $4nm$ lines, the savings potential reduces to $\frac{1}{2} - (m + n) / 4nm$ which for a very large mesh approaches the value of ½ or 50%.

9 Adapted from:

J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, 1996. Computer Graphics: Principles and Practice, 2nd Edition, (Addison-Wesley Publishing Company, Reading, Massachusetts). Page 475.

$$f(x,y,z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0$$

Equation 15 Generic function defining a quadric surface.

From the point of view of representation, curved surfaces expressed in this form are extremely accurate and are limited only by the number of surface points determined for display purposes. In terms of modelling, such implicit definitions of 3D objects are useful since they provide an exact mathematical solution for every location on the surface without requiring complex interpolation between points or across mesh surfaces.

Quadric surfaces are also efficient for data storage since such a function could represent a complex surface or object boundary. However, definitions of this form do not lend themselves to Boolean Set operations because of the prohibitive increase in formula complexity. For the construction of non-uniform figures, a number of patches are usually required and are analogous to hard chines in a hull form. Where two surfaces intersect at a chine it is possible that there will be poor agreement regarding the shape and position of the shared edge. For this reason Quadric Surfaces are generally only used to define uniform objects such as spheres and toroids.

A1.8.4 Bicubic Surfaces

Similar to Quadric Surfaces, a Bicubic Surface definition seeks to mathematically represent a surface through the use of a surface function. However, instead of representing an entire surface expanse, Bicubic Surfaces are applied on a patch-by-patch basis. To this end, the surface of each square patch is treated as discrete cubic functions of *s* and *t* as shown in **Equation 16**.

$$x(s,t) = S \cdot M \cdot G_x \cdot M^T \cdot T^T$$
$$y(s,t) = S \cdot M \cdot G_y \cdot M^T \cdot T^T$$
$$z(s,t) = S \cdot M \cdot G_z \cdot M^T \cdot T^T$$

Equation 16 where $S = [\, s^3 \;\; s^2 \;\; s^1 \;\; s \,]$ and s lies in the range $0 \le s \le 1$, $T = [t^3 \;\; t^2 \;\; t^1 \;\; t\,]$ and t lies in the range $0 \le t \le 1$, M is a 4 x 4 matrix of coefficients appropriate to the type of curve being represented, and G is another 4 x 4 matrix of coefficients specific to this particular surface form.

Generally *Bicubic Surfaces* are formed by providing a mesh of vertex points and then indicating that the surface which links those points is one of a number of cubic forms such as Hermite, Bézier, Uniform B-spline, Uniformly Shaped B-spline, Nonuniform B-spline, Catmull-Rom, and Kockanek-Bartels. The regions between each set of four vertices are then discretized by the variables *s* and *t* and form a surface patch. The use of the variables *s* and *t* in this representation make it possible for the surface to be created independent of its location in the design space. Therefore the surface function is unique for each patch but is controlled by its neighbouring vertices through the mathematics which define the function. A detailed derivation and explanation of the creation and manipulation of *Bicubic Surfaces* may be found in the texts of Foley & Van Dam.[10,11] Ship hulls are examples of complex surfaces whose spline derivations make their representation only possible through the use of *Bicubic Surface* definitions.

---

10 J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, 1996. Computer Graphics: Principles and Practice, 2nd Edition, (Addison-Wesley Publishing Company, Reading, Massachusetts).

11 J.D. Foley and A. Van Dam, 1984. Fundamentals of Interactive Computer Graphics, (Addison-Wesley Publishing Company, Reading, Massachusetts).

A1.9 Spatial Partitioning

This category of solid model is the three-dimensional equivalent of the block layout formulation. Here, unique and non-intersecting primitive objects are assembled to fill regions of the design space, thereby defining more complex objects. The important characteristic of all Spatial Partitioning models is that the primitives themselves cannot be combined directly; that is, they are unique and discrete objects which cannot be united or divided to form new objects. However, their grouping can be used to represent more complex objects, in which the primitives continue to appear as distinct entities.

A1.9.1 Spatial-Occupancy Enumeration

Here space is divided into discrete objects, generally cubes, and objects are represented by the locations of filled cubes. A picture of a thousand cubes is worth a thousand words of description; hence **Figure 95**.[12] This is the 3D equivalent of the Block Layout representation commonly used in Facility Layout Algorithms.

A1.9.2 Octrees

This format addresses the obviously high storage requirements of the *Spatial Occupancy Enumeration* formulation. Like that formulation, *Octrees* employ simple geometric forms with which to 'fill' space. The name refers to the cube format since each cube can be divided into cube-shaped octants. In application, *Octrees* seek to reduce the design space into cubes which are

---

12 A.H.J. Christensen, SIGGRAPH '80 Conference Proceedings, *Computer Graphics* (14)3, July 1980. Referenced in:

J.D. Foley and A. van Dam, 1984. Fundamentals of Interactive Computer Graphics, (Addison-Wesley Publishing Company, Reading, Massachusetts).

either wholly contained within or wholly excluded from the object being represented. Where a large cube is only partially 'filled' by the object, it is divided into its octants and each of the smaller cubes evaluated in the same manner. Thus complex objects can be defined using this method through increasing levels of subdivision until an acceptable cut-off resolution has been achieved (see **Figure 96**[13]).

A1.9.3 Binary Space Partitioning Trees

A simplification which further improves the problem of storage requirements from the *Octree* formulation is that employing Binary Space Partitioning Trees. In this method each large cube-shaped primitive which is only partially filled by the object being represented is divided into two sub-spaces, separated by a plane of arbitrary orientation and position. A more detailed explanation of *Binary Space Partitioning Trees* may be found in Foley et al.[14]

A1.10 Constructive Solid Geometry

The most recognized solids representation, Constructive Solid Geometry (CSG) employs simple geometric forms and Boolean Set operators to create complex objects. Common primitive shapes include blocks, cylinders, spheres, and toroids. This use of primitives is different from that of *Spatial-Partitioning Representations* in that when a Boolean transaction alters a primitive, the primitive remains intact but the model is altered mathematically to reflect the

13 Adapted from:

J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, 1996. Computer Graphics: Principles and Practice, 2nd Edition, (Addison-Wesley Publishing Company, Reading, Massachusetts). Page 550.

14 J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, 1996. Computer Graphics: Principles and Practice, 2nd Edition, (Addison-Wesley Publishing Company, Reading, Massachusetts).

transaction. Therefore shapes can be altered by changing the size and position of their underlying primitives. In contrast, *Spatial-Partitioning Representations* retain only their current shape.

The primitives do not lend themselves to representing objects whose surfaces are of a higher order because higher order objects almost invariably require unique multi-faceted surfaces. Thus, using simple boxes, cones and cylinders, it is nearly impossible to reproduce complex 3D surfaces as might be found comprising the hull of a ship. The creation of complex objects by Constructive Solid Geometry is a sophisticated problem and does not lend itself to automation. For example, recreating an existing object through combination of primitive shapes is a process which is very difficult to automate because different combinations of primitives may be used to form the object, and the decision as to how to apply the Boolean Set combinations is non-trivial. This makes *CSG* a good example of a CAD system which is effective in representation but requires a human to interpret and coordinate the model's primitives.

Advances in Chess software have made possible accurate prediction of outcomes on the basis of few inputs. The combination of brute force computation, optimization and knowledge-based systems employed in the most recent iteration of IBM's Deep Blue offers the potential to automate primitive manipulation. However, such algorithms when applied to CAD are far from development or commercial application.

A1.11 Semi-Solids

Although applicable to civilian applications, this project was originally intended to facilitate the design of ships. As such a key piece of data is the hull form. Hull models are generally exported from one software package to another as lines or as faceted surfaces, and not as valid solids (valid means that the object completely encloses a volume) so any representation format must be able to cope with a mixture of surfaces and solid objects. Unfortunately, solids and surfaces are neither interchangeable nor compatible in the representations described in this chapter. This means that a solid cannot be truncated by a surface. Because of the need to model curved surfaces and the difficulty in developing such curves using common Solid models, some sort of hybrid of solids and surface modelling is required.

From the characteristics of the formats described in the previous section, it is clear that there are significant trade-offs between different representation formats. For Semi-Solids it was decided that the emphasis should be placed on the topics of *Accuracy*, *Domain*, *Compactness*, and, *Efficiency*. To this end most of the formulations presented were immediately ruled out leaving only *Space-Partitioning*, *Boundary Representations*, and *Constructive Solid Geometry* as potential formats. Because accuracy requires such a high resolution be used in *Spatial-Occupancy Enumeration*, and because of the computational complexity of *Binary Space Partitioning Trees* it was decided that of this class only *Octrees* would be considered. Further, *Constructive Solid Geometry* was entirely ruled out because of its inability to cope with the automated creation of complex objects — a necessity for an automated Facility Layout algorithm. While the simplicity of the *Octree* representation was recognized, it was also believed that the time required to traverse the *Octree* model for location information would become significant under automation. Similarly, although it is difficult to manage and manipulate the data of a *Boundary Representation*, external factors such as the importation and display of data eventually tipped the scales in favour of this format. Its meshed underpinnings make it ideal for application in current analysis and display software. However, there may be potential for computational speed gains for the Facility Layout process through the application of *Octrees* and other *Spatial Partitioning* formats which should not be overlooked. Hence, under the topic of Future Work in Chapter 7, suggestions for research in this direction were discussed. The critical problem is the logistics of translation between *Boundary Representation* and *Spatially Partitioned* formats, particularly where angled facets are required.

Developed to address some of these concerns, *Semi-Solids* takes its name from its ability to bridge between the surface and solid representations. It falls under the category of Boundary Representations in that a region of space is defined by a boundary comprised of a mesh of 2D facets. Objects are not composites of primitive objects such as cubes and cylinders but are instead complete entities.

The manipulation of objects for the purpose of Facility Layout requires a different process from that of the Boolean combinations generally associated with Solid modelling. Models are to be constructed by projecting the sides of a primitive object onto the surroundings of the new object Then patch by patch, the object could take on the shape of its surroundings — to effectively 'fit' itself against its neighbouring objects. In Chapter 8, a balloon model for Facility Layout was suggested and the representation used here is consistent with that concept.

Hull forms are imported as surfaces and not solids. *Semi-Solids* treats the hull model as just another meshed surface thereby avoiding the difficulty of creating a solid from the imported surface. Each space in the layout is stored as a single entity, whose reference number points to a spatial definition for the space. The drawing database also contains a mesh definition for the referenced object. Interaction between spaces or a space and the hull boundary is carried out by means of a six-step process:

1. Search for neighbouring mesh elements.
2. Determine nature of interaction.
3. Alter the space's boundary mesh to coincide with the other object or surface.
4. Determine and create the patches required to close the adjacent sides of the object.
5. Alter the dimensions of the space to correct for the portions which were added or removed by repeating the process.
6. Remove unnecessary patches from the mesh.

Unlike most surface representations, Semi-Solids takes advantage of the mathematics of the surface of each mesh element. The methods for Semi-Solids are equally appropriate for curved surfaces instead of flat facets which suggests the potential of a relatively simple approach to further increase the accuracy of a modelled surface.

A1.12 Representation Comparison

The three tables (**Table 18**, **Table 19** and **Table 20**) which conclude this chapter show an evaluation of the performance of solid model formulations on the basis of a number of commonly accepted characteristics.[15] Falling under the headings *Accuracy*, *Domain*, *Uniqueness*, *Validity*, *Closure*, and *Completeness*, these characteristics have been used to provide a basis for comparison of the strengths and weaknesses of solid modelling approaches.

The term *Accuracy* refers to a model's ability to represent objects. The *Spatial Partitioning* methods described in the previous section are examples of models which can represent curved objects only to the precision afforded by the size of their primitive unit. Hence curved surfaces are approximated by right-angled steps.

Domain suggests a measure of the capabilities of the model representation to depict objects. The greater the versatility of the model format the greater its domain. For this reason, where curved surfaces and edges are used in *Boundary Representations*, the domain is greater than that of *Constructive Solid Geometry* which is generally limited by the shapes of its primitives.

Where modelled objects can be created in only one configuration of primitives or surface elements, the modelling representation is said to have *Uniqueness*. For example, *Constructive Solid Geometry* tends not to lead to unique solutions because its formulation makes the creation of objects possible by a number of different combinations of primitives and Boolean Set Transactions.

A representation which can ensure *Validity* is one in which each of the objects in the model has a volume. The creation of objects without volume is a problem in model representations

---

15 A.A.G. Requicha, 1980. "Representations for Rigid Solids: Theory, Methods, and Systems", *ACM Computing Surveys*, (Association of Computing Machinery). Volume 12, Number 4.

employing Boolean Operations. For example, given two adjoining cube-shaped objects, a subtraction of the objects leaves a single two-dimensional plane.

Closure refers to the ability of a model to be able to form whole and continuously bounded solids following a number of Boolean transactions. An example of objects for which closure is not possible is the case of *Sweeps* where the union of two swept objects does not necessarily form a new sweep object.

Finally, *Compactness* and *Efficiency* refer to the data and its manipulation in a model formulation. For example, models constructed through *Spatial Partitioning* require a large quantity of data to discretise each primitive object, but require little programmed analysis to carry out Boolean transactions. In general, the terms *Compactness* and *Efficiency* are mutually exclusive. That is, the fewer primitive objects required to define a complex model, the greater computational manipulation is required in their manipulation.

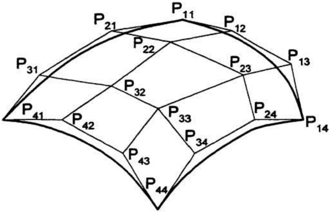


Figure 88 Four-sided Bezier bicubic surface patch showing the 16 required control points.

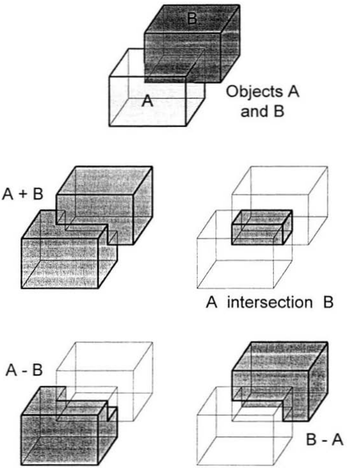


Figure 89 Boolean Operations for two objects. Given objects A and B, the middle left depiction shows $A \cup B$ (effectively $A + B$), the middle right is $A \cap B$, and the lower left and right show $A - B$ and $B - A$ respectively.

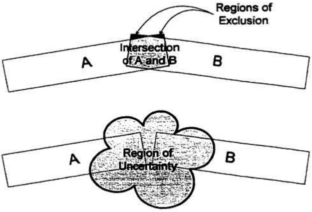
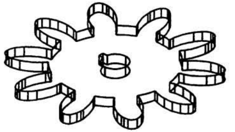


Figure 90 Examples of how Boolean Operations can be effective for identifying the intersection of two objects, but are unable to offer any information in the case where objects are not in contact. As an aside, the *Regions of Exclusion* are impossible to remove without the use of additional objects or without altering the dimensions of the original objects.



Gear Specs

Diameter:	4.0
Axle:	0.5
Thickness:	0.25
Teeth:	10
Key:	Yes

Figure 91 A gear developed through primitive instancing. The data to the right was used to prescribe the solid model.

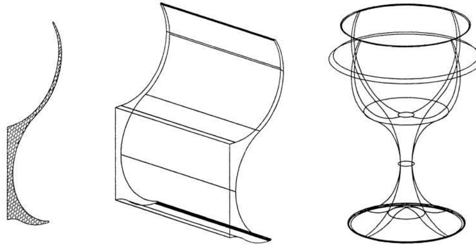
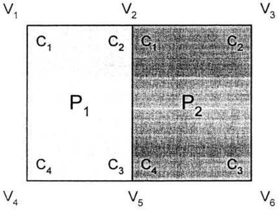


Figure 92 Solids created by translational and rotational sweeps.

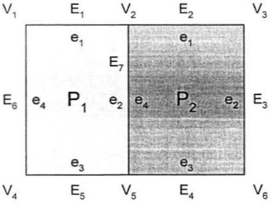


$$V = (V_1, V_2, V_3, V_4, V_5, V_6)$$
$$= ((x_1, y_1, z_1), \ldots, (x_6, y_6, z_6))$$

$$P_1 = (C_1, C_2, C_3, C_4)$$
$$= (V_1, V_2, V_5, V_4)$$

$$P_2 = (C_1, C_2, C_3, C_4)$$
$$= (V_2, V_3, V_6, V_5)$$

Figure 93 A polygon mesh in which each patch is defined by pointers to a single long list of vertices. The vertices in the list are unique, thereby facilitating editing and reducing storage requirements.



$$V = (V_1, V_2, V_3, V_4, V_5, V_6) = ((x_1, y_1, z_1), \ldots, (x_6, y_6, z_6))$$

$$E_1 = (V_1, V_2, P_1, \text{null})$$
$$E_2 = (V_2, V_3, P_2, \text{null})$$
$$E_3 = (V_3, V_6, P_2, \text{null})$$
$$E_4 = (V_5, V_6, P_2, \text{null})$$
$$E_5 = (V_4, V_5, P_1, \text{null})$$
$$E_6 = (V_1, V_4, P_1, \text{null})$$
$$E_7 = (V_2, V_5, P_1, P_2)$$

$$P_1 = (e_1, e_2, e_3, e_4) = (E_1, E_7, E_5, E_6)$$

$$P_2 = (e_1, e_2, e_3, e_4) = (E_2, E_3, E_4, E_7)$$

Figure 94 A polygon mesh in which each facet is defined by pointers to a list of edges. Each edge in the list is unique and in turn contains pointers to a list of unique vertex coordinates. The format is intended to accelerate the depiction of the mesh since shared edges are drawn only once.

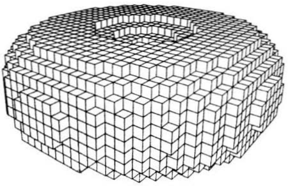


Figure 95 Torus represented by Spatial-occupancy Enumeration.

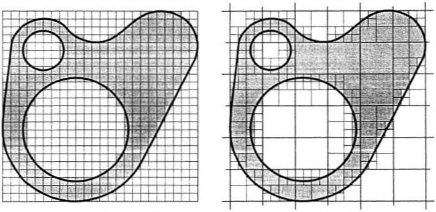


Figure 96 A comparison of Spatial-Occupancy Enumeration and Quadtrees. A Quadtree is the 2D equivalent of an Octree. The Quadtree formulation is able to represent the same object using many fewer cubic units.

Table Pertaining to Appendix 1

Criteria	Primitive Instancing	Sweeps
Accuracy *(refers to the precision by which an object is represented)*	Limited by the accuracy of the underlying coded structure.	Limited to that of the swept object.
Domain *(a measure of the capacity of the model to depict a wide variety of shapes and objects)*	Limited because of the difficulty of programming complex objects.	Limited ability to depict complex objects.
Uniqueness *(where modelled objects can be created in only one configuration of primitives or surface elements)*	Not necessarily. e.g., a sphere may be represented through either sphere or ellipsoid functions.	Not necessarily. e.g., a cube may be swept from any of its faces.
Validity *(refers to the creation of solid objects without volumes)*	Always since there is no object manipulation outside of the underlying coded structure.	Problematic where rotational sweeps circle back on themselves.
Closure *(the ability of a model to be able to form whole and continuously bounded solids)*	Always since objects cannot be used in partial form or in combination.	Can only be closed in Boolean transactions where the same sweep motion is applied to more than one object.
Compactness *(refers to the quantity of data by which objects are modelled)*	Only as compact as the code which defines an object.	As compact as the storage required for the swept object.
Efficiency *(ease by which models are created and depicted — efficiency and compactness are mutually exclusive)*	Not applicable since objects cannot be combined.	Not applicable since objects are rarely combined.

Table 18 Solid model representation comparison — Primitive Instancing and Sweeps.

Criteria	Spatial Partitioning	Constructive Solid Geometry
Accuracy *(refers to the precision by which an object is represented)*	Produces only approximations for objects which are curved or require a finer primitive unit. Resolution can become impractical.	Accurate where primitives are not constructed from polyhedral representations.
Domain *(a measure of the capacity of the model to depict a wide variety of shapes and objects)*	Can represent any solid within the limits of the cube primitive approximations.	Cannot represent high order curves without a template object.
Uniqueness *(where modelled objects can be created in only one configuration of primitives or surface elements)*	Very, as there is only one way to represent an object with a specified size and location.	Not necessarily. Shapes can be produced in a number of combinations of primitives.
Validity *(refers to the creation of solid objects without volumes)*	Almost always valid as a grid cube is either occupied or unoccupied.	Only simple checking is required to catch errors.
Closure *(the ability of a model to be able to form whole and continuously bounded solids)*	Since each primitive is indivisible and closed, the whole is also closed.	Since primitives are bounded so are complex objects.
Compactness *(refers to the quantity of data by which objects are modelled)*	Storage is proportional to the model accuracy and hence the quantity of primitives required for the object.	Very compact since all that need be referenced are the primitives and the applied transactions.
Efficiency *(ease by which models are created and depicted — efficiency and compactness are mutually exclusive)*	Computationally efficient because model merely moves 'blocks'.	An unevaluated model such that CSG evaluates each primitive for each calculation. Therefore changes to primitives are reflected quickly but the format is slow where the model must be evaluated many times.

Table 19 Solid model representation comparison — Spatial Partitioning and Constructive Solid Geometry.

Criteria	Boundary Representations	Semi-Solids (a Boundary Representation)
Accuracy *(refers to the precision by which an object is represented)*	Polygonal Boundary representations may only approximate models. e.g. a faceted sphere. Resolution can become impractical.	Models are approximate because of faceted surfaces but can be improved by decreasing facet size.
Domain *(a measure of the capacity of the model to depict a wide variety of shapes and objects)*	Greatest domain of all representations depending on surface type — e.g. flat facets vs. curved patches and edges.	Wide domain although limited to flat facets.
Uniqueness *(where modelled objects can be created in only one configuration of primitives or surface elements)*	Not unique since a variety of combinations of patches of a great variety of sizes and shapes may be used in a depiction.	Somewhat unique since models begin as simple cubic objects and then take on the shape of their surroundings. Hence, given the surroundings, the same representation will be produced.
Validity *(refers to the creation of solid objects without volumes)*	Most difficult to ensure vertex, edge and face data is consistent. Most difficult to determine interference.	Can be ensured through careful manipulation and error checking. Interference checking is the purpose of the representation.
Closure *(the ability of a model to be able to form whole and continuously bounded solids)*	Can be ensured through careful tracking of boundary elements such as vertices and surfaces.	Can be ensured with careful manipulation and error checking.
Compactness *(refers to the quantity of data by which objects are modelled)*	Moderate storage demands but storage of regular or curved objects is quite compact relative to *Spatial Partitioned* models.	Moderate storage demands but storage of regular or curved objects is quite compact relative to *Spatial Partitioned* models.
Efficiency *(ease by which models are created and depicted — efficiency and compactness are mutually exclusive)*	An evaluated model in that Boolean transactions are reflected in the object's current form. However, more computation may be required for evaluation than might be required for the equivalent *CSG* model.	Efficient for boundary comparisons since the modelled object is in its final form.

Table 20 Solid model representation comparison — Boundary Representations and Semi-Solids.

A2

Appendix 2: Code and Pseudocode

The contents of this Appendix represent the code and pseudocode developed during this research program. Consistent with *Visual Basic*, related functions and subroutines have been grouped into blocks of code called modules which appear below.

Module: Constraint Creation

Sub AddIndex

(tableName As String, indexName As String, keyField As String)

I discovered that a table created by a query does not automatically create table definitions or indexes. This routine uses another SQL statement to create an index for the defined table. Note that this routine is intended to edit only the TemporaryDB variable.

```
Dim IndexQ As QueryDef                                                  dimension IndexQ as a query definition

Set IndexQ = TemporaryDB.CreateQueryDef()                               link IndexQ and the TemporaryDB
IndexQ.Name = tableName & " Table Index Creation - " & indexName        name the new query
IndexQ.SQL = "CREATE INDEX " & indexName & " ON " & tableName           set the SQL information
IndexQ.SQL = IndexQ.SQL & " (" & keyField & ");"

TemporaryDB.QueryDefs.Append IndexQ                                     add the query to the TemporaryDB variable
IndexQ.Execute                                                          run the query
IndexQ.Close                                                            close the query

End Sub
```

Sub AssignSpaceID

This routine adds new entries to the SLTable, and creates a new name for that entry. Where appropriate it provides a Class_ID number.

```
Dim i As Integer                                                        a counter variable
Dim lastRecord As Long                                                  a position marker

Set SOTable = ActiveDB.OpenRecordset("Ship Overall", DB_OPEN_TABLE)
Set CLTable = ActiveDB.OpenRecordset("Class List", DB_OPEN_TABLE)
Set SLTable = ActiveDB.OpenRecordset("Space List", DB_OPEN_TABLE)

SOTable.Index = "Class_ID"                                              set the index of the SOTable to Class_ID
CLTable.Index = "Class_ID"                                              set the index of the CLTable to Class_ID
SLTable.Index = "Space_Name"                                            set the index of the SLTable to Space_Name

SOTable.MoveFirst
Do Until SOTable.EOF                                                    repeat until the name is not found
    CLTable.Seek "=", SOTable.Fields("Class_ID")
    For i = 1 To SOTable.Fields("Quantity")                             loop through the quantity of each space
        SLTable.Seek "=", (CLTable.Fields("Class_Name") & Str$(i))      seek a space name
        If SLTable.NoMatch Then                                         check for repeated name
            lastRecord = SeekLastRecord("ACTIVE", (SLTable.Name))       get the last record number
            SLTable.AddNew                                              add the new entry
            SLTable.Fields("Space_ID") = lastRecord + 1                 increment the Space_ID for the new record
            SLTable.Fields("Space_Name") = (CLTable.Fields("Class_Name") & Str$(i))     name the new space
            SLTable.Fields("Class_ID") = CLTable.Fields("Class_ID")     number the new class
            SLTable.Update                                              complete the entry
        Else
                                                                        space has already been defined
        End If
    Next i
    SOTable.MoveNext
Loop

End Sub
```

Sub CloseConstraintTables

```
CLTable.Close
SLTable.Close
ConstraintsTable.Close

MinTable.Close
PrefTable.Close
MaxTable.Close
ShapeTable.Close

End Sub
```

Sub ConstraintCreationMain

This is the main routine in this module.

The module takes all of the dimensional data in the active database, fills in all the wholes and missing information, and then writes all of this to the temporaryDB.

PrepareTemporaryDB	*Clears, purges and opens the Temporary database*
AssignSpaceID	*Calls a routine which takes each item defined in the Ship Overall List and copies them, with a number, to the Space List Table*
CreateTemporaryTable "Minimum"	*Creates the temporary tables in which all the dimension data is stored.*
CreateTemporaryTable "Preferred" CreateTemporaryTable "Maximum"	
GetShapeData	*Creates a temporary table in which the shape information is stored*
AddIndex "Shape", "PrimaryKey", "Shape_ID"	*Creates indexes for the shape table since Access will not allow this to take place during a Make Table query*
AddIndex "Shape", "Shape_ID", "Shape_ID"	
SetConstraintTables FillConstraintTables	*Puts all the dimensional information into the new tables*
CloseConstraintTables	

End Sub

Sub FillConstraintTables

```
SLTable.MoveFirst
Do Until SLTable.EOF                                        Repeat until the name is not found
     If SLTable.Fields("Class_ID") > 0 Then                 Check to see if the current item is an object such as a
                                                            hull, or a space requiring placement
          GetConstraintRecords (SLTable.Fields("Space_ID")) Calls a routine to get all the table contents
     End If
     SLTable.MoveNext
Loop

End Sub
```

Sub CreateTemporaryTable

(tableName As String)

This routine creates temporary tables containing minimum, maximum and preferred dimension values for each space

The routine is fairly self-explanatory. It essentially creates each of the elements of the table (Fields and Indices) and appends these to the newTableDef definition. In turn, this definition is appended to the TemporaryDB.TableDefs collection, thus creating the tables.

```
ReDim f(6) As New Field
ReDim i(7) As New Index
Dim newTblDef As New TableDef

f(1).Name = "Space_ID"                              ' Create fields
f(1).Type = DB_LONG

f(2).Name = "Length"
f(2).Type = DB_DOUBLE

f(3).Name = "Width"
f(3).Type = DB_DOUBLE

f(4).Name = "Height"
f(4).Type = DB_DOUBLE

f(5).Name = "Area"
f(5).Type = DB_DOUBLE

f(6).Name = "Volume"
f(6).Type = DB_DOUBLE

i(1).Name = "PrimaryKey"                            ' Create indices
i(1).Fields = "Space_ID"
i(1).primary = True
newTblDef.Indexes.Append I(1)                       ' Add it to the collection

For j = 1 To 6
    newTblDef.Fields.Append f(j)                    ' Add it to the collection

    i(j + 1).Name = f(j).Name
    i(j + 1).Fields = f(j).Name
    i(j + 1).primary = False
    newTblDef.Indexes.Append i(j + 1)               ' Add it to the collection
Next j

newTblDef.Name = tableName                          ' Name the new table
TemporaryDB.TableDefs.Append newTblDef              ' Now append the new Table object to the TableDefs collection.

End Sub
```

Sub GetConstraintRecords

(Space_ID As Long)

This routine crudely updates dimensional data in the database. In future it is to be replaced by a knowledge-based system.

```
Dim lengthR As DimensionSet                                      Set dimension variables
Dim widthR As DimensionSet
Dim heightR As DimensionSet
Dim areaR As DimensionSet
Dim volumeR As DimensionSet

SLTable.Index = "Space_ID"  Find the current space_ID in the Space List table
SLTable.Seek "=", Space_ID
CLTable.Index = "PrimaryKey"                                     Find the current Class_ID in the Class List table
CLTable.Seek "=", SLTable.Fields("Class_ID")
ConstraintsTable.Index = "PrimaryKey"
ConstraintsTable.Seek "=", CLTable.Fields("Constraints_ID")      Find the current Constraints_ID entry in the
                                                                 Constraints table

lengthFlag = ConstraintsTable.Fields("Length")                   Get the parameter flags
widthFlag = ConstraintsTable.Fields("Width")
heightFlag = ConstraintsTable.Fields("Height")
areaFlag = ConstraintsTable.Fields("Area")
volumeFlag = ConstraintsTable.Fields("Volume")

ShapeTable.Index = "PrimaryKey"                                  Find the current Shape_ID in the Shape Table in the
                                                                 TemporaryDB

ShapeTable.Seek "=", ConstraintsTable.Fields("Shape_ID")
floatingARFlag = ShapeTable.Fields("Floating_Aspect_Ratio")      Store the Fixed_Aspect_Ratio value
aspectRatio = ShapeTable.Fields("Aspect_Ratio")                  Store the Aspect_Ratio value

If lengthFlag = True Then GetDimension "Length", lengthR, (ConstraintsTable.Fields("Length_ID"))  Read in the dimensions for
                                                                                                   each flag value
If widthFlag = True Then GetDimension "Width", widthR, (ConstraintsTable.Fields("Width_ID"))
If heightFlag = True Then GetDimension "Height", heightR, (ConstraintsTable.Fields("Height_ID"))
If areaFlag = True Then GetDimension "Area", areaR, (ConstraintsTable.Fields("Area_ID"))
If volumeFlag = True Then GetDimension "Volume", volumeR, (ConstraintsTable.Fields("Volume_ID"))

If ((lengthFlag = True) And (widthFlag = True) And (heightFlag = True)) Then

    areaR.min = lengthR.min * widthR.min
    areaR.pref = lengthR.pref * widthR.pref
    areaR.max = lengthR.max * widthR.max

    volumeR.min = lengthR.min * widthR.min * heightR.min
    volumeR.pref = lengthR.pref * widthR.pref * heightR.pref
    volumeR.max = lengthR.max * widthR.max * heightR.max

ElseIf ((lengthFlag = True) And (widthFlag = True) And (volumeFlag = True)) Then

    areaR.min = lengthR.min * widthR.min
    areaR.pref = lengthR.pref * widthR.pref
    areaR.max = lengthR.max * widthR.max

    heightR.min = volumeR.min / (lengthR.min * widthR.min)
    heightR.pref = volumeR.pref / (lengthR.pref * widthR.pref)
    heightR.max = volumeR.max / (lengthR.max * widthR.max)
```

```
ElseIf ((lengthFlag = True) And (heightFlag = True) And (volumeFlag = True)) Then

    widthR.min = volumeR.min / (lengthR.min * heightR.min)
    widthR.pref = volumeR.pref / (lengthR.pref * heightR.pref)
    widthR.max = volumeR.max / (lengthR.max * heightR.max)

    areaR.min = lengthR.min * widthR.min
    areaR.pref = lengthR.pref * widthR.pref
    areaR.max = lengthR.max * widthR.max

ElseIf ((lengthFlag = True) And (areaFlag = True) And (volumeFlag = True)) Then

    heightR.min = volumeR.min / areaR.min
    heightR.pref = volumeR.pref / areaR.min
    heightR.max = volumeR.max / areaR.min

    widthR.min = areaR.min / lengthR.min
    widthR.pref = areaR.pref / lengthR.pref
    widthR.max = areaR.max / lengthR.max

ElseIf ((widthFlag = True) And (heightFlag = True) And (areaFlag = True)) Then

    lengthR.min = areaR.min / widthR.min
    lengthR.pref = areaR.pref / widthR.pref
    lengthR.max = areaR.max / widthR.max

    volumeR.min = lengthR.min * widthR.min * heightR.min
    volumeR.pref = lengthR.pref * widthR.pref * heightR.pref
    volumeR.max = lengthR.max * widthR.max * heightR.max

ElseIf ((widthFlag = True) And (heightFlag = True) And (volumeFlag = True)) Then

    lengthR.min = volumeR.min / (widthR.min * heightR.min)
    lengthR.pref = volumeR.pref / (widthR.pref * heightR.pref)
    lengthR.max = volumeR.max / (widthR.max * heightR.max)

    areaR.min = lengthR.min * widthR.min
    areaR.pref = lengthR.pref * widthR.pref
    areaR.max = lengthR.max * widthR.max

ElseIf ((widthFlag = True) And (areaFlag = True) And (volumeFlag = True)) Then

    heightR.min = volumeR.min / areaR.min
    heightR.pref = volumeR.pref / areaR.min
    heightR.max = volumeR.max / areaR.min

    lengthR.min = areaR.min / widthR.min
    lengthR.pref = areaR.pref / widthR.pref
    lengthR.max = areaR.max / widthR.max


ElseIf (floatingARFlag = False) Then
    If (lengthFlag = True) And (widthR.pref = Null) Then
        widthR.min = lengthR.min / aspectRatio
        widthR.pref = lengthR.pref / aspectRatio
        widthR.max = lengthR.max / aspectRatio
    ElseIf (widthFlag = True) And (lengthR.pref = Null) Then
        lengthR.min = widthR.min * aspectRatio
        lengthR.pref = widthR.pref * aspectRatio
        lengthR.max = widthR.max * aspectRatio
    ElseIf (lengthFlag = False) And (widthFlag = False) Then
```

```
                If (areaFlag = True) Then
                    widthR.min = (areaR.min / aspectRatio) ^ (.5)
                    widthR.pref = (areaR.pref / aspectRatio) ^ (.5)
                    widthR.max = (areaR.max / aspectRatio) ^ (.5)

                    lengthR.min = (areaR.min * aspectRatio) ^ (.5)
                    lengthR.pref = (areaR.pref * aspectRatio) ^ (.5)
                    lengthR.max = (areaR.max * aspectRatio) ^ (.5)
                ElseIf (volumeFlag = True) And (heightFlag = True) Then
                    widthR.min = (volumeR.min / heightFlag / aspectRatio) ^ (.5)
                    widthR.pref = (volumeR.pref / heightFlag / aspectRatio) ^ (.5)
                    widthR.max = (volumeR.max / heightFlag / aspectRatio) ^ (.5)

                    lengthR.min = (volumeR.min / heightFlag * aspectRatio) ^ (.5)
                    lengthR.pref = (volumeR.pref / heightFlag * aspectRatio) ^ (.5)
                    lengthR.max = (volumeR.max / heightFlag * aspectRatio) ^ (.5)
                End If
            End If

    ElseIf (floatingARFlag = True) Then
        If (lengthFlag = True) And (widthR.pref = Null) Then
            widthR.min = lengthR.min / aspectRatio
            widthR.pref = lengthR.pref / (aspectRatio / 2)
            widthR.max = lengthR.max
        ElseIf (widthFlag = True) And (lengthR.pref = Null) Then
            lengthR.min = widthR.min
            lengthR.pref = widthR.pref * (aspectRatio / 2)
            lengthR.max = widthR.max * aspectRatio
        ElseIf (lengthFlag = False) And (widthFlag = False) Then
            If (areaFlag = True) Then
                widthR.min = (areaR.min / aspectRatio) ^ (.5)
                widthR.pref = (areaR.pref / aspectRatio) ^ (.5)
                widthR.max = (areaR.max / aspectRatio) ^ (.5)

                lengthR.min = (areaR.min * aspectRatio) ^ (.5)
                lengthR.pref = (areaR.pref * aspectRatio) ^ (.5)
                lengthR.max = (areaR.max * aspectRatio) ^ (.5)
            ElseIf (volumeFlag = True) And (heightFlag = True) Then
                widthR.min = (volumeR.min / heightFlag / aspectRatio) ^ (.5)
                widthR.pref = (volumeR.pref / heightFlag / aspectRatio) ^ (.5)
                widthR.max = (volumeR.max / heightFlag / aspectRatio) ^ (.5)

                lengthR.min = (volumeR.min / heightFlag * aspectRatio) ^ (.5)
                lengthR.pref = (volumeR.pref / heightFlag * aspectRatio) ^ (.5)
                lengthR.max = (volumeR.max / heightFlag * aspectRatio) ^ (.5)
            End If
        End If

        If volumeR.pref = 0 Then
            volumeR.min = lengthR.min * widthR.min * heightR.min
            volumeR.pref = lengthR.pref * widthR.pref * heightR.pref
            volumeR.max = lengthR.max * widthR.max * heightR.max
        End If
    End If

    MinTable.AddNew                                        Send values to the tables in the TemporaryDB
        MinTable.Fields("Space_ID") = Space_ID
        MinTable.Fields("Length") = lengthR.min
        MinTable.Fields("Width") = widthR.min
        MinTable.Fields("Height") = heightR.min
        MinTable.Fields("Area") = areaR.min
```

```
        MinTable.Fields("Volume") = volumeR.min
    MinTable.Update

    PrefTable.AddNew
        PrefTable.Fields("Space_ID") = Space_ID
        PrefTable.Fields("Length") = lengthR.pref
        PrefTable.Fields("Width") = widthR.pref
        PrefTable.Fields("Height") = heightR.pref
        PrefTable.Fields("Area") = areaR.pref
        PrefTable.Fields("Volume") = volumeR.pref
    PrefTable.Update

    MaxTable.AddNew
        MaxTable.Fields("Space_ID") = Space_ID
        MaxTable.Fields("Length") = lengthR.max
        MaxTable.Fields("Width") = widthR.max
        MaxTable.Fields("Height") = heightR.max
        MaxTable.Fields("Area") = areaR.max
        MaxTable.Fields("Volume") = volumeR.max
    MaxTable.Update

    If ShapeTable.Fields("Aspect_Ratio") = Null Then
        ShapeTable.Edit
            ShapeTable.Fields("Aspect_Ratio") = lengthR.pref / widthR.pref
        ShapeTable.Update
    End If

    End Sub
```

Sub SetConstraintTables

Assign table variables for the tables in the ActiveDB

```
Set SLTable = ActiveDB.OpenRecordset("Space List", DB_OPEN_TABLE)
Set CLTable = ActiveDB.OpenRecordset("Class List", DB_OPEN_TABLE)
Set ConstraintsTable = ActiveDB.OpenRecordset("Constraints", DB_OPEN_TABLE)
```

Assign table variables for the tables in the TemporaryDB

```
Set MinTable = TemporaryDB.OpenRecordset("Minimum", DB_OPEN_TABLE)
Set PrefTable = TemporaryDB.OpenRecordset("Preferred", DB_OPEN_TABLE)
Set MaxTable = TemporaryDB.OpenRecordset("Maximum", DB_OPEN_TABLE)
Set ShapeTable = TemporaryDB.OpenRecordset("Shape", DB_OPEN_TABLE)


End Sub
```

Sub GetShapeData

This routine creates a tableQuery which stores a list of shape data in the TemporaryDB.

```
Dim ShapeQ As QueryDef
```

The name of a query definition which creates a list of all the patches associated with a particular Space_ID number

```
Set ShapeQ = ActiveDB.CreateQueryDef()
ShapeQ.Name = "Shape"

On Error Resume Next
    ActiveDB.QueryDefs.Delete ShapeQ.Name
On Error GoTo 0
```

Wipe out old QueryDefs

```
ShapeQ.SQL = "SELECT DISTINCTROW [Space List].Space_ID, [Constraints Shape].* "
ShapeQ.SQL = ShapeQ.SQL & "INTO [Shape] "
ShapeQ.SQL = ShapeQ.SQL & "IN " & Chr$(34) & TemporaryDB.Name & Chr$(34) & " "
ShapeQ.SQL = ShapeQ.SQL & "FROM ([Constraints Shape] "
ShapeQ.SQL = ShapeQ.SQL & "INNER JOIN [Constraints] "
ShapeQ.SQL = ShapeQ.SQL & "ON [Constraints Shape].Shape_ID = [Constraints].Shape_ID) "
ShapeQ.SQL = ShapeQ.SQL & "INNER JOIN ([Class List] "
ShapeQ.SQL = ShapeQ.SQL & "INNER JOIN [Space List] "
ShapeQ.SQL = ShapeQ.SQL & "ON [Class List].Class_ID = [Space List].Class_ID) "
ShapeQ.SQL = ShapeQ.SQL & "ON [Constraints].Constraints_ID = [Class List].Constraints_ID;"

ActiveDB.QueryDefs.Append ShapeQ
ShapeQ.Execute
ShapeQ.Close

End Sub
```

Sub GetDimension

(dimName As String, dimR As DimensionSet, ID As Long)

This routine is called by the GetConstraintRecord routine. It is called when a flag has been found for the use of a specific dimension.

```
Dim tempTable As Recordset

Set tempTable = ActiveDB.OpenRecordset(("Constraints " & dimName), DB_OPEN_TABLE)  Set the temporary table variable

tempTable.Index = "PrimaryKey"                                     Find the current row of the temp table
tempTable.Seek "=", ID

dimR.pref = tempTable.Fields("Preferred")                          Assign the preferred value
If tempTable.Fields("Fixed") = True Then                           Check for a fixed dimension
    dimR.min = dimR.pref
    dimR.max = dimR.pref
Else                                                               If not fixed...
    If tempTable.Fields("Minimum_by_Contents") = True Then         And if minimum is to be calculated from the room
                                                                   contents then...
        dimR.min = GetMinimumbyContents                            TO BE IMPLEMENTED LATER
    ElseIf tempTable.Fields("Minimum_by_Percentage") = True Then   Or if minimum is to be calculated from a percentage
        dimR.min = dimR.pref * tempTable.Fields("Minimum_Percentage") / 100
    Else                                                           Otherwise use the min value
        dimR.min = tempTable.Fields("Minimum")
    End If

    If tempTable.Fields("Maximum_by_Percentage") = True Then       And if the maximum is to be calculated from a
                                                                   percentage
        dimR.max = dimR.pref * tempTable.Fields("Maximum_Percentage") / 100
    Else
        dimR.max = tempTable.Fields("Maximum")                     Otherwise use the min value
    End If
End If

End Sub
```

Module: Patch Table Fillers

Sub Adjacentcies

This routine is used to fill in entries in the Patch Adjacentcy table. It figures out if a patch is beside a particular patch, and then stores the ID value for the adjacent patch in a record identified by a Patch_ID. Therefore a record exists for each patch, and in each record is stored the Patch_ID values for the four patches which are directly adjacent to the patch.

Note that throughout this model it has been assumed that no more than a single patch can adjoin an edge. The image depicted below is invalid.

```
 __  __________
   |     |     |
   |  2  |  3  |
 __|_____|_____|__
   |           |
   |     1     |
 __|___________|__
```

This routine assumes that patch 2 will completely share an edge with patch 1.

```
Dim vertex1 As Long
```

These values store the vertex pointers of a particular patch

```
Dim vertex2 As Long
Dim vertex3 As Long
Dim vertex4 As Long

Set AdjTable = ActiveDB.OpenRecordset("Patch Adjacency", DB_OPEN_TABLE)
```

Set the tables used in this routine

```
Set CTable = ActiveDB.OpenRecordset("Patch Corners", DB_OPEN_TABLE)
Set PTable = ActiveDB.OpenRecordset("Patch List", DB_OPEN_TABLE)

PTable.MoveFirst
CTable.Index = "PrimaryKey"
```

Set the Patch Corners table index to the Patch_ID value

```
AdjTable.Index = "PrimaryKey"

ActiveDB.BeginTrans
```

Use transactions to hasten this routine

```
Do Until Ptable.EOF
```

Loop until the end of the patches table is reached

```
    AdjTable.Seek "=", PTable.Fields("Patch_ID")
    If AdjTable.NoMatch Then
        AdjTable.AddNew
```

Update the EqTable with the new entry

```
    Else
        AdjTable.Edit
    End If

    AdjTable.Fields("Patch_ID") = PTable.Fields("Patch_ID")

    CTable.Index = "PrimaryKey"
```

Get the corner pointer data from the CTable

```
    CTable.Seek "=", PTable.Fields("Patch_ID")
    vertex1 = CTable.Fields("Vertex1")
    vertex2 = CTable.Fields("Vertex2")
    vertex3 = CTable.Fields("Vertex3")
    vertex4 = CTable.Fields("Vertex4")

    CTable.Index = "Vertex4"
```

Seek patch adjacent to side 14

```
    CTable.Seek "=", vertex1
    If CTable.NoMatch Then
```

```
            AdjTable.Fields("Patch1") = Null
        Else
            AdjTable.Fields("Patch1") = CTable.Fields("Patch_ID")
        End If

        CTable.Index = "Vertex1"                          Seek patch adjacent to side 23
        CTable.Seek "=", vertex2
        If CTable.NoMatch Then
            AdjTable.Fields("Patch2") = Null
        Else
            AdjTable.Fields("Patch2") = CTable.Fields("Patch_ID")
        End If

        CTable.Index = "Vertex1"                          Seek patch adjacent to side 34
        CTable.Seek "=", vertex4
        If CTable.NoMatch Then
            AdjTable.Fields("Patch3") = Null
        Else
            AdjTable.Fields("Patch3") = CTable.Fields("Patch_ID")
        End If

        CTable.Index = "Vertex2"                          Seek patch adjacent to side 41
        CTable.Seek "=", vertex1
        If CTable.NoMatch Then
            AdjTable.Fields("Patch4") = Null
        Else
            AdjTable.Fields("Patch4") = CTable.Fields("Patch_ID")
        End If

        AdjTable.Update
        PTable.MoveNext
    Loop

    ActiveDB.CommitTrans

    AdjTable.Close
    CTable.Close
    PTable.Close

    End Sub
```

Sub Equations

This routine generates the a, b, c and d equation parameters for each patch and stores them in the Patch Equations table.

The equation takes the form of: $aX + bY + cZ + d = 0$

```
Dim i As Integer                    An indexing variable
Dim a As Double                     Equation parameters
Dim b As Double
Dim c As Double
Dim d As Double
ReDim x(3) As Double                Arrays of coordinate information used to derive a patch

ReDim y(3) As Double
ReDim z(3) As Double


Set PTable = ActiveDB.OpenRecordset("Patch List", DB_OPEN_TABLE)
Set CTable = ActiveDB.OpenRecordset("Patch Corners", DB_OPEN_TABLE)      Assign variables for the tables used in this routine

Set EqTable = ActiveDB.OpenRecordset("Patch Equation", DB_OPEN_TABLE)
Set VTable = ActiveDB.OpenRecordset("Vertex List", DB_OPEN_TABLE)

VTable.Index = "PrimaryKey"         Set the indices of the tables being searched to their ID values

CTable.Index = "PrimaryKey"
EqTable.Index = "PrimaryKey"
PTable.MoveFirst

ActiveDB.BeginTrans                 Begin a transaction to facilitate the efficiency of the routine.


Do Until Ptable.EOF                 Examine the entire Patch List table

    CTable.Seek "=", Ptable.Fields("Patch_ID")      Move to the entry in the corner table corresponding to the current patch_ID entry in the PTable

    For i = 1 To 3                  Loop through the first three corners
        VTable.Seek "=", Ctable.Fields(i)           Seek the vertex coordinate data for the particular corner pointer from the VTable

        x(i) = VTable.Fields("x")
        y(i) = VTable.Fields("y")
        z(i) = VTable.Fields("z")
    Next i

    a = y(1) * (z(2) - z(3))        Generate the equation variables
    a = a - z(1) * (y(2) - y(3))
    a = a + (y(2) * z(3) - y(3) * z(2))

    b = x(1) * (z(2) - z(3))
    b = b - z(1) * (x(2) - x(3))
    b = b + (x(2) * z(3) - x(3) * z(2))
    b = b * (-1)

    c = x(1) * (y(2) - y(3))
    c = c - y(1) * (x(2) - x(3))
    c = c + (x(2) * y(3) - y(2) * x(3))

'   d = x(1) * (y(2) * z(3) - z(2) * y(3))
'   d = d - y(1) * (x(2) * z(3) - x(3) * z(2))
```

```
'   d = d + z(1) * (x(2) * y(3) - x(3) * y(2))
'   d = d * (-1)

    a = a / (a ^ 2 + b ^ 2 + c ^ 2) ^ .5         Make normal values 'unit normals'
    b = b / (a ^ 2 + b ^ 2 + c ^ 2) ^ .5
    c = c / (a ^ 2 + b ^ 2 + c ^ 2) ^ .5

    d = -1 * (a * x(1) + b * y(1) + c * z(1))

    EqTable.Seek "=", PTable.Fields("Patch_ID")
    If EqTable.NoMatch Then
        EqTable.AddNew                           Update the EqTable with the new entry
    Else
        EqTable.Edit
    End If

    EqTable.Fields("Patch_ID") = PTable.Fields("Patch_ID")

    EqTable.Fields("a") = a
    EqTable.Fields("b") = b
    EqTable.Fields("c") = c
    EqTable.Fields("d") = d

    EqTable.Update
    CTable.MoveNext
    PTable.MoveNext
Loop

ActiveDB.CommitTrans                             Finish the transaction

Ctable.Close                                     Clear references to the database tables.
EqTable.Close
PTable.Close
VTable.Close

End Sub
```

Sub HiddenEdges

This routine examines the mesh contained in the database and determines instances in which the boundaries of a patch could be double written. It then sets flags to indicate that one of the edges should be stored as a hidden edge.

For instances where hidden edge information has been collected by the DXFImport and DXFDigest routines, the hidden edge flags are overwritten. The reason for this is that Autocad appears to be inconsistent in the flagging order of its hidden patches. Here I have assumed that only sides 3 and 4 can be blanked when there is an adjoining patch.

The routine is not necessary for the operation of this database, but makes for a cleaner .DXF output.

This routine will require updating since it currently treats all of the entries in the Patch List table as parts of a single mesh. This will undoubtably cause future errors, and should therefore be revisited. Nesting this routine within one which points to individual spaces should solve the problem.

A second problem with this routine which will require further work lies in the assumption that the HETable already contains Patch_ID entries for all the patches in the Patch List (PTable). Once the creation of spaces and patches in subsequent routines is completed, entries will exist in other tables which should be reflected in this one.

Set AdjTable = ActiveDB.OpenRecordset("Patch Adjacency", DB_OPEN_TABLE)	*Assign variables to the tables utilized by this routine*
Set HETable = ActiveDB.OpenRecordset("Patch Hidden Edges", DB_OPEN_TABLE) Set PTable = ActiveDB.OpenRecordset("Patch List", DB_OPEN_TABLE)	
PTable.Index = "PrimaryKey" PTable.MoveFirst AdjTable.Index = "PrimaryKey" HETable.Index = "PrimaryKey"	*Set the indexes of the tables to be searched*
ActiveDB.BeginTrans	*Begin the database transaction*
Do Until Ptable.EOF	*Scan through the entire patch list*
AdjTable.Seek "=", Ptable.Fields("Patch_ID")	*Find the entry in the*
HETable.Seek "=", AdjTable.Fields("Patch_ID") HETable.Edit	
If HETable.Fields("Edge1") Then HETable.Fields("Edge1") = False	*Test to see if changes are required for the current Edge1 entry*
If HETable.Fields("Edge2") Then HETable.Fields("Edge2") = False	*Test to see if changes are required for the current Edge2 entry*
If AdjTable.Fields("Patch3") Then HETable.Fields("Edge3") = True	*Test to see if there is a patch adjacent to Edge3*
If AdjTable.Fields("Patch4") Then HETable.Fields("Edge4") = True	*Test to see if there is a patch adjacent to Edge4*
HETable.Update	*Complete the record*
Ptable.MoveNext Loop	*Move to the next patch*
ActiveDB.CommitTrans	*Complete the transaction*
AdjTable.Close HETable.Close PTable.Close	*Clear the table variables*
End Sub	

Sub KillVertexRepeats

(db As String, vTempTableName As String, pTempTableName As String)

This routine identifies repeated vertices contained in the given Vertex table. It then uses the sorted PatchSet(i) dynasets to quickly update the patch corner pointers contained in the Patches Surface table. The routine then deletes the unnecessary entries.

This routine could be made more efficient by enabling it to kill repeats on a space by space basis — so that the entire vertex dataset is not examined every time the routine runs.

```
Dim pointer0 As Long                                    Pointer to the first entry
Dim pointer1 As Long                                    Pointer to the second entry
Dim pt0 As Point3DDouble                                Coordinate values of the first entry
Dim pt1 As Point3DDouble                                Coordinate values of the second entry
Dim vTempTable As Recordset
Dim pTempTable As Recordset

If db = "ACTIVE" Then                                   Test to determine which database the tableName
                                                        should be associated
    Set vTempTable = ActiveDB.OpenRecordset(vTempTableName, DB_OPEN_TABLE)    Assign the tempTable variable
    Set pTempTable = ActiveDB.OpenRecordset(pTempTableName, DB_OPEN_TABLE)    Assign the tempTable variable
    ActiveDB.BeginTrans
ElseIf db = "TEMPORARY" Then
    Set vTempTable = TemporaryDB.OpenRecordset(vTempTableName, DB_OPEN_TABLE)
    Set pTempTable = TemporaryDB.OpenRecordset(pTempTableName, DB_OPEN_TABLE)    Assign the tempTable
                                                                                  variable
    TemporaryDB.BeginTrans
End If

If (vTempTable.BOF And vTempTable.EOF) Then             Test to see if any Vertices have been stored. If not...
Else                                                    If so then...

    vTempTable.Index = "XYZ"
    vTempTable.MoveFirst                                Move to the first item in the VTempTable
    pointer0 = vTempTable.Fields("Vertex_ID")           Assign the Vertex_ID to pointer0
    pt0.x = vTempTable.Fields("X")                      Assign the vertex coordinates to pt0
    pt0.y = vTempTable.Fields("Y")
    pt0.z = vTempTable.Fields("Z")

    vTempTable.MoveNext                                 Move to the next item in the VTempTable
    Do Until vTempTable.EOF                             Repeat until the VTempTable is exhausted
        pointer1 = vTempTable.Fields("Vertex_ID")       Assign the Vertex_ID to pointer1
        pt1.x = vTempTable.Fields("X")                  Assign the vertex coordinates to pt1
        pt1.y = vTempTable.Fields("Y")
        pt1.z = vTempTable.Fields("Z")

        If EqualPts(pt0, pt1) Then                      Test to see if pt1 is a repetition of pt0
            For i = 1 To 4                              Begin looping through the 4 fields
                pTempTable.Index = pTempTable.Fields(i).Name    Find the first instance of pointer1 in the PatchSet
                pTempTable.Seek "=", pointer1           Find the first instance of pointer1 in the PatchSet
                Do Until pTempTable.NoMatch             Begin looping until no other instances of pointer1
                                                        appear in Field i
                    pTempTable.Edit                     Allow editing of the PatchSet
                    pTempTable.Fields(i) = pointer0     Replace pointer1 with pointer0
                    pTempTable.Update                   Update the PatchSet and the PTempTable
                    pTempTable.Seek "=", pointer1       Find the next instance of pointer1 in the PatchSet
                Loop
            Next i
            vTempTable.Delete                           Delete the repeated item in the VTempTable
        Else
```

```
            pointer0 = pointer1                          Copy pointer1 to pointer
            CopyPts pt1, pt0                             Assign new point to old point
        End If

        vTempTable.MoveNext                              Move to the next item in the VTempTable
    Loop

    If db = "ACTIVE" Then                                Test to determine which database the tableName
                                                         should be associated
        ActiveDB.CommitTrans
    ElseIf db = "TEMPORARY" Then
        TemporaryDB.CommitTrans
    End If

End If

pTempTable.Close                                         Clear the table variables
vTempTable.Close

End Sub
```

Sub Renumber

(db As String, tableName As String)

This routine renumbers the ID values in the tempTable table. The update of related values is carried out by means of the relationship between linked tables. The relationship is a one to many with a cascade update and delete.

Note that the routine is generic to whatever table is passed to it.

```
Dim counter As Long
Dim tempTable As Recordset

If db = "ACTIVE" Then                                          Test to determine which database the tableName
                                                               should be associated
    Set tempTable = ActiveDB.OpenRecordset(tableName, DB_OPEN_TABLE)    Assign the tempTable variable
ElseIf db = "TEMPORARY" Then
    Set tempTable = TemporaryDB.OpenRecordset(tableName, DB_OPEN_TABLE)
End If

If (tempTable.BOF And tempTable.EOF) Then                      Test to see if the table is empty.
Else                                                           Otherwise...

    ActiveDB.BeginTrans

    counter = 1                                                Set the counter to 1
    tempTable.Index = "PrimaryKey"
    tempTable.MoveFirst                                        Move to the first record in the tempTable
    Do Until tempTable.EOF                                     Loop until the end of the tempTable is reached
        tempTable.Edit                                         Allow editing of the current record
        tempTable.Fields(0) = counter                          Replace the ID value with the counter value
        tempTable.Update                                       Save the record changes
        tempTable.MoveNext                                     Move to the next record
        counter = counter + 1                                  Increment the counter
    Loop

    ActiveDB.CommitTrans

End If

End Sub
```

Module: Patch Tests

```
Option Compare Database                                        ' Use database order for string comparisons
Dim VIPOIPTable As Recordset
```

Sub TestMain

Checks to see if new patch corners violate exterior boundaries.

This routine creates a tableQuery which stores a list of patches associated with a particular reference (or Space_ID) ID value.

Finally, the routine assigns a variable to the new table.

```
Dim eq As Equation

TempPTable.MoveFirst
'Do Until TempPTable.EOF                                        Repeat until all six faces have been created

'    GetTempEqValues (TempPTable.Fields("Patch_ID")), eq

     Test1_POIData (TempPTable.Fields("Patch_ID"))               Generate Prism Data for POI
     Test2_VintoPOI (TempPTable.Fields("Patch_ID"))              Substitute all Vertices into POI Prism equations
     Test3_VertexZone                                            Determine Position Zones
     Test4_PatchesToConsider (TempPTable.Fields("Patch_ID"))     Collect Data in terms of dataset patches
     Test5_PatchestoExclude                                      Remove patches whose points lie wholly outside a
                                                                 Prism boundary plane

...Exclusion process

'Set VIPOIPTable = TemporaryDB.OpenRecordset("Vertices Inside POI Prism", DB_OPEN_TABLE)

'    TempPTable.MoveNext
'Loop

End Sub
```

Sub Test1_POIData

(POI_ID As Long)

This routine creates a query which generates a set of equation parameters for planes which are perpendicular to the POI.

```
Dim TempQ As New QueryDef

Set TempQ = TemporaryDB.CreateQueryDef()
TempQ.Name = "Interference - Data - POI"

On Error Resume Next
TemporaryDB.QueryDefs.Delete TempQ.Name
On Error GoTo 0

TempQ.SQL = "SELECT DISTINCTROW [Temporary Patches].Patch_ID, "
TempQ.SQL = TempQ.SQL & "[Temporary Vertices_1].Vertex_ID, [Temporary Vertices_1].X AS x1, "
TempQ.SQL = TempQ.SQL & "[Temporary Vertices_1].Y AS y1, [Temporary Vertices_1].Z AS z1, "
TempQ.SQL = TempQ.SQL & "[Temporary Vertices_2].Vertex_ID, [Temporary Vertices_2].X AS x2, "
TempQ.SQL = TempQ.SQL & "[Temporary Vertices_2].Y AS y2, [Temporary Vertices_2].Z AS z2, "
TempQ.SQL = TempQ.SQL & "[Temporary Vertices_3].Vertex_ID, [Temporary Vertices_3].X AS x3, "
TempQ.SQL = TempQ.SQL & "[Temporary Vertices_3].Y AS y3, [Temporary Vertices_3].Z AS z3, "
TempQ.SQL = TempQ.SQL & "[Temporary Vertices_4].Vertex_ID, [Temporary Vertices_4].X AS x4, "
TempQ.SQL = TempQ.SQL & "[Temporary Vertices_4].Y AS y4, [Temporary Vertices_4].Z AS z4, "
TempQ.SQL = TempQ.SQL & "[Temporary Equations].a, [Temporary Equations].b, "
TempQ.SQL = TempQ.SQL & "[Temporary Equations].c, [Temporary Equations].d, "
TempQ.SQL = TempQ.SQL & "(([z2]-[z1])*[b]-[c]*([y2]-[y1])) AS Eq_21_a, "
TempQ.SQL = TempQ.SQL & "-1*(([z2]-[z1])*[a]-[c]*([x2]-[x1])) AS Eq_21_b, "
TempQ.SQL = TempQ.SQL & "(([y2]-[y1])*[a]-[b]*([x2]-[x1])) AS Eq_21_c, "
TempQ.SQL = TempQ.SQL & "-1*([Eq_21_a]*[x1]+[Eq_21_b]*[y1]+[Eq_21_c]*[z1]) AS Eq_21_d, "
TempQ.SQL = TempQ.SQL & "(([z3]-[z2])*[b]-[c]*([y3]-[y2])) AS Eq_32_a, "
TempQ.SQL = TempQ.SQL & "-1*(([z3]-[z2])*[a]-[c]*([x3]-[x2])) AS Eq_32_b, "
TempQ.SQL = TempQ.SQL & "(([y3]-[y2])*[a]-[b]*([x3]-[x2])) AS Eq_32_c, "
TempQ.SQL = TempQ.SQL & "-1*([Eq_32_a]*[x2]+[Eq_32_b]*[y2]+[Eq_32_c]*[z2]) AS Eq_32_d, "
TempQ.SQL = TempQ.SQL & "(([z4]-[z3])*[b]-[c]*([y4]-[y3])) AS Eq_43_a, "
TempQ.SQL = TempQ.SQL & "-1*(([z4]-[z3])*[a]-[c]*([x4]-[x3])) AS Eq_43_b, "
TempQ.SQL = TempQ.SQL & "(([y4]-[y3])*[a]-[b]*([x4]-[x3])) AS Eq_43_c, "
TempQ.SQL = TempQ.SQL & "-1*([Eq_43_a]*[x3]+[Eq_43_b]*[y3]+[Eq_43_c]*[z3]) AS Eq_43_d, "
TempQ.SQL = TempQ.SQL & "(([z1]-[z4])*[b]-[c]*([y1]-[y4])) AS Eq_14_a, "
TempQ.SQL = TempQ.SQL & "-1*(([z1]-[z4])*[a]-[c]*([x1]-[x4])) AS Eq_14_b, "
TempQ.SQL = TempQ.SQL & "(([y1]-[y4])*[a]-[b]*([x1]-[x4])) AS Eq_14_c, "
TempQ.SQL = TempQ.SQL & "-1*([Eq_14_a]*[x4]+[Eq_14_b]*[y4]+[Eq_14_c]*[z4]) AS Eq_14_d "
TempQ.SQL = TempQ.SQL & "FROM [Temporary Vertices] AS [Temporary Vertices_4] "
TempQ.SQL = TempQ.SQL & "INNER JOIN ([Temporary Vertices] AS [Temporary Vertices_3] "
TempQ.SQL = TempQ.SQL & "INNER JOIN ([Temporary Vertices] AS [Temporary Vertices_1] "
TempQ.SQL = TempQ.SQL & "INNER JOIN ([Temporary Vertices] AS [Temporary Vertices_2] "
TempQ.SQL = TempQ.SQL & "INNER JOIN ([Temporary Patches] INNER JOIN [Temporary Equations] "
TempQ.SQL = TempQ.SQL & "ON [Temporary Patches].Patch_ID = [Temporary Equations].Patch_ID) "
TempQ.SQL = TempQ.SQL & "ON [Temporary Vertices_2].Vertex_ID = [Temporary Patches].Vertex2) "
TempQ.SQL = TempQ.SQL & "ON [Temporary Vertices_1].Vertex_ID = [Temporary Patches].Vertex1) "
TempQ.SQL = TempQ.SQL & "ON [Temporary Vertices_3].Vertex_ID = [Temporary Patches].Vertex3) "
TempQ.SQL = TempQ.SQL & "ON [Temporary Vertices_4].Vertex_ID = [Temporary Patches].Vertex4;"

TemporaryDB.QueryDefs.Append TempQ
'TempQ.Execute

End Sub
```

Sub Test2_VintoPOI

(POI_ID As Long)

This routine creates a query which solves the equations of the POI by substituting all the vertices in the database. Planes 1 to 4 refer to the sides of a prism which is perpendicular to the plane of interest.

Because of Access limitations, the results of this query are stored as a table.

```
Dim TempQ As New QueryDef

Set TempQ = TemporaryDB.CreateQueryDef()

TempQ.Name = "Interference - Vertices - Solutions for All"

On Error Resume Next
TemporaryDB.QueryDefs.Delete TempQ.Name
On Error GoTo 0

TempQ.SQL = "SELECT DISTINCTROW [Vertex List].Vertex_ID, "
TempQ.SQL = TempQ.SQL & "[Interference - Data - POI].Patch_ID AS [POI Patch ID], "
TempQ.SQL = TempQ.SQL & "([Interference - Data - POI]![a]*[Vertex List]![X]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![b]*[Vertex List]![Y]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![c]*[Vertex List]![Z]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![d]) AS POI, "
TempQ.SQL = TempQ.SQL & "([Interference - Data - POI]![Eq_21_a]*[Vertex List]![X]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_21_b]*[Vertex List]![Y]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_21_c]*[Vertex List]![Z]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_21_d]) AS Plane1, "
TempQ.SQL = TempQ.SQL & "([Interference - Data - POI]![Eq_32_a]*[Vertex List]![X]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_32_b]*[Vertex List]![Y]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_32_c]*[Vertex List]![Z]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_32_d]) AS Plane2, "
TempQ.SQL = TempQ.SQL & "([Interference - Data - POI]![Eq_43_a]*[Vertex List]![X]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_43_b]*[Vertex List]![Y]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_43_c]*[Vertex List]![Z]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_43_d]) AS Plane3, "
TempQ.SQL = TempQ.SQL & "([Interference - Data - POI]![Eq_14_a]*[Vertex List]![X]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_14_b]*[Vertex List]![Y]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_14_c]*[Vertex List]![Z]"
TempQ.SQL = TempQ.SQL & "+[Interference - Data - POI]![Eq_14_d]) AS Plane4, "
TempQ.SQL = TempQ.SQL & "0 AS Zone "
TempQ.SQL = TempQ.SQL & "INTO [Query - I - V - Solutions for All] "
TempQ.SQL = TempQ.SQL & "FROM [Vertex List], [Interference - Data - POI] "
TempQ.SQL = TempQ.SQL & "WHERE (([Interference - Data - POI].Patch_ID = "
TempQ.SQL = TempQ.SQL & Str$(POI_ID) & ")) "
TempQ.SQL = TempQ.SQL & "ORDER BY [Vertex List].Vertex_ID;"

TemporaryDB.QueryDefs.Append TempQ

TempQ.Execute

End Sub
```

Sub Test3_VertexZone

This routine determines which vertex lies in which zones from the data contained in the 'Query - I - V - Solutions for All' Table.

If [Plane1] >= 0 And [Plane2] >= 0 And [Plane3] >= 0 And [Plane4] >= 0 Then Zone 1

If [Plane1] < 0 And [Plane2] >= 0 And [Plane3] >= 0 And [Plane4] >= 0 Then Zone 2
If [Plane1] >= 0 And [Plane2] < 0 And [Plane3] >= 0 And [Plane4] >= 0 Then Zone 3
If [Plane1] >= 0 And [Plane2] >= 0 And [Plane3] < 0 And [Plane4] >= 0 Then Zone 4
If [Plane1] >= 0 And [Plane2] >= 0 And [Plane3] >= 0 And [Plane4] < 0 Then Zone 5

If [Plane1] < 0 And [Plane2] < 0 Then Zone 6
If [Plane2] < 0 And [Plane3] < 0 Then Zone 7
If [Plane3] < 0 And [Plane4] < 0 Then Zone 8
If [Plane4] < 0 And [Plane1] < 0 Then Zone 9

If [Plane1] < 0 And [Plane3] < 0 And [Plane4] < 0 Then Zone 10
If [Plane1] < 0 And [Plane2] < 0 And [Plane3] < 0 Then Zone 11
If [Plane2] < 0 And [Plane3] < 0 And [Plane4] < 0 Then Zone 12
If [Plane1] < 0 And [Plane3] < 0 And [Plane4] < 0 Then Zone 13

```
          \ 10 /
           \  /
            \/
            /\
     9     /  \     6
 \        | 2  |        / Plane3
  \  /---|----|---\  /
   \/    |    |    \/
13 /\  5 |  1 | 3  /\  11
  /  \---|----|---/  \
 /       | 4  |       \
          \  /           Plane1
     8     \/      7
           /\
          /  \
         / 12 \
   Plane4       Plane3
```

Zone 1 is the POI prism

```
Dim VISfATable As Recordset

Set VISfATable = TemporaryDB.OpenRecordset("Query - I - V - Solutions for All", DB_OPEN_TABLE)

VISfATable.MoveFirst
If VISfATable.EOF And VISfATable.BOF Then Return

Do While Not VISfATable.EOF
    VISfATable.Edit
    If ((VISfATable.Fields("Plane1") >= 0) And (VISfATable.Fields("Plane2") >= 0) And (VISfATable.Fields("Plane3") >= 0)
    And (VISfATable.Fields("Plane4") >= 0)) Then VISfATable.Fields("Zone") = 1
    ElseIf ((VISfATable.Fields("Plane1") < 0) And (VISfATable.Fields("Plane2") >= 0) And (VISfATable.Fields("Plane3") >=
    0) And (VISfATable.Fields("Plane4") >= 0)) Then VISfATable.Fields("Zone") = 2
    ElseIf ((VISfATable.Fields("Plane1") >= 0) And (VISfATable.Fields("Plane2") < 0) And (VISfATable.Fields("Plane3") >=
    0) And (VISfATable.Fields("Plane4") >= 0)) Then VISfATable.Fields("Zone") = 3
    ElseIf ((VISfATable.Fields("Plane1") >= 0) And (VISfATable.Fields("Plane2") >= 0) And (VISfATable.Fields("Plane3") <
    0) And (VISfATable.Fields("Plane4") >= 0)) Then VISfATable.Fields("Zone") = 4
```

```
ElseIf ((VISfATable.Fields("Plane1") >= 0) And (VISfATable.Fields("Plane2") >= 0) And (VISfATable.Fields("Plane3")
>= 0) And (VISfATable.Fields("Plane4") < 0)) Then VISfATable.Fields("Zone") = 5
ElseIf ((VISfATable.Fields("Plane1") < 0) And (VISfATable.Fields("Plane2") < 0)) Then VISfATable.Fields("Zone") = 6
ElseIf ((VISfATable.Fields("Plane2") < 0) And (VISfATable.Fields("Plane3") < 0)) Then VISfATable.Fields("Zone") = 7
ElseIf ((VISfATable.Fields("Plane3") < 0) And (VISfATable.Fields("Plane4") < 0)) Then VISfATable.Fields("Zone") = 8
ElseIf ((VISfATable.Fields("Plane4") < 0) And (VISfATable.Fields("Plane1") < 0)) Then VISfATable.Fields("Zone") = 9

ElseIf ((VISfATable.Fields("Plane1") < 0) And (VISfATable.Fields("Plane3") < 0) And (VISfATable.Fields("Plane4") < 0))
Then VISfATable.Fields("Zone") = 10
ElseIf ((VISfATable.Fields("Plane1") < 0) And (VISfATable.Fields("Plane2") < 0) And (VISfATable.Fields("Plane3") < 0))
Then VISfATable.Fields("Zone") = 11
ElseIf ((VISfATable.Fields("Plane2") < 0) And (VISfATable.Fields("Plane3") < 0) And (VISfATable.Fields("Plane4") < 0))
Then VISfATable.Fields("Zone") = 12
ElseIf ((VISfATable.Fields("Plane1") < 0) And (VISfATable.Fields("Plane3") < 0) And (VISfATable.Fields("Plane4") < 0))
Then VISfATable.Fields("Zone") = 13
End If
VISfATable.Update
VISfATable.MoveNext
Loop

End Sub
```

Sub TestZoneExamination

This routine determines which vertex lies in which zones from the data contained in the 'Query - I - V - Solutions for All' Table.

If [Plane1] >= 0 And [Plane2] >= 0 And [Plane3] >= 0 And [Plane4] >= 0 Then Zone 1

If [Plane1] < 0 And [Plane2] >= 0 And [Plane3] >= 0 And [Plane4] >= 0 Then Zone 2
If [Plane1] >= 0 And [Plane2] < 0 And [Plane3] >= 0 And [Plane4] >= 0 Then Zone 3
If [Plane1] >= 0 And [Plane2] >= 0 And [Plane3] < 0 And [Plane4] >= 0 Then Zone 4
If [Plane1] >= 0 And [Plane2] >= 0 And [Plane3] >= 0 And [Plane4] < 0 Then Zone 5

If [Plane1] < 0 And [Plane2] < 0 Then Zone 6
If [Plane2] < 0 And [Plane3] < 0 Then Zone 7
If [Plane3] < 0 And [Plane4] < 0 Then Zone 8
If [Plane4] < 0 And [Plane1] < 0 Then Zone 9

If [Plane1] < 0 And [Plane3] < 0 And [Plane4] < 0 Then Zone 10
If [Plane1] < 0 And [Plane2] < 0 And [Plane3] < 0 Then Zone 11
If [Plane2] < 0 And [Plane3] < 0 And [Plane4] < 0 Then Zone 12
If [Plane1] < 0 And [Plane3] < 0 And [Plane4] < 0 Then Zone 13

```
         \ 10 /
          \  /
           \/
           /\
      9   /  \   6
 \       | 2  |       / Plane3
  \  /---|----|---\  /
   \/    |    |    \/
13 /\  5 | 1  | 3  /\  11
  /  \---|----|---/  \
 /       | 4  |       \
          \  /          Plane1
      8    \/    7
           /\
          /  \
         / 12 \
   Plane4       Plane3
```

Zone 1 is the POI prism

If Zone1 = Zone2 = Zone3 = Zone4 = 1 then Patch is entirely enclosed by the POI prism.
If Zone1 = 1 And Zone2 = 1 And Zone3 = 1 Then

```
Dim currentPID As Long
Dim inVertCount As Integer
ReDim inVert(4) As Integer

VIPOIPTable.MoveFirst
If VIPOIPTable.EOF And VIPOIPTable.BOF Then Return

Do While Not VIPOIPTable.EOF
    GetVerticesInside (VIPOIPTable.Fields("Patch_ID")), inVert(), inVertCount
    Select Case inVertCount
    Case 1:                    OneVertIn
    Case 2:                    TwoVertIn inVert()
    Case 3:                    ThreeVertIn inVert()
    Case 4:                    FourVertIn - there is no need to add vertices in this case

    End Select
Loop

End Sub
```

Sub Test4_PatchesToConsider

(POI_ID As Long)

This routine creates a query which combines vertex information determined in the query 'Query - I - V - Solutions for All' for all patch and space ID values in the database.

```
Dim TempQ As New QueryDef

Set TempQ = TemporaryDB.CreateQueryDef()

TempQ.Name = "Interference - Patches - Patches to Consider"

On Error Resume Next
TemporaryDB.QueryDefs.Delete TempQ.Name
On Error GoTo 0

TempQ.SQL = "SELECT DISTINCTROW [Patch List].Space_ID, [Patch Corners].Patch_ID, "
TempQ.SQL = TempQ.SQL & "[Temporary Equations].Patch_ID AS [POI Patch_ID], "
TempQ.SQL = TempQ.SQL & "[Patch Corners].Vertex1, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_1].Zone AS V1_Zone, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_1].POI AS V1_POI, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_1].Plane1 AS V1_Plane1, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_1].Plane2 AS V1_Plane2, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_1].Plane3 AS V1_Plane3, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_1].Plane4 AS V1_Plane4, "
TempQ.SQL = TempQ.SQL & "[Patch Corners].Vertex2, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_2].Zone AS V2_Zone, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_2].POI AS V2_POI, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_2].Plane1 AS V2_Plane1, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_2].Plane2 AS V2_Plane2, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_2].Plane3 AS V2_Plane3, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_2].Plane4 AS V2_Plane4, "
TempQ.SQL = TempQ.SQL & "[Patch Corners].Vertex3, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_3].Zone AS V3_Zone, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_3].POI AS V3_POI, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_3].Plane1 AS V3_Plane1, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_3].Plane2 AS V3_Plane2, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_3].Plane3 AS V3_Plane3, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_3].Plane4 AS V3_Plane4, "
TempQ.SQL = TempQ.SQL & "[Patch Corners].Vertex4, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_4].Zone AS V4_Zone, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_4].POI AS V4_POI, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_4].Plane1 AS V4_Plane1, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_4].Plane2 AS V4_Plane2, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_4].Plane3 AS V4_Plane3, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_4].Plane4 AS V4_Plane4, "
TempQ.SQL = TempQ.SQL & "(([Patch Equation]![a]*[Temporary Equations]![a]"
TempQ.SQL = TempQ.SQL & "+[Patch Equation]![b]*[Temporary Equations]![b]"
TempQ.SQL = TempQ.SQL & "+[Patch Equation]![c]*[Temporary Equations]![c])"
TempQ.SQL = TempQ.SQL & "/(Sqr([Patch Equation]![a]*[Patch Equation]![a]"
TempQ.SQL = TempQ.SQL & "+[Patch Equation]![b]*[Patch Equation]![b]"
TempQ.SQL = TempQ.SQL & "+[Patch Equation]![c]*[Patch Equation]![c])"
TempQ.SQL = TempQ.SQL & "*Sqr([Temporary Equations]![a]*[Temporary Equations]![a]"
TempQ.SQL = TempQ.SQL & "+[Temporary Equations]![b]*[Temporary Equations]![b]"
TempQ.SQL = TempQ.SQL & "+[Temporary Equations]![c]*[Temporary Equations]![c]))) AS InOrOut "
TempQ.SQL = TempQ.SQL & "INTO [Query - I - P - Patches to Consider] "
TempQ.SQL = TempQ.SQL & "FROM [Temporary Equations], [Patch List] "
TempQ.SQL = TempQ.SQL & "INNER JOIN (([Query - I - V - Solutions for All] "
TempQ.SQL = TempQ.SQL & "AS [Query - I - V - Solutions for All_4] "
```

```
TempQ.SQL = TempQ.SQL & "INNER JOIN ([Query - I - V - Solutions for All] "
TempQ.SQL = TempQ.SQL & "AS [Query - I - V - Solutions for All_3] "
TempQ.SQL = TempQ.SQL & "INNER JOIN ([Query - I - V - Solutions for All] "
TempQ.SQL = TempQ.SQL & "AS [Query - I - V - Solutions for All_2] "
TempQ.SQL = TempQ.SQL & "INNER JOIN ([Query - I - V - Solutions for All] "
TempQ.SQL = TempQ.SQL & "AS [Query - I - V - Solutions for All_1] "
TempQ.SQL = TempQ.SQL & "INNER JOIN [Patch Corners] "
TempQ.SQL = TempQ.SQL & "ON [Query - I - V - Solutions for All_1].Vertex_ID "
TempQ.SQL = TempQ.SQL & "= [Patch Corners].Vertex1) "
TempQ.SQL = TempQ.SQL & "ON [Query - I - V - Solutions for All_2].Vertex_ID "
TempQ.SQL = TempQ.SQL & "= [Patch Corners].Vertex2) "
TempQ.SQL = TempQ.SQL & "ON [Query - I - V - Solutions for All_3].Vertex_ID "
TempQ.SQL = TempQ.SQL & "= [Patch Corners].Vertex3) "
TempQ.SQL = TempQ.SQL & "ON [Query - I - V - Solutions for All_4].Vertex_ID "
TempQ.SQL = TempQ.SQL & "= [Patch Corners].Vertex4) "
TempQ.SQL = TempQ.SQL & "INNER JOIN [Patch Equation] "
TempQ.SQL = TempQ.SQL & "ON [Patch Corners].Patch_ID = [Patch Equation].Patch_ID) "
TempQ.SQL = TempQ.SQL & "ON ([Patch List].Patch_ID = [Patch Corners].Patch_ID) "
TempQ.SQL = TempQ.SQL & "AND ([Patch List].Patch_ID = [Patch Equation].Patch_ID) "
TempQ.SQL = TempQ.SQL & "WHERE (([Temporary Equations].Patch_ID ="
TempQ.SQL = TempQ.SQL & Str$(POI_ID) & ")) "
TempQ.SQL = TempQ.SQL & "ORDER BY [Query - I - V - Solutions for All_1].POI, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_2].POI, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_3].POI, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_4].POI;"

TemporaryDB.QueryDefs.Append TempQ

TempQ.Execute

End Sub
```

Sub Test5_PatchestoExclude

This routine creates a query which examines the list of patches in the table 'Query - I - P - Patches to Consider' and determines patches which potentially interfere with the POI prism.

The query tests to see if a particular patch lies wholly outside a particular plane. If so, the query deletes this patch from 'Query - I - P - Patches to Consider'.

```
Dim TempQ As New QueryDef

Set TempQ = TemporaryDB.CreateQueryDef()

TempQ.Name = "Interference - Patches - Patches to Exclude"

On Error Resume Next
TemporaryDB.QueryDefs.Delete TempQ.Name
On Error GoTo 0

TempQ.SQL = "DELETE DISTINCTROW [Query - I - P - Patches to Consider].*, "
TempQ.SQL = TempQ.SQL & "[Patch Corners].Patch_ID, "
TempQ.SQL = TempQ.SQL & "[Patch Corners].Vertex1, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_1].Plane1 AS V1_Plane1, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_1].Plane2 AS V1_Plane2, "
```

```
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_1].Plane3 AS V1_Plane3, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_1].Plane4 AS V1_Plane4, "
TempQ.SQL = TempQ.SQL & "[Patch Corners].Vertex2, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_2].Plane1 AS V2_Plane1, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_2].Plane2 AS V2_Plane2, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_2].Plane3 AS V2_Plane3, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_2].Plane4 AS V2_Plane4, "
TempQ.SQL = TempQ.SQL & "[Patch Corners].Vertex3, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_3].Plane1 AS V3_Plane1, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_3].Plane2 AS V3_Plane2, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_3].Plane3 AS V3_Plane3, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_3].Plane4 AS V3_Plane4, "
TempQ.SQL = TempQ.SQL & "[Patch Corners].Vertex4, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_4].Plane1 AS V4_Plane1, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_4].Plane2 AS V4_Plane2, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_4].Plane3 AS V4_Plane3, "
TempQ.SQL = TempQ.SQL & "[Query - I - V - Solutions for All_4].Plane4 AS V4_Plane4 "
TempQ.SQL = TempQ.SQL & "FROM ([Query - I - V - Solutions for All] "
TempQ.SQL = TempQ.SQL & "AS [Query - I - V - Solutions for All_4] "
TempQ.SQL = TempQ.SQL & "INNER JOIN ([Query - I - V - Solutions for All] "
TempQ.SQL = TempQ.SQL & "AS [Query - I - V - Solutions for All_3] "
TempQ.SQL = TempQ.SQL & "INNER JOIN ([Query - I - V - Solutions for All] "
TempQ.SQL = TempQ.SQL & "AS [Query - I - V - Solutions for All_2] "
TempQ.SQL = TempQ.SQL & "INNER JOIN ([Query - I - V - Solutions for All] "
TempQ.SQL = TempQ.SQL & "AS [Query - I - V - Solutions for All_1] "
TempQ.SQL = TempQ.SQL & "INNER JOIN [Patch Corners] "
TempQ.SQL = TempQ.SQL & "ON [Query - I - V - Solutions for All_1].Vertex_ID "
TempQ.SQL = TempQ.SQL & "= [Patch Corners].Vertex1) "
TempQ.SQL = TempQ.SQL & "ON [Query - I - V - Solutions for All_2].Vertex_ID "
TempQ.SQL = TempQ.SQL & "= [Patch Corners].Vertex2) "
TempQ.SQL = TempQ.SQL & "ON [Query - I - V - Solutions for All_3].Vertex_ID "
TempQ.SQL = TempQ.SQL & "= [Patch Corners].Vertex3) "
TempQ.SQL = TempQ.SQL & "ON [Query - I - V - Solutions for All_4].Vertex_ID "
TempQ.SQL = TempQ.SQL & "= [Patch Corners].Vertex4) "
TempQ.SQL = TempQ.SQL & "INNER JOIN [Query - I - P - Patches to Consider] "
TempQ.SQL = TempQ.SQL & "ON [Patch Corners].Patch_ID "
TempQ.SQL = TempQ.SQL & "= [Query - I - P - Patches to Consider].Patch_ID "
TempQ.SQL = TempQ.SQL & "WHERE ((([Query - I - V - Solutions for All_1].Plane1<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_2].Plane1<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_3].Plane1<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_4].Plane1<0)) "
TempQ.SQL = TempQ.SQL & "OR (([Query - I - V - Solutions for All_1].Plane2<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_2].Plane2<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_3].Plane2<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_4].Plane2<0)) "
TempQ.SQL = TempQ.SQL & "OR (([Query - I - V - Solutions for All_1].Plane3<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_2].Plane3<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_3].Plane3<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_4].Plane3<0)) "
TempQ.SQL = TempQ.SQL & "OR (([Query - I - V - Solutions for All_1].Plane4<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_2].Plane4<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_3].Plane4<0) "
TempQ.SQL = TempQ.SQL & "AND ([Query - I - V - Solutions for All_4].Plane4<0));"

TemporaryDB.QueryDefs.Append TempQ

TempQ.Execute

End Sub
```

Module: ShipArrT Main Module

```
Option Compare Database
```

Use database order for string comparisons

```
Type Point3DDouble
    x As Double
    y As Double
    z As Double
End Type

Type Patch
    Vertex1 As Point3DDouble
    Vertex2 As Point3DDouble
    Vertex3 As Point3DDouble
    Vertex4 As Point3DDouble
End Type

Type Equation
    a As Double
    b As Double
    c As Double
    d As Double
End Type
```

Define an Equation data type
A, b, c, and d are equation coefficients
For an expression of the form
aX + bY + cZ + d = 0

```
Type Prism
    POI As Equation
    Plane1 As Equation
    Plane2 As Equation
    Plane3 As Equation
    Plane4 As Equation
End Type

Type DimensionSet
    min As Double
    pref As Double
    max As Double
End Type
```

Defines a range of a particular dimension

```
Global Const TempDBFName = "c:\workingf\tempfile\~temp"
```

Constant for the temporary file name. If the temporary database is used for other routines it is likely that this will be moved to the ShipArrTMain module as a global definition.

```
Global ActiveDB As Database
Global TemporaryDB As Database
```

Refers to the current database
Refers to a temporary database created and used by several modules

```
Global AdjTable As Recordset
Global CTable As Recordset
Global EqTable As Recordset
Global HETable As Recordset
Global VTable As Recordset
Global PTable As Recordset

Global ClassTable As Recordset
Global SLTable As Recordset
Global SOTable As Recordset
Global CLTable As Recordset
```

Refers to the table of adjacent patch references
Refers to the table of patch corner references
Refers to the table of patch equation values
Refers to the table of patch hidden edge flags
Refers to the table of patch vertex points
Refers to the table containing the list of patches for each space
Refers to the table containing class information
Refers to the table containing space ID numbers
Refers to the table containing the Ship Overall listings
Refers to the table containing the list of class specifications

Global POTable As Recordset	*Refers to the table containing the Placement order for each space*
Global ConstraintsTable As Recordset	*Refers to the table containing the Constraint Pointers for each space*
Global CAreaTable As Recordset	*Refers to the table containing the Area constraint for each space*
Global CLengthTable As Recordset	*Refers to the table containing the Length constraint for each space*
Global CWidthTable As Recordset	*Refers to the table containing the Width constraint for each space*
Global CHeightTable As Recordset	*Refers to the table containing the Height constraint for each space*
Global CVolumeTable As Recordset	*Refers to the table containing the Volume constraint for each space*
Global CShapeTable As Recordset	*Refers to the table containing the Shape constraint for each space*
Global VOTable As Recordset	*Refers to the table containing the Vertex ordering data*
Global TempVTable As Recordset	*Refers to the table containing the Vertex information for a newly created space*
Global TempPTable As Recordset	*Refers to the table containing the Patch information for a newly created space*
Global TempEqTable As Recordset	*Refers to the table containing the Equation information for the patches of a newly created space.*
Global TempAdjTable As Recordset	*Refers to the table containing the Patch Adjacentcy information for the patches of a newly created space.*

Sub PurgeWorkspace

It seems that if an error occurs in the midst of a Database transaction Access fails to automatically clear the transaction variables. This brief routine will do so for the occasions where such a problem exists. The error handlers will account for instances in which the .CommitTrans command returns an error (generally due to the lack of a .BeginTrans command).

```
On Error Resume Next
DBEngine.Workspaces(0).CommitTrans
On Error GoTo 0

End Sub
```

Sub PlaceFSMain

This routine places the first space into the layout domain.

```
AttachAdditionalTable (ActiveDB.Name), "Vertex List"
AttachAdditionalTable (ActiveDB.Name), "Patch List"
AttachAdditionalTable (ActiveDB.Name), "Patch Equation"
AttachAdditionalTable (ActiveDB.Name), "Patch Corners"
AttachAdditionalTable (ActiveDB.Name), "Vertex Order"
AttachAdditionalTable (ActiveDB.Name), "Placement Order"

CreateFSPatchTable
CreateFSVertexTable
CreateFSEquationTable
CreateFSAdjacentcyTable

OpenFSTables

POTable.MoveFirst
'Do Until POTable.EOF                                    Repeat until the name is not found
'    TemporaryDB.BeginTrans
     LocateNewSpace
     CreateNewSpace (POTable.Fields("Space_ID"))         Calls a routine to get all the table contents
     KillVertexRepeats "TEMPORARY", (TempVTable.Name), (TempPTable.Name)
     TempEquations
     TestMain


'    TemporaryDB.Rollback
'    TemporaryDB.CommitTrans
'    POTable.MoveNext
'Loop

CloseFSTables

'TemporaryDB.Rollback
'TemporaryDB.CommitTrans

End Sub
```

Sub PrepareTemporaryDB

Ordinarily, one would create the temporary database at the beginning of this module and delete it at the end. Unfortunately, I have not been able to figure out how to get Access to relinquish its locks on the TemporaryDB at the end of the DXFExportMain routine. It therefore leaves an empty TemporaryDB on disk. The file is empty because of the use of the transactions commands in the loop.

The contents of this section ensure that it is purged prior to any new operations on the TemporaryDB.

```
On Error Resume Next
Kill TempDBFName & ".MDB"
Kill TempDBFName & ".LDB"
On Error GoTo 0

Set TemporaryDB = DBEngine.Workspaces(0).CreateDatabase((TempDBFName & ".MDB"), DB_LANG_GENERAL)
Set TemporaryDB = DBEngine.Workspaces(0).OpenDatabase((TempDBFName & ".MDB"), True)

End Sub
```

Function SeekLastRecord

(db As String, tableName As String) As Long

This routine is a generic routine accepting database object names from either the temporary or active databases. It tests to determine if the table holds any entries, and if so, advances the table's position pointer to the last entry where it then stores the ID value for the last record.

```
Dim tempTable As Recordset

If db = "ACTIVE" Then                                                    Test to determine which database the tableName
                                                                         should be associated
    Set tempTable = ActiveDB.OpenRecordset(tableName, DB_OPEN_TABLE)     Assign the tempTable variable
ElseIf db = "TEMPORARY" Then
    Set tempTable = TemporaryDB.OpenRecordset(tableName, DB_OPEN_TABLE)
End If

If tempTable.BOF And tempTable.EOF Then                                  Test for a table without any entries
    SeekLastRecord = 0      and return a 0
Else
    tempTable.Index = "PrimaryKey"                                       Set the index to the key containing the ID values
    tempTable.MoveLast                                                   And move to the last entry in the table
    SeekLastRecord = tempTable.Fields(0)                                 Store the ID value of the last entry
End If

tempTable.Close                                                          Purge the tempTable variable.

End Function
```

Sub ShipArrTMain

(routineName As String)

This is the primary routine in this Database - all directions from the forms are channelled through this routine. The reason is that this allows the use of several global variables throughout the model.

```
PurgeWorkspace

Set ActiveDB = DBEngine.Workspaces(0).Databases(0)                     Assign the database variable

If routineName = "DXFImport" Then                                      Test the routine Name
    DXFImportMain                                                      Import the DXF file
    KillVertexRepeats "ACTIVE", "Vertex List", "Patch Corners"         Purge unnecessary vertex data
    Renumber "ACTIVE", "Vertex List"                                   Renumber the vertex list
    Equations                                                          Generate all of the Equations table entries
    Adjacentcies                                                       Generate all of the Adjacent Patch table entries
    HiddenEdges                                                        Generate all of the Hidden Edge table entries
ElseIf routineName = "DXFExport" Then
    DXFExportMain                                                      Export the object dataset as a DXF file
ElseIf routineName = "ConstraintCreationMain" Then
    ConstraintCreationMain
ElseIf routineName = "SpaceCreationMain" Then
    ConstraintCreationMain
    SpaceCreationMain
ElseIf routineName = "PlaceFSMain" Then
    ConstraintCreationMain
    SpaceCreationMain
    PlaceFSMain
ElseIf routineName = "PlaceSpacesMain" Then
    ConstraintCreationMain
    SpaceCreationMain
    PlaceFSMain
'   PlaceSpacesMain

'Else

End If

End Sub
```

Module: Space Creation Module

```
Option Compare Database                    ' Use database order for string comparisons

Dim Centroid_X As Double
Dim Centroid_Y As Double
Dim Centroid_Z As Double
```

Sub Create_Deck

(eq As Equation)

This module can be significantly improved. Currently I have assumed a simple cutting plane for a deck. The plane takes the form $aX + bY + cZ + d = 0$ where $a = 0$, $b = 0$, $c = 1$, and $d = -1$.

```
eq.a = 0
eq.b = 0
eq.c = 1
eq.d = -1

End Sub
```

Sub CreateCorner

(cornerNum As Integer)

```
Dim corner As String

corner = "Vertex" & Right$(Str$(cornerNum), 1)

TempVTable.AddNew

TempVTable.Fields("Vertex_ID") = SeekLastRecord("TEMPORARY", "Temporary Vertices") + 1
TempVTable.Fields("X") = Centroid_X + RelativeToCentroid(corner & " - X") * PrefTable.Fields("Length") / 2
TempVTable.Fields("Y") = Centroid_Y + RelativeToCentroid(corner & " - Y") * PrefTable.Fields("Width") / 2
TempVTable.Fields("Z") = Centroid_Z + RelativeToCentroid(corner & " - Z") * PrefTable.Fields("Height") / 2
TempPTable.Fields(corner) = TempVTable.Fields("Vertex_ID")

TempVTable.Update

End Sub
```

Sub CreateNewSpace

(TempPID As Long)

Vertices are added in clockwise direction as viewed from inside the space.

```
PrefTable.Index = "Space_ID"
PrefTable.Seek "=", TempPID

VOTable.MoveFirst
Do Until VOTable.EOF                                        Repeat until all six faces have been created
    TempPTable.AddNew

    TempPTable.Fields("Patch_ID") = SeekLastRecord("TEMPORARY", "Temporary Patches") + 1
    TempPTable.Fields("Face_Name") = VOTable.Fields("Face_Name")

    CreateCorner 1
    CreateCorner 2
    CreateCorner 3
    CreateCorner 4

    TempPTable.Update

    VOTable.MoveNext
Loop

End Sub
```

Sub LocateNewSpace

This routine can be fleshed out to accomodate random placements, etc. It may be worthwhile filling this out in phases — fix Z and let X and Y go random.

For the moment I will give a specific start point

```
Centroid_X = (30 - (-1.27401)) / 2        Amid ship
Centroid_Y = 0                            Amid ship
Centroid_Z = 1 + 1.5                      Assumed base of deck lies at Z = 1 and deck height
                                          is 3m

End Sub
```

Function RelativeToCentroid

```
(vertexName As String) As Double

If VOTable.Fields(vertexName) = True Then          Points to positive side of Centroid
    RelativeToCentroid = 1
Else                                               Points to negative side of Centroid
    RelativeToCentroid = (-1)
End If

End Function
```

Sub TempEquations

This routine generates the a, b, c and d equation parameters for each patch and stores them in the Temporary Patch Equations table.

The equation takes the form of: $aX + bY + cZ + d = 0$

```
Dim i As Integer                                   An indexing variable
Dim a As Double                                    Equation parameters
Dim b As Double
Dim c As Double
Dim d As Double
ReDim X(3) As Double                               Arrays of coordinate information used to derive a
                                                   patch
ReDim Y(3) As Double
ReDim Z(3) As Double

TempVTable.Index = "PrimaryKey"                    Set the indices of the tables being searched to their ID
                                                   values
TempPTable.Index = "PrimaryKey"
TempEqTable.Index = "PrimaryKey"

TempPTable.MoveFirst

Do Until TempPTable.EOF                            Examine the entire Patch List table

    For i = 1 To 3                                 Loop through the first three corners
        TempVTable.Seek "=", (TempPTable.Fields(i))    Seek the vertex coordinate data for the particular
                                                   corner pointer from the VTable
        X(i) = TempVTable.Fields("X")
        Y(i) = TempVTable.Fields("Y")
        Z(i) = TempVTable.Fields("Z")
    Next i

    a = Y(1) * (Z(2) - Z(3))                       Generate the equation variables
    a = a - Z(1) * (Y(2) - Y(3))
    a = a + (Y(2) * Z(3) - Y(3) * Z(2))

    b = X(1) * (Z(2) - Z(3))
    b = b - Z(1) * (X(2) - X(3))
    b = b + (X(2) * Z(3) - X(3) * Z(2))
    b = b * (-1)
```

```
        c = X(1) * (Y(2) - Y(3))
        c = c - Y(1) * (X(2) - X(3))
        c = c + (X(2) * Y(3) - Y(2) * X(3))

'       d = x(1) * (y(2) * z(3) - z(2) * y(3))
'       d = d - y(1) * (x(2) * z(3) - x(3) * z(2))
'       d = d + z(1) * (x(2) * y(3) - x(3) * y(2))
'       d = d * (-1)

        a = a / (a ^ 2 + b ^ 2 + c ^ 2) ^ .5                    Make normal values 'unit normals'
        b = b / (a ^ 2 + b ^ 2 + c ^ 2) ^ .5
        c = c / (a ^ 2 + b ^ 2 + c ^ 2) ^ .5

        d = -1 * (a * X(1) + b * Y(1) + c * Z(1))

        TempEqTable.Seek "=", TempPTable.Fields("Patch_ID")
        If TempEqTable.NoMatch Then
            TempEqTable.AddNew                                  Update the EqTable with the new entry
        Else
            TempEqTable.Edit
        End If

        TempEqTable.Fields("Patch_ID") = TempPTable.Fields("Patch_ID")

        TempEqTable.Fields("a") = a
        TempEqTable.Fields("b") = b
        TempEqTable.Fields("c") = c
        TempEqTable.Fields("d") = d
        TempEqTable.Update
        TempPTable.MoveNext
    Loop

    End Sub
```

Module: Space Placement Tables

```
Option Compare Database                                        ' Use database order for string comparisons
```

Sub AttachAdditionalTable

```
(fileName As String, tableName As String)

Dim TempTableDef As TableDef


Set TempTableDef = TemporaryDB.CreateTableDef(tableName)
TempTableDef.Connect = ";DATABASE=" & fileName
TempTableDef.SourceTableName = tableName
TemporaryDB.TableDefs.Append TempTableDef                      ' Attach table.
ConnectSource = True

End Sub
```

Sub OpenFSTables

```
Set POTable = TemporaryDB.OpenRecordset("Placement Order", DB_OPEN_DYNASET)     ' Assign table variables for the tables
                                                                                ' in the ActiveDB
Set VOTable = TemporaryDB.OpenRecordset("Vertex Order", DB_OPEN_DYNASET)

Set MinTable = TemporaryDB.OpenRecordset("Minimum", DB_OPEN_TABLE)              ' Assign table variables for the tables
                                                                                ' in the TemporaryDB
Set PrefTable = TemporaryDB.OpenRecordset("Preferred", DB_OPEN_TABLE)
Set MaxTable = TemporaryDB.OpenRecordset("Maximum", DB_OPEN_TABLE)
Set ShapeTable = TemporaryDB.OpenRecordset("Shape", DB_OPEN_TABLE)
Set TempVTable = TemporaryDB.OpenRecordset("Temporary Vertices", DB_OPEN_TABLE)
Set TempPTable = TemporaryDB.OpenRecordset("Temporary Patches", DB_OPEN_TABLE)
Set TempEqTable = TemporaryDB.OpenRecordset("Temporary Equations", DB_OPEN_TABLE)
Set TempAdjTable = TemporaryDB.OpenRecordset("Temporary Adjacentcies", DB_OPEN_TABLE)

Set VTable = TemporaryDB.OpenRecordset("Vertex List", DB_OPEN_DYNASET)
Set PTable = TemporaryDB.OpenRecordset("Patch List", DB_OPEN_DYNASET)
Set EqTable = TemporaryDB.OpenRecordset("Patch Equation", DB_OPEN_DYNASET)
Set CTable = TemporaryDB.OpenRecordset("Patch Corners", DB_OPEN_DYNASET)

End Sub
```

Sub CloseFSTables

```
POTable.Close
VOTable.Close

MinTable.Close
PrefTable.Close
MaxTable.Close
ShapeTable.Close
TempVTable.Close
TempPTable.Close
TempEqTable.Close
TempAdjTable.Close

End Sub
```

Sub CreateFSAdjacentcyTable

```
Dim TempTable As Recordset

Set AdjTable = ActiveDB.OpenRecordset("Patch Adjacency", DB_OPEN_TABLE)

DoCmd CopyObject TemporaryDB.Name, "Temporary Adjacentcies", A_TABLE, AdjTable.Name
AdjTable.Close

Set TempAdjTable = TemporaryDB.OpenRecordset("Temporary Adjacentcies", DB_OPEN_TABLE)

TempAdjTable.MoveFirst

If ((TempAdjTable.EOF = True) And (TempAdjTable.BOF = True)) Then

Else
    Do Until TempAdjTable.EOF
        TempAdjTable.Delete
        TempAdjTable.MoveNext
    Loop
End If

TempAdjTable.Close

End Sub
```

Set the tables used in this routine

Do nothing

Sub CreateFSEquationTable

This routine creates temporary tables containing patch equation variables for the current space under construction.

The routine is fairly self explanatory. It essentially creates each of the elements of the table (Fields and Indices) and appends these to the newTableDef definition. In turn, this definition is appended to the TemporaryDB.TableDefs collection, thus creating the tables.

```
ReDim f(5) As New Field
ReDim i(6) As New Index
Dim newTblDef As New TableDef

newTblDef.Name = "Temporary Equations"          ' Name the new table

f(1).Name = "Patch_ID"                          ' Create fields
f(1).Type = DB_LONG

f(2).Name = "a"
f(2).Type = DB_DOUBLE

f(3).Name = "b"
f(3).Type = DB_DOUBLE

f(4).Name = "c"
f(4).Type = DB_DOUBLE

f(5).Name = "d"
f(5).Type = DB_DOUBLE

i(1).Name = "PrimaryKey"                        ' Create indices
i(1).Fields = "Patch_ID"
i(1).primary = True
newTblDef.Indexes.Append I(1)                   ' Add it to the collection

For j = 1 To 5
    newTblDef.Fields.Append f(j)                ' Add it to the collection

    i(j + 1).Name = f(j).Name
    i(j + 1).Fields = f(j).Name
    i(j + 1).primary = False
    newTblDef.Indexes.Append i(j + 1)           ' Add it to the collection
Next j

TemporaryDB.TableDefs.Append newTblDef          ' Now append the new Table object to the TableDefs collection.

End Sub
```

Sub CreateFSPatchTable

This routine creates temporary tables containing vertex pointers patches and vertex pointers for a particular space.

The routine is fairly self explanatory. It essentially creates each of the elements of the table (Fields and Indices) and appends these to the newTableDef definition. In turn, this definition is appended to the TemporaryDB.TableDefs collection, thus creating the tables.

```
ReDim f(6) As New Field
ReDim i(7) As New Index
Dim newTblDef As New TableDef


newTblDef.Name = "Temporary Patches"                          Name the new table

f(1).Name = "Patch_ID"        Create fields
f(1).Type = DB_LONG

f(2).Name = "Vertex1"
f(2).Type = DB_LONG

f(3).Name = "Vertex2"
f(3).Type = DB_LONG

f(4).Name = "Vertex3"
f(4).Type = DB_LONG

f(5).Name = "Vertex4"
f(5).Type = DB_LONG

f(6).Name = "Face_Name"
f(6).Type = DB_TEXT


i(1).Name = "PrimaryKey"                                      Create indices
i(1).Fields = "Patch_ID"
i(1).primary = True
newTblDef.Indexes.Append I(1)                                 Add it to the collection

For j = 1 To 6
    newTblDef.Fields.Append f(j)                              Add it to the collection

    i(j + 1).Name = f(j).Name
    i(j + 1).Fields = f(j).Name
    i(j + 1).primary = False
    newTblDef.Indexes.Append i(j + 1)                         Add it to the collection
Next j

TemporaryDB.TableDefs.Append newTblDef                        Now append the new Table object to the TableDefs
                                                              collection.


End Sub
```

Sub CreateFSVertexTable

This routine creates temporary tables containing vertex ID values and vertex coordinates.

The routine is fairly self explanatory. It essentially creates each of the elements of the table (Fields and Indices) and appends these to the newTableDef definition. In turn, this definition is appended to the TemporaryDB.TableDefs collection, thus creating the tables.

```
ReDim f(4) As New Field
ReDim i(6) As New Index
Dim newTblDef As New TableDef


newTblDef.Name = "Temporary Vertices"                    Name the new table

f(1).Name = "Vertex_ID"                                  Create fields
f(1).Type = DB_LONG

f(2).Name = "X"
f(2).Type = DB_DOUBLE

f(3).Name = "Y"
f(3).Type = DB_DOUBLE

f(4).Name = "Z"
f(4).Type = DB_DOUBLE


i(1).Name = "PrimaryKey"                                 Create indices
i(1).Fields = "Vertex_ID"
i(1).primary = True
newTblDef.Indexes.Append I(1)                            Add it to the collection

For j = 1 To 4
    newTblDef.Fields.Append f(j)                         Add it to the collection

    i(j + 1).Name = f(j).Name
    i(j + 1).Fields = f(j).Name
    i(j + 1).primary = False
    newTblDef.Indexes.Append i(j + 1)                    Add it to the collection
Next j

i(6).Name = "XYZ"                                        Create indices
i(6).Fields = "+X;+Y;+Z"
i(6).primary = False
newTblDef.Indexes.Append I(6)                            Add it to the collection

TemporaryDB.TableDefs.Append newTblDef                   Now append the new Table object to the TableDefs
                                                         collection.

End Sub
```

Module: Space Table Routines

```
Option Compare Database                Use database order for string comparisons

Global MinTable As Recordset           Refers to the table containing the minimum dimensions
                                       for each space
Global PrefTable As Recordset          Refers to the table containing the preferred dimensions
                                       for each space
Global MaxTable As Recordset           Refers to the table containing the maximum
                                       dimensions for each space
Global ShapeTable As Recordset         Refers to the table containing the shape rules for each
                                       space
```

Sub CloseCreationTables

```
(dbName As String)

CLTable.Close
SLTable.Close
ConstraintsTable.Close

MinTable.Close
PrefTable.Close
MaxTable.Close
ShapeTable.Close

End Sub
```

Sub SetCreationTables

```
Set TemporaryDB = DBEngine.Workspaces(0).OpenDatabase((TempDBFName & ".MDB"), True)

Set SOTable = ActiveDB.OpenRecordset("Ship Overall", DB_OPEN_TABLE)
Set SLTable = ActiveDB.OpenRecordset("Space List", DB_OPEN_TABLE)
Set CLTable = ActiveDB.OpenRecordset("Class List", DB_OPEN_TABLE)

Set SLTable = ActiveDB.OpenRecordset("Space List", DB_OPEN_TABLE)       Assign table variables for the tables in the
                                                                        ActiveDB
Set CLTable = ActiveDB.OpenRecordset("Class List", DB_OPEN_TABLE)
Set ConstraintsTable = ActiveDB.OpenRecordset("Constraints", DB_OPEN_TABLE)

Set MinTable = TemporaryDB.OpenRecordset("Minimum", DB_OPEN_TABLE)      Assign table variables for the tables in the
                                                                        TemporaryDB
Set PrefTable = TemporaryDB.OpenRecordset("Preferred", DB_OPEN_TABLE)
Set MaxTable = TemporaryDB.OpenRecordset("Maximum", DB_OPEN_TABLE)
Set ShapeTable = TemporaryDB.OpenRecordset("Shape", DB_OPEN_TABLE)

End Sub
```

Sub SpaceCreationMain

```
Dim i As Integer                                    A counter variable
Dim lastRecord As Long                              A position marker

SetCreationTables

CLTable.Index = "Class_ID"                          Set the index of the CLTable to Class_ID
SLTable.Index = "Class_ID"                          Set the index of the SLTable to Space_Name
SOTable.Index = "Class_ID"

SOTable.MoveFirst
Do Until SOTable.EOF                                Repeat until the name is not found
    CLTable.Seek "=", (SOTable.Fields("Class_ID"))  Seek a space name
    SLTable.Seek "=", (SOTable.Fields("Class_ID"))

    If SLTable.NoMatch Then                         Check for repeated name
        For i = 1 To SOTable.Fields("Quantity")     Loop through the quantity of each space
            lastRecord = SeekLastRecord("ACTIVE", (SLTable.Name))   Get the last record number
            SLTable.AddNew                          Add the new entry
            SLTable.Fields("Space_ID") = lastRecord + 1   Increment the Space_ID for the new record
            SLTable.Fields("Space_Name") = (CLTable.Fields("Class_Name") & Str$(i))   Name the new space
            SLTable.Fields("Class_ID") = CLTable.Fields("Class_ID")   Number the new class
            SLTable.Update                          Complete the entry
        Next i
    Else
                                                    Space has already been defined
    End If
    SOTable.MoveNext
Loop

End Sub
```

Module: Utility Subroutines

```
Option Compare Database                                          Use database order for string comparisons

'Const SurfCorners = 4
'Type Surf
'    cPt(SurfCorners) As Point3DDouble
'End Type
'Dim Surface() As Surf
'Dim SurfCount As Long
'Dim TriSurfCount As Long
'Dim SurfFNum As Integer

'Type SurfSortPos
'    cPos(4) As Long
'End Type
'Dim SurfSort() As SurfSortPos
'Dim SurfSortFNum As Integer

'Dim VertCount As Long
'Dim VertFNum As Integer
'Dim Vertex() As Point3DDouble
```

Function MaxPoint

```
(pt0 As Point3DDouble, pt1 As Point3DDouble) As Integer          Boolean

A structure to determine the relative positions of the two 3d Points provided.

Dim equalx As Integer                                            Boolean
Dim equaly As Integer                                            Boolean

If pt0.x = pt1.x Then equalx = True
If pt0.y = pt1.y Then equaly = True

If pt0.x > pt1.x Then
    MaxPoint = True
ElseIf equalx And pt0.y > pt1.y Then
    MaxPoint = True
ElseIf equalx And equaly And pt0.z > pt1.z Then
    MaxPoint = True
Else
    MaxPoint = False
End If

End Function
```

Sub CopyPts

(pt0 As Point3DDouble, pt1 As Point3DDouble)

```
pt1.x = pt0.x
pt1.y = pt0.y
pt1.z = pt0.z

End Sub
```

Function SurfacePos

(pt0 As Point3DDouble, corner As Integer) As Long

Binary search for the first corner of a 3d surface from one of the sorted surface arrays.

Returns the position in the Surface file (and hence the Vertex number in the DXF file) of the given 3d Point.

```
Dim lo As Long
Dim hi As Long
Dim indx As Long
Dim found As Integer                                    Boolean
'Dim surfPos As SurfSortPos
'Dim surf1 As Surf

Let lo = 1
Let hi = SurfCount
Let indx = Int((hi - lo) / 2)
Let found = False

Do While found = False And lo <= hi

'     If SurfCount <= ArrayMax Then
'          surfPos = SurfSort(indx)
'          CopySurfs Surface(surfPos.cPos(corner)), surf1
'     Else
'          Get #SurfSortFNum, indx, surfPos
'          Get #SurfFNum, surfPos.cPos(corner), surf1
'     End If

'     If MaxPoint(pt0, surf1.cPt(corner)) Then
'          lo = indx + 1
'     ElseIf EqualPts(pt0, surf1.cPt(corner)) Then
'          found = True
'          SurfacePos = surfPos.cPos(corner)
'          Exit Do
'     Else
'          hi = indx - 1
'     End If
'
'     indx = Int((lo + hi) / 2)
'
Loop

End Function
```

Function EqualPts

(pt0 As Point3DDouble, pt1 As Point3DDouble) As Integer

Compares the two 3d Points provided for equality.

```
If (pt0.x = pt1.x) And (pt0.y = pt1.y) And (pt0.z = pt1.z) Then
      EqualPts = True
Else
      EqualPts = False
End If

End Function
```

Sub SwapPts

(pt0 As Point3DDouble, pt1 As Point3DDouble)

This routine simply exchanges two 3D points.

```
Dim temp As Point3DDouble

CopyPts pt0, temp
CopyPts pt1, pt0
CopyPts temp, pt1

End Sub
```

Sub SwapValues

(value1 As Long, value2 As Long)

This routine simple exchanges two variable values.

```
junk = value1
value1 = value2
value2 = junk

End Sub
```

Finding Potential Vertices — Pseudocode Corresponding to Section 4.3

Pseudocode for the derivation of the 24 potential vertices for the creation of new Patches. Note that the bold portions of code refer to simple algebraic subroutines.

```
k = 0
For I = 1 to 4
    k = k + 1
    tempVertex( k ) = Patch_Vertex( I )
    For j = 1 to 4
        k = k + 1
        tempVertex( k ) = Intersection( Patch_Equation, Patch_Side_Equation( I ), POI_Side_Equation( j ) )
    Next j
    If I = 4 then
        m = 1
    else
        m = I + 1
    endif
    tempVertex( k ) = Intersection( Patch_Equation, POI_Side_Equation( I ), POI_Side_Equation( m ) )
Next I
```

Verification of Vertices — Pseudocode Corresponding to Section 4.4

Pseudocode for the substitution of the 24 vertices into the retion of Validity defined by the shared region of the POI Prism and the Patch Prism. Note again that the bold portions of code refer to simple algebraic subroutines which are not shown.

```
k = 0
For I = 1 to 24
    whollyContainedFlag = True
    For j = 1 to 4
        If not Contained( tempVertex( I ), PatchPlane( j ) ) then whollyContainedFlag = False
        If not Contained( tempVertex( I ), POIPlane( j ) ) then whollyContainedFlag = False
        If whollyContainedFlag = False then j = 4
    Next j
    If whollyContainedFlag = True then
        k = k + 1
        patchVertex( I ) = tempVertex( I )
    Endif
Next I
```

Counting the Vertices — Pseudocode Corresponding to Section 4.5

Pseudocode which counts the Vertices found in the substitution step described in the previous section.

```
counter = 0
Loop
    counter = counter + 1
Until patchVertex( k ) = null
counter = counter - 1
```

Creating Patches — Pseudocode Corresponding to Section 4.8

Pseudocode by which new patches are created on the Patch Plane.

```
j = 0
For I = 1 to VertexCount
    j = j + 1
    NewPatch.Corner( j ) = patchVertex( I )
    If j = 4 then
        WriteNewPatch( NewPatch )
        NewPatch.Corner( 2 ) = patchVertex( I )
        NewPatch.Corner( 3 ) = Null
        NewPatch.Corner( 4 ) = Null
        j = 2
    Endif
Next I

If NewPatch.Corner( 4 ) = Null then
    NewPatch.Corner( 4 ) = NewPatch.Corner( 3 )
Endif
```

Determining the Vertices — Pseudocode Corresponding to Section 5.2

Pseudocode which performs the substitution of vertex coordinates into the Prism Plane equations for each of the four prism sides.

```
For i = 1 to 4
    PrismPlane( i )
    k = 1
    For j = 1 to newVertexCount
        solution = PrismPlane( i ).a * Vertex( j ).x + PrismPlane( i ).b * Vertex( j ).y
        solution = solution + PrismPlane( i ).c * Vertex( j ).z + PrismPlane( i ).d
        If solution = 0 then
            VertexList( i, k ) = Vertex( j )
            k = k + 1
        Endif
    Next j
Next I
```

Creating an Ordered Vertex List — Pseudocode Corresponding to Section 5.3

Pseudocode for the main Vertex Ordering routine.

```
a = 1
SortedVertexList( Side#, a ) = EndVertex

FindFirstPatch
Loop
     RemoveCurrentVertex
     FindNextVertex
     RemoveCurrentPatch
     FindNextPatch
Until VertexList.Count = 0

a = SortedVertexList( Side# ).Count
SortedVertexList( Side#, a + 1 ) = OtherEndVertex
```

Sub FindFirstPatch — Pseudocode

```
For i = 1 to NewPatchCount
     For j = 1 to 4
          If NewPatch( i ).Vertex( j ) = EndVertex then
               EndPatch = False
               For k = 1 to 4
                    If j <> k then
                         If NewPatch( i ).Vertex( k ) = OtherEndVertex then
                              EndPatch = True
                              k = 4
                         Endif
                    Endif
               Next k
               If EndPatch = False then
                    CurrentPatch = i
                    CurrentVertex = j
               Endif
          Endif
     Next j
Next i

End Sub
```

Sub FindNextVertex — Pseudocode

```
For m = 1 to VertexList.Count
    For n = 1 to 4
        If n <> CurrentVertex then
            If VertexList( Side#, m ) <> SortedVertexList( Side#, a ) then
                If NewPatch( i ).Vertex( n ) = VertexList( Side#, m )
                    a = a + 1
                    SortedVertexList( Side#, a ) = VertexList( Side#, m )
                    j = 4
                Endif
            Endif
        Endif
    Next n
Next m

End Sub
```

Sub RemoveCurrentPatch — Pseudocode

```
SortedPatchCount = SortedPatchCount + 1
SortedPatchList( SortedPatchCount ) = NewPatchList( CurrentPatch )

NewPatchCount = NewPatchCount - 1
j = 1
For i = 1 to NewPatchList.Count
    If i = CurrentPatch then
        j = j + 1
    Endif
    NewPatchList( i ) = NewPatchList( j )
Next i

EndSub
```

Sub RemoveCurrentVertex — Pseudocode

```
SortedVertexList.Count = SortedVertexList.Count + 1
SortedVertexList( SortedVertexCount ) = VertexList( CurrentVertex )

VertexList.Count = VertexList.Count - 1
j = 1
For i = 1 to VertexList.Count
    If i = CurrentVertex then
        j = j + 1
    Endif
    VertexList( i ) = VertexList( j )
Next i

EndSub
```

Sub FindNextPatch — Pseudocode

```
NewPatch
For i = 1 to NewPatchCount
    For j = 1 to 4
        If i <> CurrentPatch then
            If NewPatch( i ).Vertex( j ) = SortedVertexList( Side#, a ) then
                CurrentPatch = i
                i = NewPatchCount
                CurrentVertex = j
                j = 4
            Endif
        Endif
    Next j
Next i

EndSub
```

Calculating Angles — Pseudocode Corresponding to Section 5.4

Pseudocode which determines the angles formed by the vertices of this plane.

```
For i = 1 to VertexList( side# ).Count
    If i = 1 then
        Vector.x = VertexList( side#, VertexList.Count ).x - VertexList( side#, i ).x
        Vector.y = VertexList( side#, VertexList.Count ).y - VertexList( side#, i ).y
        Vector.z = VertexList( side#, VertexList.Count ).z - VertexList( side#, i ).z
        FindAngle( VertexList( side#, VertexList.Count ), VertexList( side#, i ), VertexList( side#, i+1 ), Angle )
    Else
        Vector.x = VertexList( side#, i+1 ).x - VertexList( side#, i ).x
        Vector.y = VertexList( side#, i+1 ).y - VertexList( side#, i ).y
        Vector.z = VertexList( side#, i+1 ).z - VertexList( side#, i ).z
        FindAngle( VertexList( side#, i-1 ), VertexList( side#, i ), VertexList( side#, i+1 ), Angle )
    Endif

    If i = VertexList.( side# )Count then
        vertexToCheck = VertexList( side#, 1 )
    Else
        vertexToCheck = VertexList( side#, i )
    Endif

    FindSide( Normal( side# ), Vector, VertexList( side#, i ), vertexToCheck, sideSolution )

    If sideSolution < 0 then
        Angle = 360 - Angle
    Endif
Next i
```

Sub FindSide — Pseudocode

```
( vector1, vector2, ptOnPlane, ptToCheck, sideSolution )

A = vector1.y * vector2.z - vector2.y * vector1.z
B = vector1.x * vector2.z - vector2.x * vector1.z
C = vector1.x * vector2.y - vector2.x * vector1.y
D = -1 * ( A * ptOnPlane.x + B * ptOnPlane.y + C * ptOnPlane.z )

sideSolution = A * ptToCheck.x + B * ptToCheck.y + C * ptToCheck.z + D

EndSub
```

Sub FindAngle — Pseudocode

```
( pt1, pt2, pt3, theta )

a1 = pt1.x - pt2.x
b1 = pt1.y - pt2.y
c1 = pt1.z - pt2.z

a2 = pt3.x - pt2.x
b2 = pt3.y - pt2.y
c2 = pt3.z - pt2.z

theta = ( a1 * a2 + b1 * b2 + c1 * c3 )
theta = theta / ( SQRT( a1^2 + b1^2 +c1^2 ) * SQRT( a2^2 + b2^2 + c2^2 ) )
theta = ARCCOS( theta )
theta = theta / Pi * 180

EndSub
```

Creating Patches — Pseudocode Corresponding to Section 5.5

```
morePatchesFlag = False

Repeat until SortedVertexList.Count <= 2
    If morePatchesFlag = False then
        changeAnchorFlag = True
        changeKedgeFlag = True
        anchor = 0
        kedge = 0
        directionFlag = 1
    Elseif changeAnchorFlag then                      check to see if the anchor is to be advanced

        anchor = anchor + 1
        directionFlag = - 1 * directionFlag
        changeAnchorFlag = False
    Elseif changeKedgeFlag                            check to see if the kedge is to be retreated

        kedge = kedge + 1
        directionFlag = - 1 * directionFlag
        changeKedgeFlag = False
    Else                                              try to build a patch
        If directionFlag = 1 then                     set the vertexListPointer

            vertexListPointer = anchor
        Elseif directionFlag = - 1
            vertexListPointer = kedge
        Endif
        newPatchVertex( 1 ) = SortedVertexList( side#, vertexListPointer )      assign the first vertex of the new patch

        If anchor + kedge +1 = SortedVertexList( side# ).Count                  check to see if the anchor or kedge will be met by the next vertex

            morePatchesFlag = False
        Elseif VertexList( side#, vertexListPointer + 1 * directionFlag ).Angle > 180 then      check to see if the angle at vertex 2 is interior or exterior (invalid patch)

            If directionFlag = 1 then
                changeAnchorFlag = True
            Else
                changeKedgeFlag = True
            Endif
        Else                                          continue patch building
            newPatchVertex( 2 ) =SortedVertexList( side#, vertexListPointer + 1 * directionFlag )      assign the second vertex of the new patch

            If anchor + kedge +2 = SortedVertexList( side# ).count      check to see if the anchor or kedge will be met by the next vertex forcing a three-sided patch

                newPatchVertex( 3 ) =SortedVertexList( side#, vertexListPointer + 2 * directionFlag )      assign the third vertex of the new patch

                newPatchVertex( 4 ) =newPatchVertex( 3 )      assign the fourth vertex of the new patch

                morePatchesFlag = False
                VerifyNewPatch
            Elseif SortedVertexList( side#, vertexListPointer + 2 * directionFlag ).Angle > 180 then      check interior angle at the third vertex of the new patch
```

```
                newPatchVertex( 3 ) =SortedVertexList( side#, vertexListPointer + 2 * directionFlag )   assign the third vertex
                                                                                                          of the new patch
                newPatchVertex( 4 ) =newPatchVertex( 3 )                                                  assign the fourth vertex
                                                                                                          of the new patch
                VerifyNewPatch
            Else                                                                                          create a four-sided patch
                newPatchVertex( 3 ) =SortedVertexList( side#, vertexListPointer + 2 * directionFlag )   assign the third vertex
                                                                                                          of the new patch
                newpatchVertex( 4 ) =SortedVertexList( side#, vertexListPointer + 3 * directionFlag )   assign the fourth vertex
                                                                                                          of the new patch
                If anchor + kedge +2 = SortedVertexList( side# ).count                                    check to see if the
                                                                                                          anchor or kedge will be
                                                                                                          met by the next vertex
                                                                                                          forcing a three-sided
                                                                                                          patch
                    morePatchesFlag = False
                Endif
                VerifyNewPatch
            Endif
        Endif
    Endif
Loop
```

Interference Checking — Pseudocode Corresponding to Section 5.6

Sub VerifyNewPatch — Pseudocode

```
angle = FindAngle( newPatchVertex( 4 ), newPatchVertex( 1 ), newPatchVertex( 2 ) )   find angle between vertices 4, 1, and 2
sideSolution = FindSide( SortedVertexList( side# ).Normal,
                newPatchVertex( 4 ), newPatchVertex( 1 ), newPatchVertex( 2 ) )   determine side for interior or exterior angle
If sideSolution < 0 then                                                             correct exterior angle
    angle = 360 - angle
Endif

If angle > 180 then                                                                  If angle is exterior then patch is invalid
    newPatchVertex( 4 ) = newPatchVertex( 3 )                                        Create a three-sided patch
Endif

SavePatchFlag = InterferenceCheck( SortedVertexList( side# ), newPatchVertex)        Check side 4-1 or 3-1 for interference with other
                                                                                     vertices
If ( newPatchVertex( 4 ) <> newPatchVertex( 3 ) ) and SavePatchFlag = False          If patch failed and is four-sided then make
                                                                                     three-sided and recheck.
    newPatchVertex( 4 ) = newPatchVertex( 3 )
    SavePatchFlag = InterferenceCheck( VertexList( side# ), newPatchVertex)
Endif

If SavePatchFlag = False then                                                        If patch fails then discard and reset the creation
                                                                                     direction
    directionFlag = - 1 * directionFlag
Else                                                                                 Check orientation of the new patch
    vector1.x = newPatchVertex( 2 ).x - newPatchVertex( 1 ).x
    vector1.y = newPatchVertex( 2 ).y - newPatchVertex( 1 ).y
    vector1.z = newPatchVertex( 2 ).z - newPatchVertex( 1 ).z
    vector2.x = newPatchVertex( 3 ).x - newPatchVertex( 2 ).x
    vector2.y = newPatchVertex( 3 ).y - newPatchVertex( 2 ).y
    vector2.z = newPatchVertex( 3 ).z - newPatchVertex( 2 ).z
    crossProduct.a = vector1.y * vector2.z - vector2.y * vector1.z
    crossProduct.b = vector1.x * vector2.z - vector2.x * vector1.z
    crossProduct.c = vector1.x * vector2.y - vector2.x * vector1.y
    If ( crossProduct.a <>VertexList( side# ).normal.a
        AND crossProduct.b <>VertexList( side# ).normal.b
            AND crossProduct.c <>VertexList( side# ).normal.c ) then
        junk = newPatchVertex( 2 )
        newPatchVertex( 2 ) = newPatchVertex( 3 )
        newPatchVertex( 3 ) = junk
        If vCount = 3 then
            newPatchVertex( 4 ) = newPatchVertex( 3 )
        Endif
    Endif
    SavePatch                  Save the new patch
    RemoveTrappedVertices  Remove trapped vertices from Vertex List
Endif
```

Sub InterferenceCheck — Pseudocode

```
( SortedVertexList( side# ), newPatchVertex)

If newPatchVertex( 4 ) = newPatchVertex( 3 ) then
    vCount = 3
Else
    vCount = 4
Endif
For I = ( vCount + 1 ) to VertexList( side# ).Count
    sideSolution = FindSide(
                    tempVertexList( side# ).Normal, newPatchVertex( 4 ), newPatchVertex( 1 ), newPatchVertex( 2 ) )
    If sideSolution ≥ then
        Set SavePatchFlag = False
        I = VertexList( side# ).count
    Else
        Set SavePatchFlag = True
    Endif
Next i
```

Module: DXF Face Import Code

```
Const InFName = "C:\workingf\dxffiles\ship.dxf"      ' Name of the input file. This will be removed once a user interface is created.

Dim InFNum As Integer                                 ' Input file number
Dim InPos As Long                                     ' Position flag for the current position in the input file
Dim InLength As Long                                  ' A variable indicating the length of the Input File

Dim NewObject As Integer                              ' A boolean flag indicating the end of an object
```

Function DecomposeHEFlag

```
(edge As Integer, hEVal As Integer) As Integer        ' Boolean
```

This function takes an edge and a Hidden Edge integer value read from the input .DXF file, and determines if the edge is visible or hidden. It returns a true or false boolean.

The hEVal value is the sum of the following:

1 if edge1 is hidden
2 if edge2 is hidden
4 if edge3 is hidden
8 if edge4 is hidden

```
Dim flag As Integer                                   ' Boolean

flag = False

If hEVal = 0 Then                                     ' No edges are hidden
    flag = False
ElseIf hEVal = 15 Then                                ' All edges are hidden
    flag = True
ElseIf edge = 1 Then                                  ' Top edge is of interest
    Select Case hEVal
    Case 1
        flag = True
    Case 3
        flag = True
    Case 5
        flag = True
    Case 7
        flag = True
    Case 9
        flag = True
    Case 11
        flag = True
    Case 13
        flag = True
    End Select
ElseIf edge = 2 Then                                  ' Right hand edge is of interest
    Select Case hEVal
    Case 2
        flag = True
    Case 3
        flag = True
```

```
        Case 6
            flag = True
        Case 7
            flag = True
        Case 10
            flag = True
        Case 11
            flag = True
        Case 14
            flag = True
        End Select
    ElseIf edge = 3 Then                          Bottom edge is of interest
        Select Case hEVal
        Case 4
            flag = True
        Case 5
            flag = True
        Case 6
            flag = True
        Case 7
            flag = True
        Case 12
            flag = True
        Case 13
            flag = True
        Case 14
            flag = True
        End Select
    ElseIf edge = 4 Then                          Left edge is of interest
        Select Case hEVal
        Case 8
            flag = True
        Case 9
            flag = True
        Case 10
            flag = True
        Case 11
            flag = True
        Case 12
            flag = True
        Case 13
            flag = True
        Case 14
            flag = True
        End Select
    End If

    DecomposeHEFlag = flag

    End Function
```

Sub DigestPatch

(pt() As Point3DDouble, hEVal As Integer)

This routine digests each patch read from the .DXF file and stores the information in the appropriate tables in the database.

Dim lastRecord As Long	*A placeholder variable*
Dim i As Integer	*An array index variable*
lastRecord = SeekLastRecord("ACTIVE", (Ptable.Name))	*Get the value of the last record in the Patch List table*
Ptable.AddNew	*Add a new record to the table*
PTable.Fields("Patch_ID") = lastRecord + 1	*Set the Patch_ID of the new record*
PTable.Fields("Space_ID") = SLTable.Fields("Space_ID")	*Set the Space_ID of the new record to the current SLTable entry*
lastRecord = SeekLastRecord("ACTIVE", (Ctable.Name))	*Get the value of the last record in the Patch Corners table*
Ctable.AddNew	*Add a new record to the table*
CTable.Fields("Patch_ID") = lastRecord + 1	*Set the Patch_ID of the new record*
CTable.Fields("Patch_ID") = Ptable.Fields("Patch_ID")	*Set the Patch_ID field in the Patches Corners table to the current Patch counter value*
lastRecord = SeekLastRecord("ACTIVE", (HETable.Name))	*Get the value of the last record in the Patch Corners table*
HETable.AddNew	*Add a new record to the table*
HETable.Fields("Patch_ID") = lastRecord + 1	*Set the Patch_ID of the new record*
HETable.Fields("Patch_ID") = Ptable.Fields("Patch_ID")	*Set the Patch_ID field in the Patches Hidden Edges table to the current Patch counter value*
For i = 1 To 4	*Begin looping through the four corners of the patch*
lastRecord = SeekLastRecord("ACTIVE", (Vtable.Name))	*Get the value of the last record in the Patch Vertex table*
Vtable.AddNew	*Add a new record to the table*
VTable.Fields("Vertex_ID") = lastRecord + 1	*Set the Vertex_ID of the new record*
VTable.Fields("x") = pt(i).x	*Set the X coordinate field in the Patches Vertex table to the X coordinate of the ith patch corner*
VTable.Fields("y") = pt(i).y	*Set the Y coordinate field in the Patches Vertex table to the Y coordinate of the ith patch corner*
VTable.Fields("z") = pt(i).z	*Set the Z coordinate field in the Patches Vertex table to the Z coordinate of the ith patch corner*
CTable.Fields(i) = Vtable.Fields("Vertex_ID")	*Set the ith Corner field in the Patches Corners table to the Vertex_ID value*
Vtable.Update	*Compete the change to the VTable*
HETable.Fields(i) = DecomposeHEFlag(i, hEVal)	*Call the DecomposeHEFlag function and determine if the ith edge is hidden. Store this result in the ith field of the Patches Hidden Edges table.*
Next i	
Ptable.Update	*Complete the change to the PTable*
Ctable.Update	*Complete the change to the CTable*
HETable.Update	*Complete the change to the HETable*
End Sub	

Sub DXFImportMain

This is the routine which controls the DXF Import routines. It reads a .DXF file containing a hull definition based on 3D Faces. It then stores the facial information in tables in the database so that it can be manipulated and exported in the form of a 3D polymesh or polysurface.

InFNum = FreeFile	*Assign the input file number to the next available file number*
Open InFName For Input As InFNum	*Open the Input File*
InLength = LengthOfFile(InFNum)	*Call the function LengthOfFile to determine the number of lines in the file*
InPos = 1	*Set the file input position variable to the first row of the input file*
Headers	*Call the Headers subroutine to scan through the initial entries of the Input file*
SetUpTables	*Prepare the tables used by this routine*
NewObject = True	*Set the NewObject flag to true*
Do While InPos <= InLength And NewObject	*Loop until the end of the file is reached or a new surface is initiated. (Assumes that a surface cannot occupy two layers)*
NewObject = False	*Sets the NewObject flag to end the loop following the Ingestion subroutine*
ActiveDB.BeginTrans	
NameNewSpace	*Call the routine to add a new space_ID and name to the database to represent the new object.*
IngestDXFFaces	*Call the IngestDXFFaces subroutine to read the ENTITIES section of the .DXF file and store this information in the database.*
ActiveDB.CommitTrans	
Loop	
CloseTables	*Clear all the table variables*
End Sub	

Sub Headers

This routine scans the header and section header information of the temporary input file. The input routines are only interested in the entity sections of the DXF file. It stops reading when the "ENTITIES" marker is found.

```
Dim entity As Integer                              ' Boolean
Dim inputLine As Variant                           ' Contains whatever information contained in a line of the .DXF file

entity = False
Do While (InPos <= InLength) And Not entity        ' Loop until the position in the file exceeds the length of the file or until the entity flag is true.

    Input #InFNum, inputLine                       ' Get a line from .DXF
    InPos = InPos + 1                              ' Advance the line counter one line

    If inputLine = "ENTITIES" Then                 ' If contents of the line are "ENTITIES"
        entity = True                              ' Set stop flag to true
    End If
Loop

End Sub
```

Sub CloseTables

Close the tables used in this module.

```
PTable.Close
CTable.Close
VTable.Close
HETable.Close
SLTable.Close

End Sub
```

Sub IngestDXFFaces

This routine takes the facial information from the input file and assigns it to several tables in the database. It also collects the other information contained in the input file.

It is a long and messy section but once you understand it you may consider it to be rather elegant.

DXF Files are broken into two line codes, the first being a Group Code and the second a numeric value or text string appropriate for the group code.

```
Dim i As Integer                              Counter variable
Dim curObj As String                          Current object name
Dim dataText As String                        Generic text string from the DXF file
Dim dataValue As Double                       Generic numeric value from the DXF file
Dim patchUpdated As Integer                       Boolean   - flag to indicate the completion of a
                                                  face
Dim endSec As Integer                         Boolean   - flag to indicate the end of a DXF object
                                              section
Dim groupCode As Integer                      DXF group code value
Dim hEdgeValue As Integer                     Hidden edge value
Dim prevLinePos As Long                       Previous line position storage variable
Dim blockLinePos As Long                      Block line position storage variable
Static cornerPt(4) As Point3DDouble           Patch variable
Dim inputLine As Variant                      Raw line read from the DXF file
Dim layerName As String


patchUpdated = False                          Clear the flag requiring a writing of a patch
endSec = False                                Clear the flag marking the end of an input section
layerName = ""                                Clear the layer name variable


Do While InPos <= InLength And Not endSec     Loop until the current position is beyond the length of
                                              the file or the end of the section is reached.

    prevLinePos = Seek(InFNum)                Store the current file position

    Line Input #InFNum, inputLine             Get a line from .DXF
    groupCode = Val(inputLine)                Assign this value as the Group Code
    InPos = InPos + 1                         Increment the Input File Position variable
    Line Input #InFNum, inputLine             Get a line from .DXF
    InPos = InPos + 1                         Increment the Input File Position variable

    If groupCode = 0 Then                     Test group code for a new object definition
        blockLinePos = prevLinePos            Store the block start position
        curObj = Ucase(inputLine)             Assign the string found on the inputLine as the
                                              Current Object
    ElseIf groupCode < 10 Then                Test group code for the presence of a string entry
        dataText = inputLine                  Assign the input line to a generic string data variable
    Else
        dataValue = inputLine                 Assign the input line to a generic numeric data
                                              variable
    End If

    Select Case groupCode                     Classify the group code
            groupcode < 10                    cardinal group codes
    Case 0                                    Start of entity, table, file separator
        If patchUpdated Then
            DigestPatch cornerPt(), hEdgeValue    Finish off previous entity
            Erase cornerPt
```

`hEdgeValue = 0`	*Clear the hidden edge value*
`patchUpdated = False`	*Clear the flag requiring a writing of a patch*
`End If`	
`If InStr(1, curObj, "EOF", 1) Then`	*Check to see that the current object is not an End of File marker. If so...*
`Seek #InFNum, prevLinePos`	*Back up two lines*
`InPos = InPos - 2`	*Back up the File Input Position variable two lines*
`endSec = True`	*Set the End of Section flag to true to complete the ingestion*
`ElseIf InStr(1, curObj, "ENDSEC", 1) Then`	*ENTITY section is complete here...*
`Seek #InFNum, prevLinePos`	*Back up two lines*
`InPos = InPos - 2`	*Back up the File Input Position variable two lines*
`endSec = True`	*Set the End of Section flag to true to complete the ingestion*
`ElseIf Not OkObject(curObj) Then`	*Test the Current Object for digestability using the OkObject function. If the object is not digestable...*
`Do`	
`prevLinePos = Seek(InFNum)`	*Set the Input position marker to the current position*
`Line Input #InFNum, inputLine`	*Get a line from .DXF and*
`InPos = InPos + 1`	*Advance through the InputDXF array to the next object*
`Loop Until InStr(1, inputLine, "0", 1) > 0 Or InPos = InLength`	*Repeat line-by-line advance until a new section 0 code appears or until the current position in the file is the same as the length of the file.*
`Seek #InFNum, prevLinePos`	*Reset position marker for Next loop to start at the zero group code*
`InPos = InPos - 1`	*Back up the File Input Position variable one line (one line since the advance stopped on a group code)*
`End If`	
`Case 1`	*Primary text value for entity (?)*
`Case 2`	*Block name, attribute tag, etc*
`Case 3`	*Other names*
`Case 4`	
`Case 5`	*Entity handle (hex string)*
`Case 6`	*Line type name is next string*
`Case 7`	*Text style name*
`Case 8`	*Layer name is next string*
`If (layerName <> dataText) And (layerName <> "") Then`	*Test to see if the layer name has changed. If so...*
`Seek #InFNum, blockLinePos`	*Move back 4 lines*
`InPos = InPos - 4`	*Back up the File Input Position counter 4 lines*
`NewObject = True`	*Set the flag indicating a new layer to true*
`Exit Do`	*Exit the Do loop (and therefore the subroutine)*
`Else`	
`layerName = dataText`	*Otherwise set the Layer Name variable equal to the text string in the dataText variable.*
`End If`	
`Case 9`	*Variable name ID (only in header)*
VERTEX	
`Case 10 To 18`	*Some X coord of a vertex*
`cornerPt(groupCode - 10 + 1).x = dataValue`	*Assign the current corner point X coordinate to be the value in the numeric dataValue variable.*
`patchUpdated = True`	*Set the flag to indicate that a new patch is to be written to the database at the appropriate time.*
`Case 20 To 28`	*Some Y coord of a vertex*
`cornerPt(groupCode - 20 + 1).y = dataValue`	*Assign the current corner point Y coordinate to be the value in the numeric dataValue variable.*
`patchUpdated = True`	*Set the flag to indicate that a new patch is to be written to the database at the appropriate time.*

Case 30 To 38	*Some Z coord of a vertex*
cornerPt(groupCode - 30 + 1).z = dataValue	*Assign the current corner point Z coordinate to be the value in the numeric dataValue variable.*
patchUpdated = True	*Set the flag to indicate that a new patch is to be written to the database at the appropriate time.*
END VERTEX	
Case 38	*Entity elevation if nonzero*
Case 39	*Entity thickness if nonzero*
Case 40 To 48	*Misc doubles*
'doubles(groupCode - 40) = dataValue	
Case 49	*Repeated value groups*
Case 50 To 58	*Misc angles*
'angles(groupCode - 50) = dataValue	
Case 62	*Color number*
'curColor = dataValue	
Case 66	*"ENTITIES FOLLOW" flag*
Case 70 To 78	*Misc ints*
hEdgeValue = Int(dataValue)	*Assign the Hidden Edge variable to the value of the dataValue variable*
'ints(groupCode - 70) = dataValue	*For 3DFACEs:*
'for POLYLINEs:	*1 means first edge is invisible*
'70 = 64 means polymesh	*2 means second edge is invisible*
'71 is vertex count	*4 means third edge is invisible*
'72 is face count	*8 means fourth edge is invisible*
'	
Case 210 Or 220 Or 230	*X, Y, Z components of extrusion direction*
End Select	

```
Loop

End Sub
```

Function OkObject

(curObj As String) As Integer *Boolean*

This routine examines the entity found in the .DXF file and determines if it can be digested by the DXFIngest & DXFDigest routines. As you can see, there is room for much work here.

Currently, it can only deal with 3DFaces.

```
If InStr(1, curObj, "3DFACE", 1) Then
    OkObject = True

'This section has not been implemented
'ElseIf InStr(1, curObj, "TRACE", 1) Then
'ElseIf InStr(1, curObj, "SOLID", 1) Then
'ElseIf InStr(1, curobj, "LINE", 1) Then              Degenerate triangle? use a line entity?
'ElseIf InStr(1, curobj, "3DLINE", 1) Then            A VERY skinny triangle!
'ElseIf InStr(1, curobj, "POINT", 1) Then             Tiny sphere
'ElseIf InStr(1, curobj, "CIRCLE", 1) Then            A short cylinder
'ElseIf InStr(1, curobj, "ARC", 1) Then               Not implemented for now
'ElseIf InStr(1, curobj, "TEXT", 1) Then              Not implemented for now
'ElseIf InStr(1, curobj, "SHAPE", 1) Then             These look very hard
'ElseIf InStr(1, curobj, "BLOCK", 1) Then             These look very hard
'ElseIf InStr(1, curobj, "ENDBLK", 1) Then            These look very hard
'ElseIf InStr(1, curobj, "INSERT", 1) Then            These look very hard
'ElseIf InStr(1, curobj, "ATTDEF", 1) Then            Not implemented for now
'ElseIf InStr(1, curobj, "ATTRIB", 1) Then            Not implemented for now
'ElseIf InStr(1, curobj, "POLYLINE", 1) Then          These look fairly hard
'ElseIf InStr(1, curobj, "VERTEX", 1) Then            These look fairly hard
'ElseIf InStr(1, curobj, "SEQEND", 1) Then            These look fairly hard
'ElseIf InStr(1, curobj, "DIMENSION", 1) Then         Not implemented for now
'ELSE no current object defined...

Else OkObject = False

End If

End Function
```

Sub SetUpTables

This subroutine assigns variables to each of the tables affected by this module and sets their indexes to follow each table's ID values.

```
Set CTable = ActiveDB.OpenRecordset("Patch Corners", DB_OPEN_TABLE)
Set HETable = ActiveDB.OpenRecordset("Patch Hidden Edges", DB_OPEN_TABLE)
Set VTable = ActiveDB.OpenRecordset("Vertex List", DB_OPEN_TABLE)
Set PTable = ActiveDB.OpenRecordset("Patch List", DB_OPEN_TABLE)
Set SLTable = ActiveDB.OpenRecordset("Space List", DB_OPEN_TABLE)

PTable.Index = "PrimaryKey"
CTable.Index = "PrimaryKey"
HETable.Index = "PrimaryKey"
VTable.Index = "PrimaryKey"
SLTable.Index = "PrimaryKey"

End Sub
```

Function LengthOfFile

(FNum)

This function determines the number of lines in the input DXF file

```
Dim junk As Variant
Dim counter As Long

Do While Not EOF(FNum)                 Loop until the end of file marker is found
    Line Input #FNum, junk             Read the file FNum line by line
    counter = counter + 1              Count the number of lines
Loop

Seek #FNum, 1                          Move the file pointer to the beginning of the file
LengthOfFile = counter

End Function
```

Sub NameNewSpace

This routine adds a new entry to the SLTable, and creates a new name for that entry.

```
Dim i As Integer                                        A counter variable
Dim lastRecord As Long                                  A position marker

i = 1                                                   Clear the counter

'SLTable.Index = "Space_Name"                           Set the index of the SLTable to Space_Name

'SLTable.Seek "=", ("Imported Object -" & Str$(i))      Seek a space name
'Do Until SLTable.NoMatch                               Repeat until the name is not found
'    SLTable.Seek "=", ("Imported Object -" & Str$(i))
'    i = i + 1                                          Increment the counter
'Loop

lastRecord = SeekLastRecord("ACTIVE", (SLTable.Name))   Get the last record number

SLTable.AddNew                                          Add the new entry
SLTable.Fields("Space_ID") = lastRecord + 1             Increment the Space_ID for the new record
'SLTable.Fields("Space_Name") = "Imported Object -" & Str$(i)   Name the new space
SLTable.Fields("Space_Name") = InFName                  Name the new space
SLTable.Update                                          Complete the entry
SLTable.MoveLast          Move to the last record in the SLTable

End Sub
```

Module: DXF Export Code

Const OutFName = "c:\workingf\tempfile\demo.dxf"	*Constant for the output file name. This will eventually be tied into a user specified item involving a form*
Const Triangles = True	*Constant refering to the type of output - squares or triangles. This will eventually be tied into a user specified item involving a form*
Const Mirror = True	*Constant refering to the type of output - the hull files from Autoship are generally just one side of the hull. This flag, which will also become a user specified item, represents this.*
Const ObjType = "3DFACES"	*Constant refering to the type of object to be created in DXF*
Dim OutFNum As Integer	*A file number variable for the output file*
Dim PatchQ As QueryDef	*The name of a query definition which creates a list of all the patches associated with a particular Space_ID number*
Dim TriPatchQ As QueryDef	*The name of a query definition which creates a list of all triangular patches associated with a particular Space_ID number*
Dim VertexQ As QueryDef	*The name of a query definition which creates a list of all the vertices associated with a particular Space_ID number*
Dim VertexTabQ As QueryDef	*The name of a query definition which reduces the VertexQ query results into a single column of vertices*
Dim PatchQTable As Recordset	*The name of a table created by the output of the PatchQ query*
Dim TriPatchQTable As Recordset	*The name of a table created by the output of the TriPatchQ query*
Dim VertexQTable As Recordset	*The name of a table created by the output of the VertexTabQ query*
Dim OutputQTable As Recordset	*The name of a table which contains a renumber set of vertices and the information required to crosslink these to the other tables in this database*

Sub CreateOutputQTable

Unlike the tables created by query in this database, it was necessary to formally create a table so that a numeric counter field could be added.

The routine is fairly self explanatory. It essentially creates each of the elements of the table (Fields and Indices) and appends these to the newTableDef definition. In turn, this definition is appended to the TemporaryDB.TableDefs collection, thus creating the table.

The routine GetOutputQTable actually contains the commands required to fill in the contents of this table.

```
Dim f1 As New Field
Dim f2 As New Field
Dim f3 As New Field
Dim f4 As New Field
Dim f5 As New Field
Dim i1 As New Index
Dim i2 As New Index
Dim newTblDef As New TableDef
```

```
newTblDef.Name = "OutputQTable"                  ' Name the new table

f1.Name = "New_Vertex_ID"                        ' Create fields
f1.Type = DB_LONG

f2.Name = "Old_Vertex_ID"
f2.Type = DB_LONG

f3.Name = "X"
f3.Type = DB_DOUBLE

f4.Name = "Y"
f4.Type = DB_DOUBLE

f5.Name = "Z"
f5.Type = DB_DOUBLE

newTblDef.Fields.Append f1                       ' Add it to the collection
newTblDef.Fields.Append f2                       ' Add it to the collection
newTblDef.Fields.Append f3                       ' Add it to the collection
newTblDef.Fields.Append f4                       ' Add it to the collection
newTblDef.Fields.Append f5                       ' Add it to the collection


i1.Name = "New_Vertex_ID"                        ' Create indices
i1.Fields = "New_Vertex_ID"
i1.Primary = True

i2.Name = "Old_Vertex_ID"
i2.Fields = "Old_Vertex_ID"

newTblDef.Indexes.Append i1                      ' Add it to the collection
newTblDef.Indexes.Append i2                      ' Add it to the collection


TemporaryDB.TableDefs.Append newTblDef           ' Now append the new Table object to the TableDefs collection.


End Sub
```

Sub CreatePolyMesh

```
(flag As Integer, layerName As String)

MeshHeaderOutput layerName
```

Call the routine which creates the beginning of a polymesh entity in a .DXF file

```
MeshVertexOutput flag, layerName
```

Call the routine which places all the vertex information into a .DXF polymesh entity

```
MeshPatchOutput flag, layerName
```

Call the routine which places all the patch pointers (pointing to vertexes) into a .DXF polymesh entity

```
Print #OutFNum, 0
Print #OutFNum, "SEQEND"
Print #OutFNum, 8
Print #OutFNum, layerName
```

Finish the .DXF polymesh entity

```
End Sub
```

Sub DXFExportMain

This is the routine which controls the DXF Export routines. It creates a .DXF file which includes a separate mesh for each object in the database.

Ordinarily, one would create the temporary database at the beginning of this routine and delete it at the end. Unfortunately, I have not been able to figure out how to get Access to relinquish its locks on the TemporaryDB at the end of the routine. Therefore, the routine leaves an empty TemporaryDB on disk. The file is empty because of the use of the transactions commands in the loop. It is purged at the beginning of a run instead as part of the PrepareTemporaryDB routine.

```
PrepareActiveTables
PrepareOutputFile
PrepareTemporaryDB
```

Initialize ActiveDB tables used in this module
Initialize the output file
Create the temporary database

```
FileHeader
```

Write the DXF file Header to the output file

```
SLTable.Index = "PrimaryKey"
```

Set the index of the SLTable to the Space_ID number (the primary key)

```
SLTable.MoveFirst
Do Until SLTable.EOF
```

For each space in the space table...

```
    TemporaryDB.BeginTrans
```

Begin the transaction - this means that all changes until the .CommitTrans or .RollBack commands are reached take place in memory and are therefore faster.

```
    GetPatches (SLTable.Fields("Space_ID"))
```

Call a routine to generate a table of patch values for the current space

```
    Set PatchQTable = TemporaryDB.OpenRecordset("PatchQTable", DB_OPEN_TABLE)

    GetTriPatches (SLTable.Fields("Space_ID"))
```

Call a routine to generate a table of triangular patches for the current space

```
    Set TriPatchQTable = TemporaryDB.OpenRecordset("TriPatchQTable", DB_OPEN_TABLE)

    GetVertices (SLTable.Fields("Space_ID"))
```

all a routine to generate a table of vertex pointers for the current space

```
    Set VertexQTable = TemporaryDB.OpenRecordset("VertexQTable", DB_OPEN_TABLE)
```

```
        CreateOutputQTable                                                          Calls a routine to create the OutputQTable definition.
        Set OutputQTable = TemporaryDB.OpenRecordset("OutputQTable", DB_OPEN_TABLE)  Assign the variable
                                                                                     OutputQTable to represent
                                                                                     the table of the same name
        GetOutputQTable                                                             Call a routine to create a table containing renumbered
                                                                                    vertex pointers

        If ObjType = "PolyMesh" Then
        CreatePolyMesh 1, (SLTable.Fields("Space_ID"))                              Not Mirror, Not Inside
        CreatePolyMesh 2, (SLTable.Fields("Space_ID"))                              Not Mirror, Inside
        If Mirror Then
            CreatePolyMesh 3, (SLTable.Fields("Space_ID"))                          Mirror, Not Inside
            CreatePolyMesh 4, (SLTable.Fields("Space_ID"))                          Mirror, Inside
        End If
        ElseIf ObjType = "3DFACES" Then
        FaceOutput 1, (SLTable.Fields("Space_ID"))                                  Not Mirror, Not Inside
'       FaceOutput 2, (SLTable.Fields("Space_ID"))                                  Not Mirror, Inside
'       If Mirror Then
'           FaceOutput 3, (SLTable.Fields("Space_ID"))                              Mirror, Not Inside
'           FaceOutput 4, (SLTable.Fields("Space_ID"))                              Mirror, Inside
'       End If
        End If

        PatchQTable.Close
        TriPatchQTable.Close
        VertexQTable.Close
        OutputQTable.Close

        TemporaryDB.Rollback                                                        Purge all changes to the temporaryDB following the
                                                                                    .BeginTrans command
        SLTable.MoveNext

    Loop

    FileFooter                                                                      Write the .DXF file footer

    Close OutFNum                                                                   Close the .DXF file

    End Sub
```

Sub FaceOutput

(flag As Integer, layerName As String)

This routine writes the information required for each polyface entry in a .DXF file

flag values: 1 = primary view
2 = backface or inside face of an exterior primary view
4 = mirrored primary view
8 = back face or inside face of an exterior mirrored view

Note that the routine uses only one writing function to set all of this up. If you make a chart on a piece of paper you will see that each of these views can be created by manipulating the corner positions of the selected surface.

Hidden edge flag values: 1 = First Edge is Invisible
2 = Second Edge is Invisible
4 = Third Edge is Invisible
8 = Fourth Edge is Invisible

```
ReDim cPt(4) As Point3DDouble          ' This array contains vertex pointers for each corner of a patch

Dim pointer As Long

CTable.Index = "PrimaryKey"            ' Set the index of the Patch Corners table to the Patch_ID value

OutputQTable.Index = "Old_Vertex_ID"   ' Set the index of the OutputQTable to the Patch_ID value

PatchQTable.MoveFirst
Do Until PatchQTable.EOF
     pointer = PatchQTable.Fields("Patch_ID")     ' Set the pointer equal to the patch_ID of the current record in the PatchQTable

     CTable.Seek "=", PatchQTable.Fields("Patch_ID")     ' Find the current patch in the Patch Corner's table
     For i = 1 To 4
          OutputQTable.Seek "=", Ctable.Fields(i)     ' Find the corner pointers in the CTable in the OutputQTable

          cPt(i).x = OutputQTable.Fields("X")     ' Find the corner vertex coordinates in the VertexTable
          cPt(i).y = OutputQTable.Fields("Y")     ' And store the coordinate values
          cPt(i).z = OutputQTable.Fields("Z")
     Next i

     If flag = 2 Or flag = 8 Then     ' Check to see if the object has been mirrored
          SwapPts cPt(1), cPt(2)      ' And swap the appropriate points
          SwapPts cPt(3), cPt(4)
     End If

     Print #OutFNum, 0
     Print #OutFNum, "3DFACE"
     Print #OutFNum, 8
     Print #OutFNum, layerName

     If Triangles Then     ' This section takes a four-cornered surface and creates in the dxf-file 2 triangular surfaces.

          If EqualPts(cPt(3), cPt(4)) Then     ' Test for a triangular face for the four-cornered surface. If true then we can plot this just as if four corners were acceptable. Later editions of the graphics routines will be able to use this more efficient format. In the mean time we must use triangle based meshes.

               For i = 1 To 4
```

```
            Print #OutFNum, 10 + i - 1
            Print #OutFNum, cPt(i).x
            Print #OutFNum, 20 + i - 1
            Print #OutFNum, cPt(i).y
            Print #OutFNum, 30 + i - 1
            Print #OutFNum, cPt(i).z
        Next i
        HEValue = 0
        If Not HiddenEdgeFlag(pointer, 12) Then HEValue = HEValue + 1
        If Not HiddenEdgeFlag(pointer, 23) Then HEValue = HEValue + 2
        If Not HiddenEdgeFlag(pointer, 34) Then HEValue = HEValue + 4
        HEValue = HEValue + 8                       Blank side which is non-existent
        Print #OutFNum, 70                          Vertex flag
        Print #OutFNum, HEValue
    Else                                            If not equal points 3 and 4, create a triangular face
                                                    from a square face.
        For i = 1 To 4
            Print #OutFNum, 10 + i - 1
            If i = 4 Then
                Print #OutFNum, cPt(3).x
            Else
                Print #OutFNum, cPt(i).x
            End If
            Print #OutFNum, 20 + i - 1
            If i = 4 Then
                Print #OutFNum, cPt(3).y
            Else
                Print #OutFNum, cPt(i).y
            End If
            Print #OutFNum, 30 + i - 1
            If i = 4 Then
                Print #OutFNum, cPt(3).z
            Else
                Print #OutFNum, cPt(i).z
            End If
        Next i
        HEValue = 0
        If Not HiddenEdgeFlag(pointer, 12) Then HEValue = HEValue + 1
        If Not HiddenEdgeFlag(pointer, 23) Then HEValue = HEValue + 2
        HEValue = HEValue + 4                       Blank shared diagonal side
        HEValue = HEValue + 8                       Blank side which is non-existent
        Print #OutFNum, 70                          Vertex flag
        Print #OutFNum, HEValue

        Print #OutFNum, 0
        Print #OutFNum, "3DFACE"
        Print #OutFNum, 8
        Print #OutFNum, layerName
        For i = 1 To 4
            Print #OutFNum, 10 + i - 1
            If ((i = 2) Or (i = 3)) Then
                Print #OutFNum, cPt(i + 1).x
            ElseIf ((i = 1) Or (i = 4)) Then
                Print #OutFNum, cPt(i).x
            End If
            Print #OutFNum, 20 + i - 1
            If ((i = 2) Or (i = 3)) Then
                Print #OutFNum, cPt(i + 1).y
            ElseIf ((i = 1) Or (i = 4)) Then
                Print #OutFNum, cPt(i).y
            End If
```

```
                    Print #OutFNum, 30 + i - 1
                    If ((i = 2) Or (i = 3)) Then
                        Print #OutFNum, cPt(i + 1).z
                    ElseIf ((i = 1) Or (i = 4)) Then
                        Print #OutFNum, cPt(i).z
                    End If
                Next i
'               HEValue = 0
'               HEValue = HEValue + 1                                          Blank shared diagonal side
'               If Not HiddenEdgeFlag(pointer, 23) Then HEValue = HEValue + 2
'               If Not HiddenEdgeFlag(pointer, 34) Then HEValue = HEValue + 4
'               HEValue = HEValue + 8                                          Blank side which is non-existent
'               Print #OutFNum, 70                                             Vertex flag
'               Print #OutFNum, HEValue
            End If
        ElseIf Not Triangles Then                                              This is a square mesh
            For i = 1 To 4
                Print #OutFNum, 10 + i - 1
                Print #OutFNum, cPt(i).x
                Print #OutFNum, 20 + i - 1
                Print #OutFNum, cPt(i).y
                Print #OutFNum, 30 + i - 1
                Print #OutFNum, cPt(i).z
            Next i
'           HEValue = 0
'           If Not HiddenEdgeFlag(pointer, 12) Then HEValue = HEValue + 1
'           If Not HiddenEdgeFlag(pointer, 23) Then HEValue = HEValue + 2
'           If Not HiddenEdgeFlag(pointer, 34) Then HEValue = HEValue + 4
'           If Not HiddenEdgeFlag(pointer, 41) Then HEValue = HEValue + 8
'           Print #OutFNum, 70                                                 Vertex flag
'           Print #OutFNum, HEValue
        End If

        PatchQTable.MoveNext

    Loop

    End Sub
```

Sub FileFooter

The contents of this routine are simply the last few lines required to complete a .DXF File.

```
Print #OutFNum, 0
Print #OutFNum, "ENDSEC"
Print #OutFNum, 0
Print #OutFNum, "EOF"

End Sub
```

Sub GetOutputQTable

This routine calls for the creation of a table called OutputQTable, sets a variable of the same name to represent it, and fills in all the entries for this table.

```
Dim i As Integer


If Not (VertexQTable.BOF And VertexQTable.EOF) Then VertexQTable.MoveLast   Check to see that there is an entry in the
                                                                            VertexQ table so that the .MoveLast command
                                                                            does not create an error.

VTable.Index = "PrimaryKey"                                                 Set the VTable index to the Vertex_ID values
                                                                            referred to in the PrimaryKey index
VertexQTable.MoveFirst
For i = 1 To VertexQTable.Recordcount                                       Loop through the entire VertexQTable

    OutputQTable.AddNew                                                     Add a new table entry
    OutputQTable.Fields("New_Vertex_ID") = I                                Assign the New_Vertex_ID to the i counter value
    OutputQTable.Fields("Old_Vertex_ID") = VertexQTable.Fields("Vertex")    Assign the Old_Vertex_ID to the vertex_ID stored
                                                                            in the VertexQTable
    VTable.Seek "=", VertexQTable.Fields("Vertex")                          Find the current vertex number in the VTable
    OutputQTable.Fields("X") = Vtable.Fields("X")                           Store the X value of the VTable in the
                                                                            OutputQTable
    OutputQTable.Fields("Y") = Vtable.Fields("Y")                           Store the Y value of the VTable in the
                                                                            OutputQTable
    OutputQTable.Fields("Z") = Vtable.Fields("Z")                           Store the Z value of the VTable in the
                                                                            OutputQTable
    OutputQTable.Update                                                     Complete the new entry

    VertexQTable.MoveNext
Next i

End Sub
```

Sub GetPatches

(ref_ID As Long)

This routine creates a tableQuery which stores a list of patches associated with a particular reference (or Space_ID) ID value.

Finally, the routine assigns a variable to the new table.

```
Set PatchQ = TemporaryDB.CreateQueryDef()

PatchQ.Name = "Count of Patches for Space_ID =" & Str$(ref_ID)

PatchQ.SQL = "SELECT DISTINCTROW [Patch List].Space_ID, [Patch List].Patch_ID "
PatchQ.SQL = PatchQ.SQL & "INTO PatchQTable FROM [Patch List] "
PatchQ.SQL = PatchQ.SQL & "IN " & Chr$(34) & Chr$(34) & "[DATABASE=" & (ActiveDB.Name) & ";] "
PatchQ.SQL = PatchQ.SQL & "GROUP BY [Patch List].Space_ID, [Patch List].Patch_ID "
PatchQ.SQL = PatchQ.SQL & "HAVING (([Patch List].Space_ID=" & Str$(ref_ID) & "));"

TemporaryDB.QueryDefs.Append PatchQ

PatchQ.Execute

End Sub
```

Sub GetTriPatches

(ref_ID As Long)

This routine creates a tableQuery which stores a list of three sided patches associated with a particular reference (or Space_ID) ID value.

Finally, the routine assigns a variable to the new table

```
Set TriPatchQ = TemporaryDB.CreateQueryDef()

TriPatchQ.Name = "Count of Triangular Patches for Space_ID =" & Str$(ref_ID)

TriPatchQ.SQL = "SELECT DISTINCT [Patch Corners].Patch_ID INTO TriPatchQTable "
TriPatchQ.SQL = TriPatchQ.SQL & "FROM [Patch Corners] "
TriPatchQ.SQL = TriPatchQ.SQL & "IN " & Chr$(34) & Chr$(34) & "[DATABASE=" & (ActiveDB.Name) & ";] "
TriPatchQ.SQL = TriPatchQ.SQL & "WHERE (((Exists (SELECT [Patch_ID] FROM [Patch List] "
TriPatchQ.SQL = TriPatchQ.SQL & "IN " & Chr$(34) & Chr$(34) & "[DATABASE=" & (ActiveDB.Name) & ";] "
TriPatchQ.SQL = TriPatchQ.SQL & "WHERE [Patch List].Space_ID=1))<>False) "
TriPatchQ.SQL = TriPatchQ.SQL & "AND ([Patch Corners].vertex4=[vertex3])) "
TriPatchQ.SQL = TriPatchQ.SQL & "GROUP BY [Patch Corners].Patch_ID;"

TemporaryDB.QueryDefs.Append TriPatchQ

TriPatchQ.Execute

End Sub
```

Sub GetVertices

(ref_ID As Long)

This routine uses two queries to generate the list of vertices associated with a particular ref_ID.

The first query, called VertexQ, is a union query which combines the four columns of the Patch Corners table of the ActiveDB into a single column. The column has no repeated values. The subSQL variable stores the repeated parts of the SQL definition.

The second query is called VertexTabQ and uses the information in the VertexQ query to generate a table in the TemporaryDB.

Finally, the routine assigns a variable to the new table.

```
Dim subSQL As String

subSQL = "FROM [Patch Corners] "
subSQL = subSQL & "IN " & Chr$(34) & Chr$(34) & "[DATABASE=" & (ActiveDB.Name) & ";] "
subSQL = subSQL & "WHERE EXISTS "
subSQL = subSQL & "(SELECT [Patch_ID] FROM [Patch List] "
subSQL = subSQL & "IN " & Chr$(34) & Chr$(34) & "[DATABASE=" & (ActiveDB.Name) & ";] "
subSQL = subSQL & "WHERE "
subSQL = subSQL & "(([Patch List].Space_ID=" & Str$(ref_ID) & "))) "

Set VertexQ = TemporaryDB.CreateQueryDef()

VertexQ.Name = "List of Vertices for Space_ID =" & Str$(ref_ID)

VertexQ.SQL = "SELECT DISTINCT [Vertex1] AS Vertex " & subSQL
VertexQ.SQL = VertexQ.SQL & "UNION SELECT [Vertex2] " & subSQL
VertexQ.SQL = VertexQ.SQL & "UNION SELECT [Vertex3] " & subSQL
VertexQ.SQL = VertexQ.SQL & "UNION SELECT [Vertex4] " & subSQL & ";"

TemporaryDB.QueryDefs.Append VertexQ


Set VertexTabQ = TemporaryDB.CreateQueryDef()

VertexTabQ.Name = "Table of " & VertexQ.Name
VertexTabQ.SQL = "SELECT DISTINCTROW [" & (VertexQ.Name) & "].Vertex "
VertexTabQ.SQL = VertexTabQ.SQL & "INTO VertexQTable "
VertexTabQ.SQL = VertexTabQ.SQL & "FROM [" & (VertexQ.Name) & "] "
VertexTabQ.SQL = VertexTabQ.SQL & "GROUP BY [" & (VertexQ.Name) & "].Vertex;"

TemporaryDB.QueryDefs.Append VertexTabQ

VertexTabQ.Execute

End Sub
```

Sub PrepareActiveTables

This routine initializes variables for the four ActiveDB tables used in this module.

```
Set HETable = ActiveDB.OpenRecordset("Patch Hidden Edges", DB_OPEN_TABLE)
Set CTable = ActiveDB.OpenRecordset("Patch Corners", DB_OPEN_TABLE)
Set VTable = ActiveDB.OpenRecordset("Vertex List", DB_OPEN_TABLE)
Set SLTable = ActiveDB.OpenRecordset("Space List", DB_OPEN_TABLE)

End Sub
```

Sub MeshHeaderOutput

(layerName As String)

This routine writes the brief header information required for each polymesh entry in a .DXF file.

The variable count is used to store a count of triangular surfaces to be written to the file. It has been assumed that the database may contain both triangular and rectangular entries. The count calculation ensures that the correct number of patches will appear in the .DXF file.

```
Dim count As Long

Print #OutFNum, 0
Print #OutFNum, "POLYLINE"
Print #OutFNum, 8                                   Layer name marker
Print #OutFNum, layerName
Print #OutFNum, 66                                  "vertices follow" code
Print #OutFNum, 1
Print #OutFNum, 70                                  70 bit code
Print #OutFNum, 64                                  "this polyline is a polyface mesh"
Print #OutFNum, 71
Print #OutFNum, VertexQTable.Recordcount            Number of vertices
Print #OutFNum, 72

If Triangles Then
    count = PatchQTable.Recordcount * 2 - TriPatchQTable.Recordcount
    Print #OutFNum, count                           Number of triangular facets to be made
Else
    Print #OutFNum, PatchQTable.Recordcount         Number of rectangular or triangular facets
End If

End Sub
```

Sub MeshPatchOutput

(flag As Integer, layerName As String)

This routine writes patch entries to the .DXF files. It employs the MeshVertexPrint routine for part of its output and creates the remainder.

flag values: 1 = primary view
2 = backface or inside face of an exterior primary view
4 = mirrored primary view
8 = back face or inside face of an exterior mirrored view

Note that the routine uses only one writing function to set all of this up. If you make a chart on a piece of paper you will see that each of these views can be created by manipulating the corner positions of the selected surface.

```
ReDim cPt(4) As Long                          ' This array contains vertex pointers for each corner of a patch

Dim nullPt As Point3DDouble                   ' A zero point
Dim pointer As Long

nullPt.x = 0                                  ' Define the zero point
nullPt.y = 0
nullPt.z = 0

CTable.Index = "PrimaryKey"                   ' Set the index of the Patch Corners table to the Patch_ID value

OutputQTable.Index = "Old_Vertex_ID"          ' Set the index of the OutputQTable to the Patch_ID value

PatchQTable.MoveFirst
Do Until PatchQTable.EOF
    pointer = PatchQTable.Fields("Patch_ID")  ' Set the pointer equal to the patch_ID of the current record in the PatchQTable

    CTable.Seek "=", PatchQTable.Fields("Patch_ID")   ' Find the current patch in the Patch Corner's table
    For i = 1 To 4
        OutputQTable.Seek "=", Ctable.Fields(i)       ' Copy find the corner pointers in the CTable in the OutputQTable

        cPt(i) = OutputQTable.Fields("New_Vertex_ID") ' And store the new vertex number
    Next i

    If flag = 2 Or flag = 8 Then              ' Check to see if the object has been mirrored
        SwapValues cPt(1), cPt(2)             ' and swap the appropriate points
        SwapValues cPt(3), cPt(4)
    End If

    If Triangles Then                         ' This section takes a four-cornered surface and creates in the dxf-file 2 triangular surfaces.

        If cPt(3) = cPt(4) Then               ' Test for a triangular face for the four-cornered surface. If true then we can plot this just as if four corners were acceptable. Later editions of the graphics routines will be able to use this more efficient format. In the mean time we must use triangle based meshes.

            MeshVertexPrint nullPt, 128, layerName   ' Means vertex is the face of a polyface mesh
            Print #OutFNum, 71                       ' This creates a facet with an outward facing normal
            Print #OutFNum, HiddenEdgeFlag(pointer, 12) * cPt(1)
            Print #OutFNum, 72
            Print #OutFNum, HiddenEdgeFlag(pointer, 23) * cPt(2)
            Print #OutFNum, 73
```

```
                Print #OutFNum, HiddenEdgeFlag(pointer, 34) * cPt(3)
                Print #OutFNum, 74
                Print #OutFNum, HiddenEdgeFlag(pointer, 41) * cPt(4)
            Else
                MeshVertexPrint nullPt, 128, layerName                  Means vertex is the face of a polyface mesh
                Print #OutFNum, 71                                      This creates a face with an outward facing normal
                Print #OutFNum, HiddenEdgeFlag(pointer, 12) * cPt(3)
                Print #OutFNum, 72
                Print #OutFNum, HiddenEdgeFlag(pointer, 23) * cPt(2)
                Print #OutFNum, 73
                Print #OutFNum, -1 * cPt(1)                             Suppress the null line
                Print #OutFNum, 74
                Print #OutFNum, -1 * cPt(1)                             Suppress diagonal line

                MeshVertexPrint nullPt, 128, layerName                  Means vertex is the face of a polyface mesh
                Print #OutFNum, 71
                Print #OutFNum, -1 * cPt(3)                             Suppress diagonal line
                Print #OutFNum, 72
                Print #OutFNum, HiddenEdgeFlag(pointer, 23) * cPt(1)
                Print #OutFNum, 73
                Print #OutFNum, -1 * cPt(4)                             Suppress the null line
                Print #OutFNum, 74
                Print #OutFNum, HiddenEdgeFlag(pointer, 41) * cPt(4)
            End If
        Else                                                            This is a square mesh
            MeshVertexPrint nullPt, 128, layerName                      Means vertex is the face of a polyface mesh
            Print #OutFNum, 71                                          This creates a facet with an outward facing normal
            Print #OutFNum, HiddenEdgeFlag(pointer, 12) * cPt(1)
            Print #OutFNum, 72
            Print #OutFNum, HiddenEdgeFlag(pointer, 23) * cPt(2)
            Print #OutFNum, 73
            Print #OutFNum, HiddenEdgeFlag(pointer, 34) * cPt(3)
            Print #OutFNum, 74
            Print #OutFNum, HiddenEdgeFlag(pointer, 41) * cPt(4)
        End If

        PatchQTable.MoveNext

    Loop

    End Sub
```

Sub FileHeader

The contents of this file are the minimum required entries to begin a valid .DXF file entry

```
Print #OutFNum, 0
Print #OutFNum, "SECTION"
Print #OutFNum, 2
Print #OutFNum, "ENTITIES"

End Sub
```

Sub MeshVertexPrint

(pt0 As Point3DDouble, Code70 As Integer, layerName As String)

This routine takes the vertex coordinate information from the MeshVertexOutput and MeshPatchOutput routines and appends it to the .DXF file

```
Print #OutFNum, 0
Print #OutFNum, "VERTEX"
Print #OutFNum, 8
Print #OutFNum, layerName
Print #OutFNum, 10
Print #OutFNum, pt0.x
Print #OutFNum, 20
Print #OutFNum, pt0.y
Print #OutFNum, 30
Print #OutFNum, pt0.z
Print #OutFNum, 70                                    Vertex flag
Print #OutFNum, Code70

End Sub
```

Function HiddenEdgeFlag

(pointer As Long, side As Integer) As Integer

The function steps through the HETAble and identifies cases where a hidden edge should be included in the .DXF files. Hidden edges in polymeshes are denoted by negative signs on particular corner vertex pointers.

The function returns an integer value of 1 or -1 depending on its determination of the visibility of a side.

```
HETable.Index = "PrimaryKey"
HETable.Seek "=", pointer

If side = 12 And HETable.Fields("Edge1") Then
    HiddenEdgeFlag = True
ElseIf side = 23 And HETable.Fields("Edge2") Then
    HiddenEdgeFlag = True
ElseIf side = 34 And HETable.Fields("Edge3") Then
    HiddenEdgeFlag = True
ElseIf side = 41 And HETable.Fields("Edge4") Then
    HiddenEdgeFlag = True
Else
    HiddenEdgeFlag = 1
End If

End Function
```

Sub MeshVertexOutput

(flag As Integer, layerName As String)

This routine collects vertex point coordinates from the OutputQTable and calls the MeshVertexPrint routine to append them to the .DXF file

```
Dim pt0 As Point3DDouble

OutputQTable.Index = "New_Vertex_ID"

OutputQTable.MoveFirst
Do Until OutputQTable.EOF

    pt0.x = OutputQTable.Fields("X")
    pt0.y = OutputQTable.Fields("Y")
    pt0.z = OutputQTable.Fields("Z")
```

Assign the vertex coordinates to pt0

```
    If flag > 2 Then pt0.y = -1 * pt0.y
```

If flag indicates mirror then mirror transversely by changing the sign of the y coordinate of each value.

```
    MeshVertexPrint pt0, 128, layerName
```

Call the MeshVertexPrint routine to append the point information to the .DXF file

```
    OutputQTable.MoveNext
Loop

End Sub
```

Sub PrepareOutputFile

This routine kills any file which currently exists which has the same name as the OutFName. The error handlers ensure smooth execution of the program in the instance that an error is created by the non-existence of a file named OutFName.

Once any existing files have been eliminated, the routine opens the OutFName file and assigns it an OutFNum - a value globally used throughout this routine.

```
On Error Resume Next
Kill OutFName
On Error GoTo 0

OutFNum = FreeFile
```

Assign the input file number to the next available file number

```
Open OutFName For Output As OutFNum
```

Open the Input File

```
End Sub
```

A3

Appendix 3: Constructing Adjacent Sides Example

The algorithm described in Chapter 5 is not nearly as straightforward as the material presented in Chapters 3 and 4. The construction of the facets adjacent to the projected POI is a complex problem where the validity of the new mesh is necessary. The following example illustrates the steps and decisions of the algorithm. The example is relatively complex so as to prove the ability of the algorithm, but in the majority of ship problems, situations as complex as the one shown in this example are highly unlikely.

The first figure of this section shows the surface which is to be fitted against an irregular adjoining surface. In this example, it is assumed that the *Vertex List* has already been updated and sorted. Terminology particular to the example includes the terms *Anchor* and *Kedge* which refer to the beginning and end of the *Vertex List* respectively. The *Vertex List* is a list of vertices which includes all the vertices which lie on the current plane.

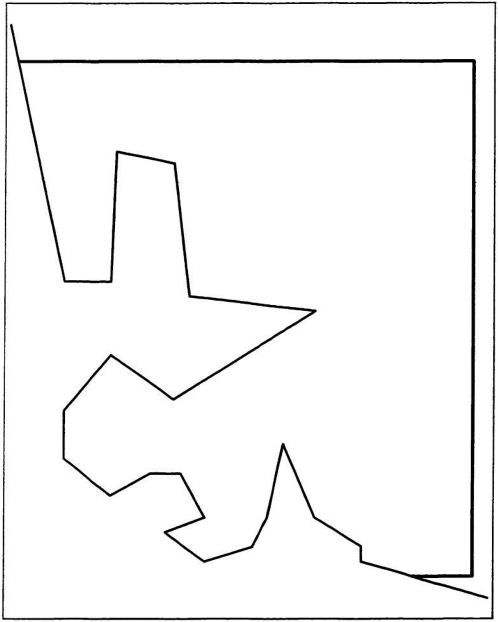


Figure 97 Example Problem. Assumes that a *Vertex List* for this surface has already been created and sorted.

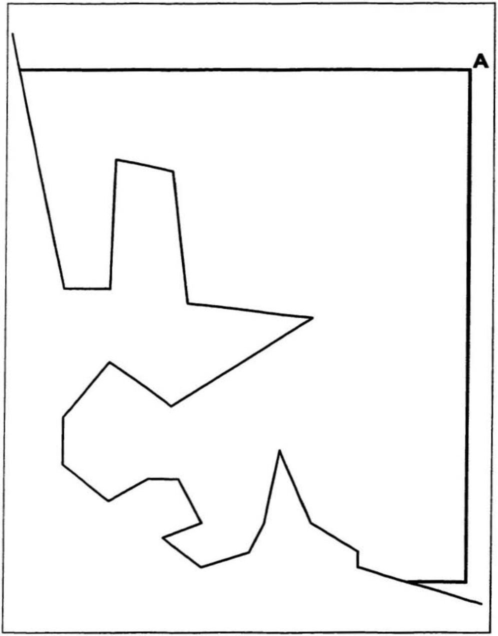


Figure 98 Set the first *Anchor* vertex, *Vertex A.*

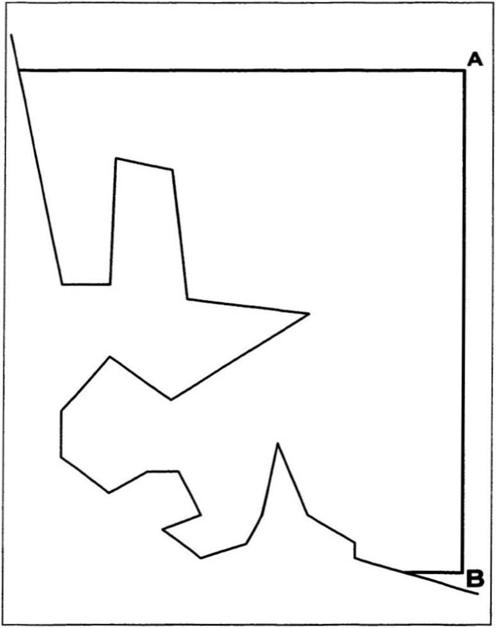


Figure 99 Switch sides. Set second anchor vertex, or *Kedge*, at *Vertex B*.

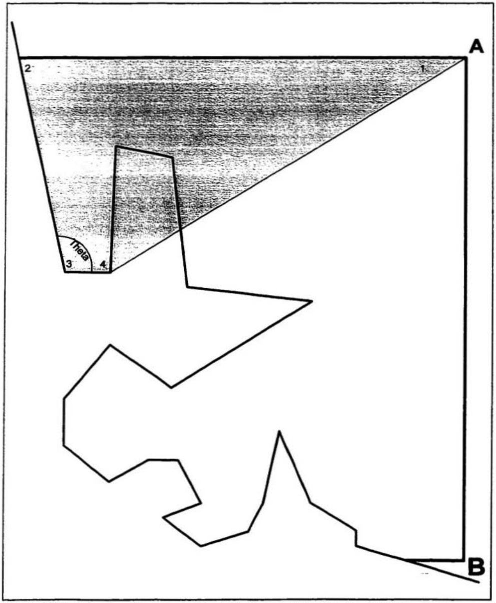


Figure 100 Switch sides. Since the angles at *Vertices 2* and *3* are less than 180 degrees, the algorithm attempts to create a four-sided patch using the first four vertices in the *Vertex List*.

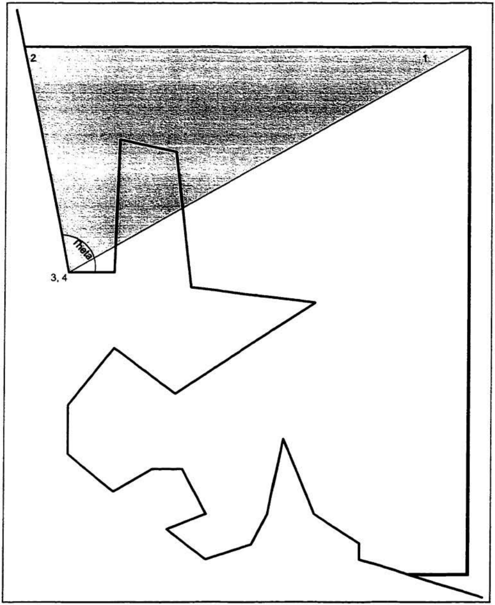


Figure 101 The algorithm, having checked and found an interference, attempts to remedy the problem by changing the new patch from one with four sides to one with only three sides.

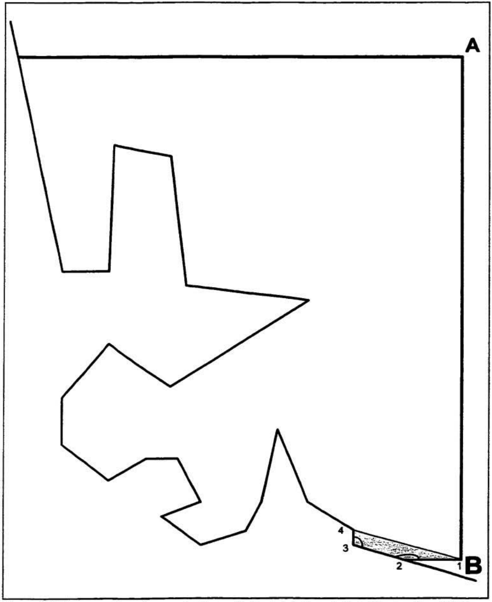


Figure 102 Because of interference the three-sided patch is discarded and the need to shift the *Anchor* vertex from *Vertex A* is noted. Switching sides, the algorithm attempts to construct a new patch.

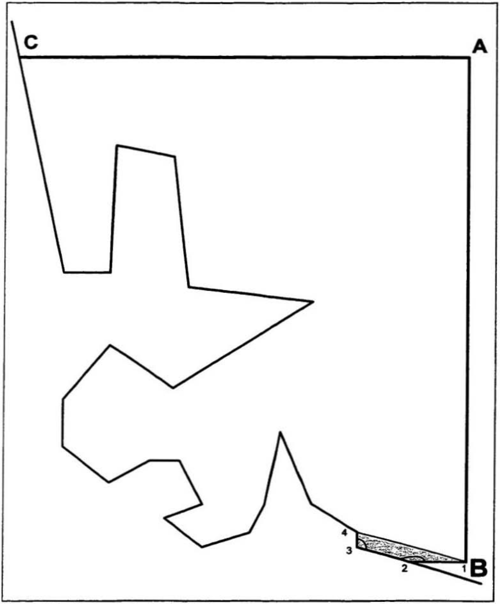


Figure 103 With this patch completed, *Vertices 2* and *3* are removed from the *Vertex List*, and the vertex angles recalculated. It then switches sides to shift the *Anchor* vertex from *A* to *C*.

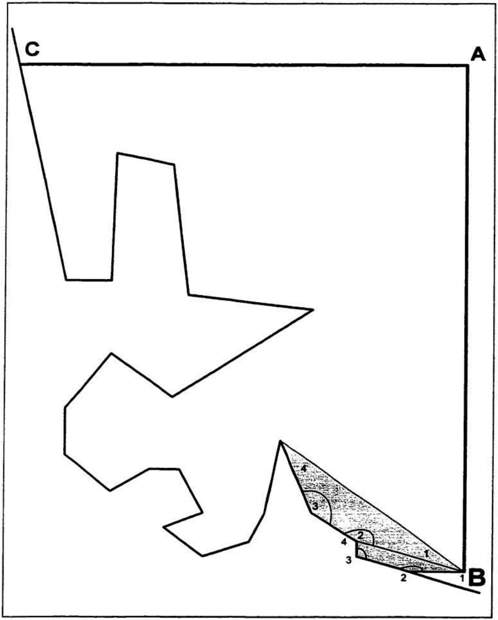


Figure 104 Returning to *Kedge B*, the algorithm successfully builds another patch. The *Vertex List* treats *Vertices 1* and *4* of the previous patch as *1* and *2* of the new patch.

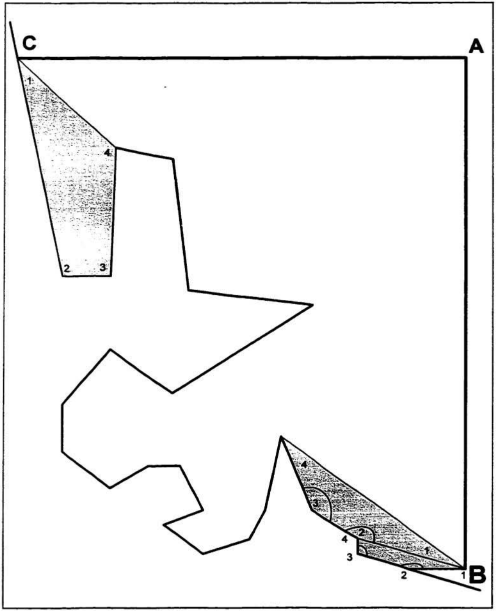


Figure 105 Having removed the 'trapped' vertices and switching sides, the algorithm now successfully constructs a patch from *Anchor C*. It then removes its 'trapped' vertices from the *Vertex List*.

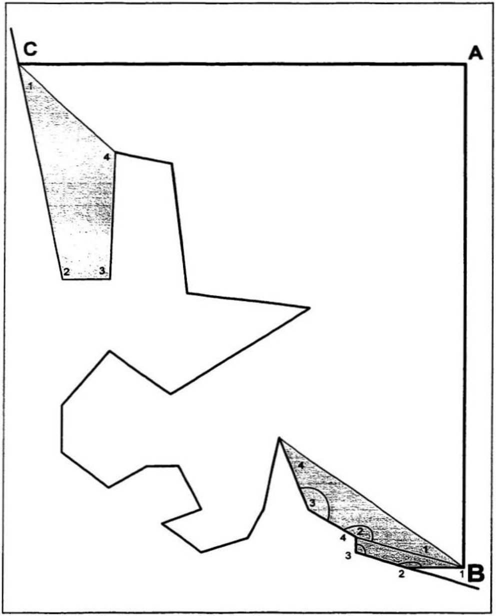


Figure 106 Although visibly unchanged, the algorithm has attempted and abandoned a new patch from *Kedge B*. The large angle at the new *Vertex 2* forced the abandonment.

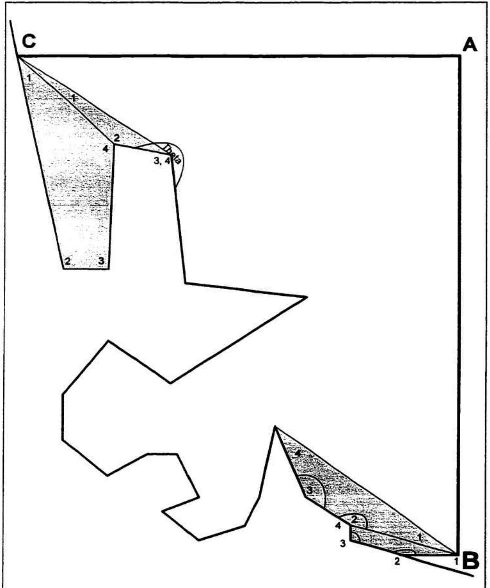


Figure 107 Switching sides once more, the algorithm constructs a second patch from *Anchor C*. The new patch has only three sides because of the large angle at *Vertex 3* of this new patch.

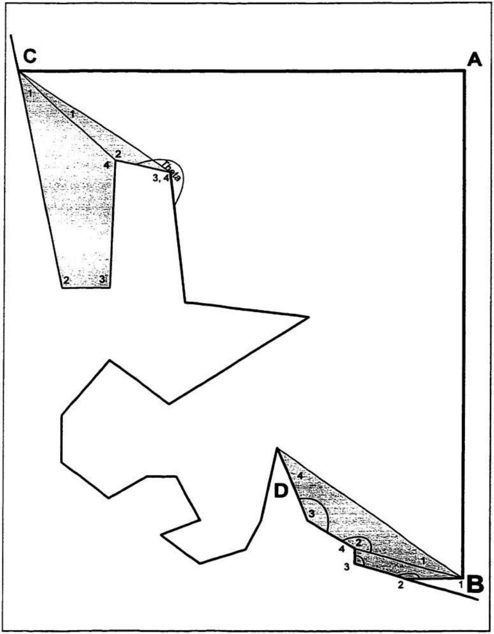


Figure 108 In this step, the algorithm switches sides and shifts the *Kedge* from *B* to *D*.

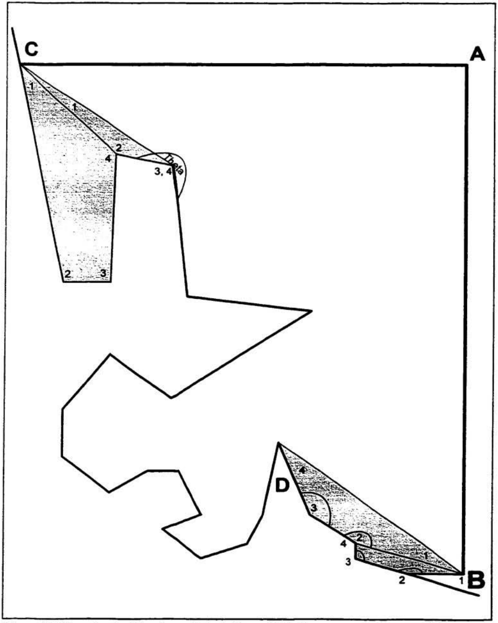


Figure 109 Here the algorithm has switched sides and failed to construct a new patch from *Anchor C* because of the large angle at *Vertex 3, 4.*

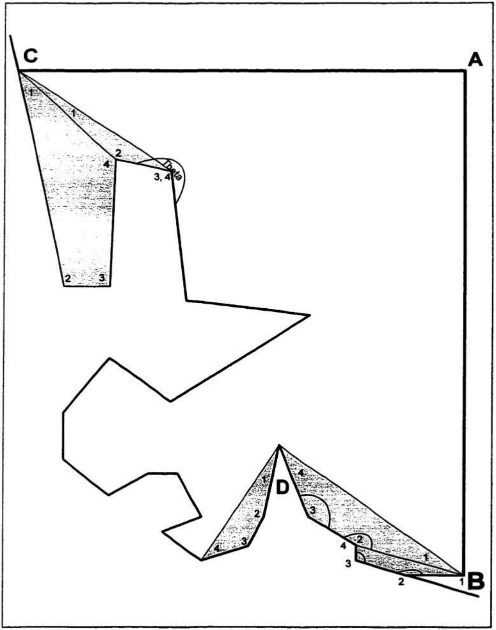


Figure 110 Switching sides, the algorithm successfully constructs a new patch from *Kedge D*.

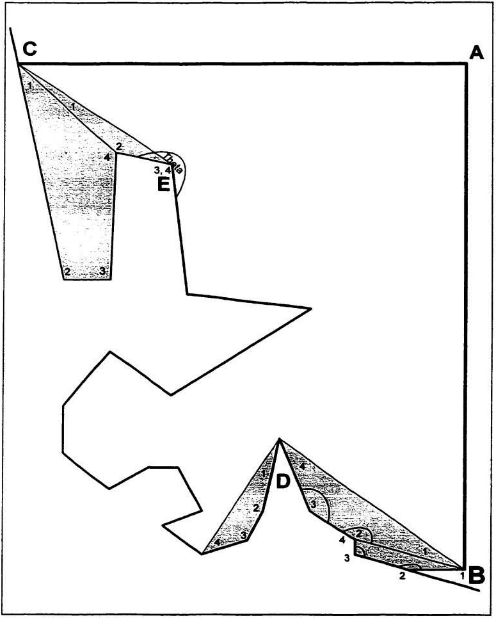


Figure 111 Here the algorithm has again switched sides, this time to shift the *Anchor* vertex from C to E.

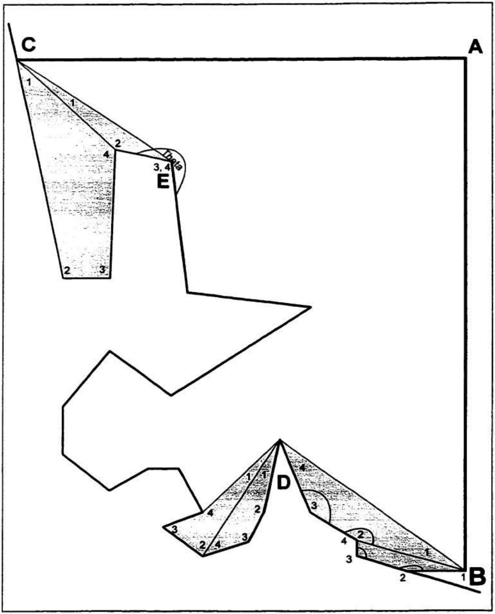


Figure 112 Switching sides, a second patch is created from *Kedge D*. The concavity at *Vertex 4* is caught through the calculation of angles in the same way that the *Vertex List* angles are calculated.

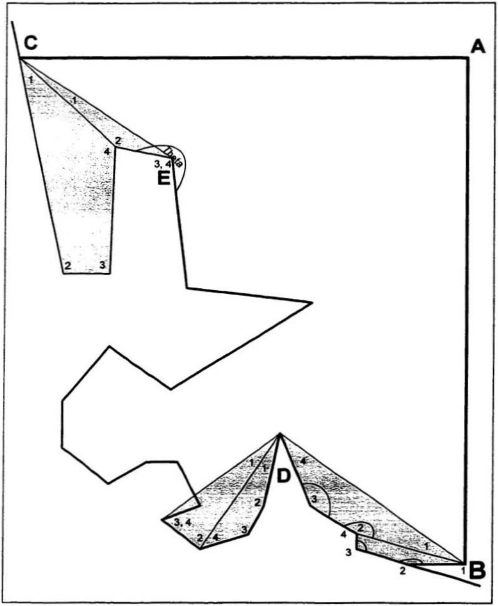


Figure 113 Because of the concavity error, a three-sided patch is attempted which leads to the invalid situation shown. The patch will be discarded and a note made to shift the *Kedge* from *D*.

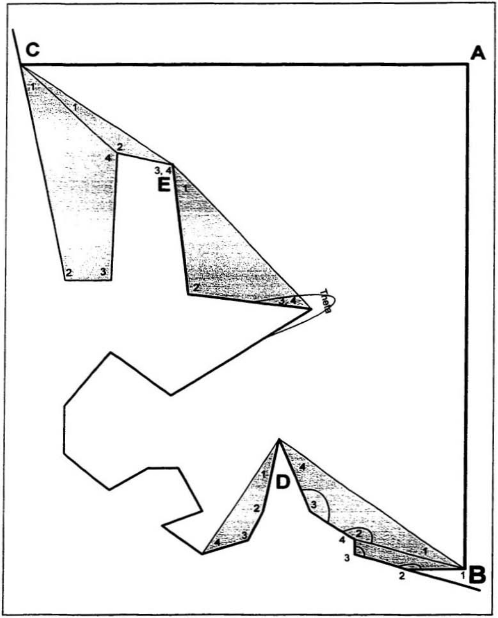


Figure 114 Switching sides, a three-sided patch is created from *Anchor E*. The large angle at *Vertex 3, 4* forced the creation of the three-sided patch.

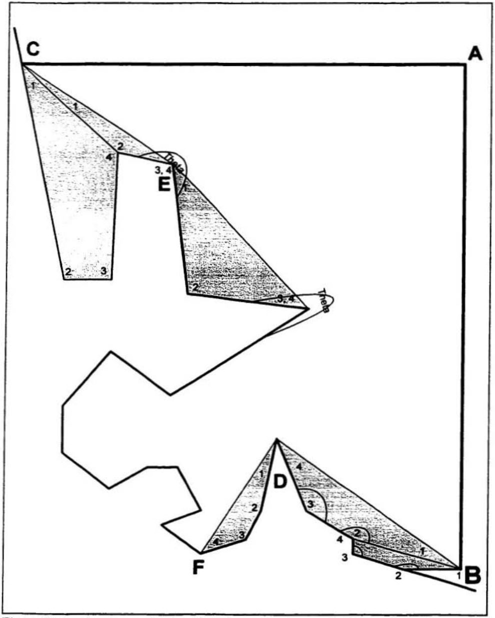


Figure 115 Another change of side, and another anchor change. In this step the *Kedge* vertex is shifted from *D* to *F*.

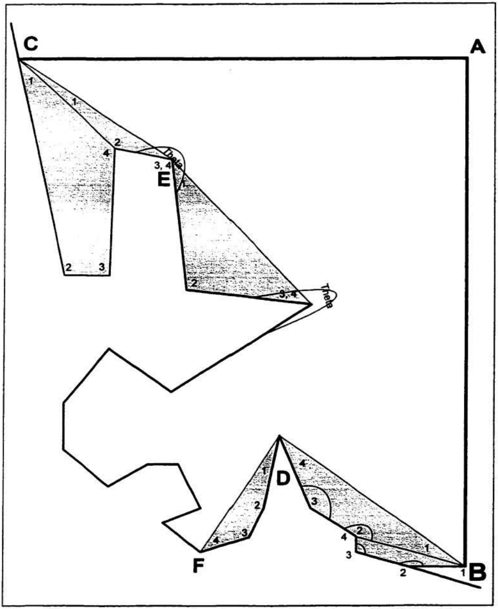


Figure 116 Here, a new patch was to be anchored on *E*, but the large angle at *Vertex 3,4* gives the new patch a concavity, forcing its abandonment. Instead, a note is made to change *Anchor E*.

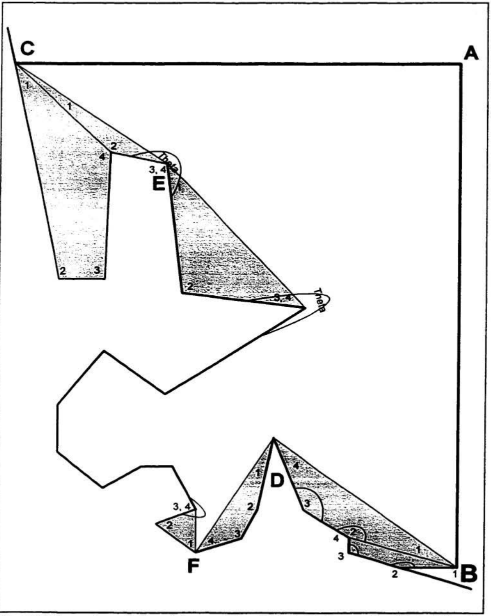


Figure 117 A new three-sided patch is created from *Kedge F*. The large angle at *Vertex 3* forced this patch configuration.

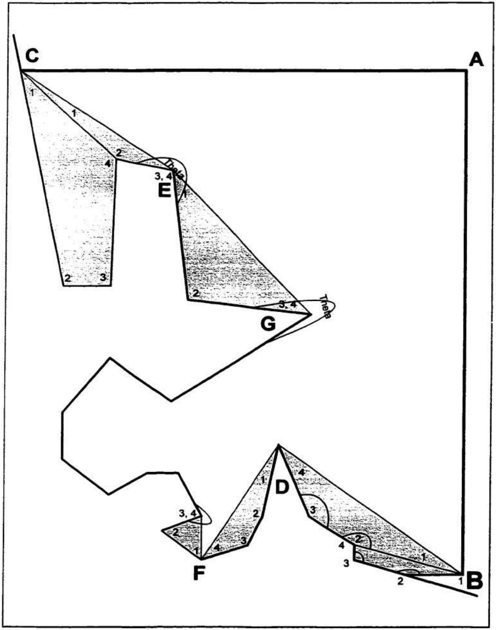


Figure 118 In this step the *Anchor* vertex is shifted from E to G.

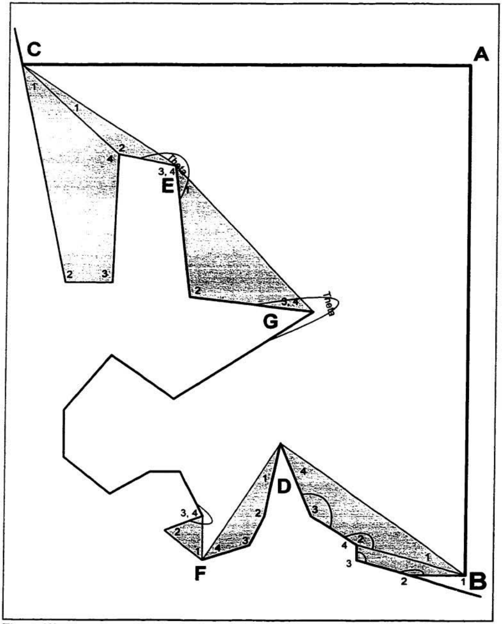


Figure 119 Here the algorithm attempts to build a new patch from *Kedge F* but fails because of the exterior angle found at the second vertex. Instead it flags *Kedge F* for change.

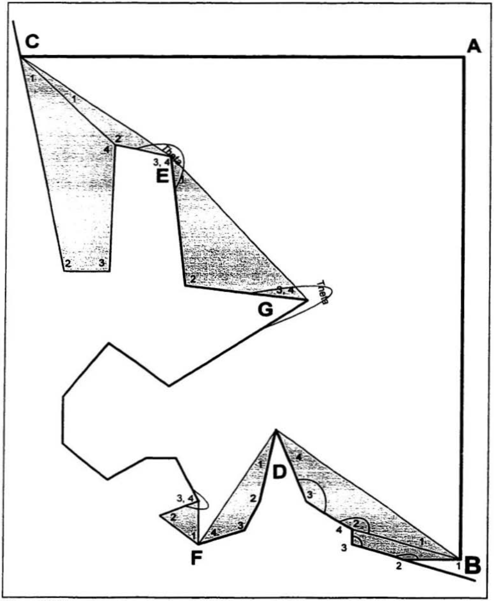


Figure 120 Switching sides, the algorithm attempts to build a new patch from *Anchor G* but fails because of the exterior angle found at the second vertex. Instead it flags *Anchor G* for change.

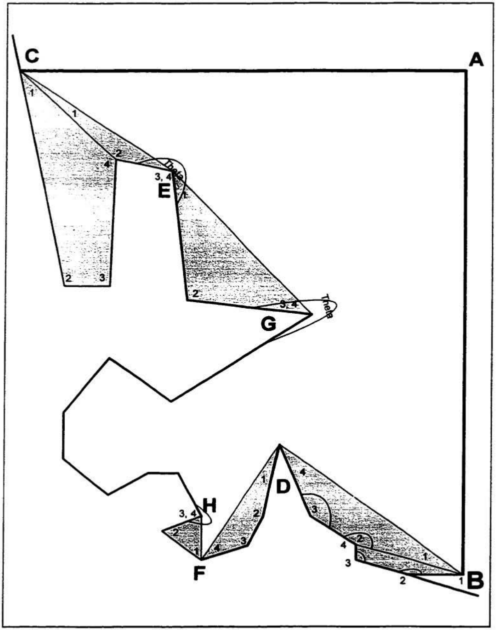


Figure 121 In this step the algorithm moves the *Kedge* vertex from *F* to *H*.

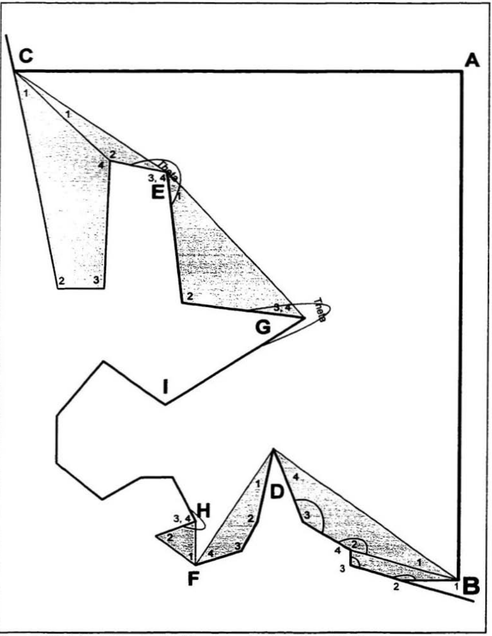


Figure 122 Switching sides, the algorithm moves the *Anchor* from *G* to *I*.

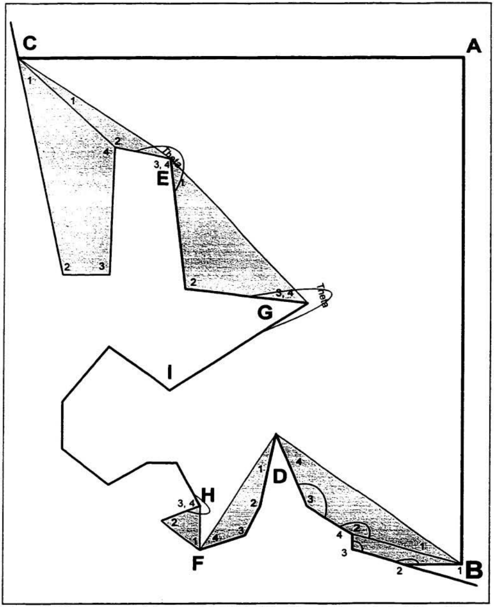


Figure 123 Here a new patch is attempted at *Kedge H*, but the exterior angle at what would be *Vertex 2* of the new patch forced its abandonment. Instead, a note is made to change *Kedge H*.

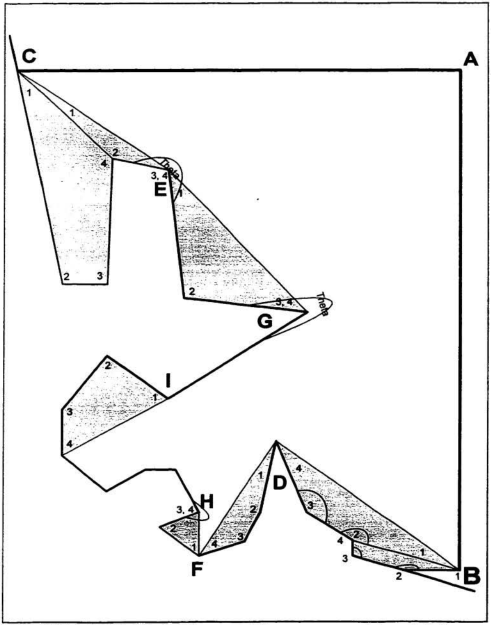


Figure 124 Switching sides, the algorithm successfully creates a new patch from *Anchor I.*

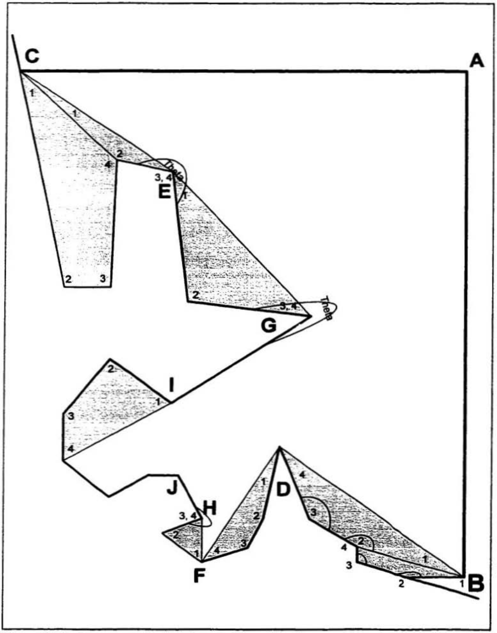


Figure 125 And once more the *Kedge* is moved from *H* to *J*.

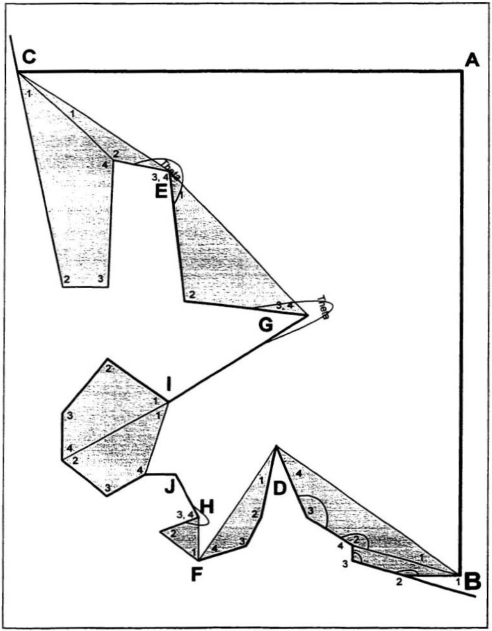


Figure 126 Switching sides, the algorithm successfully creates a second patch from *Anchor I*.

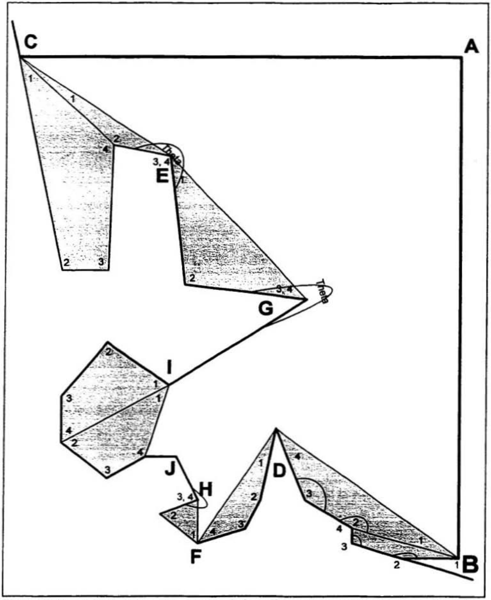


Figure 127 In attempting to create a new patch from *Kedge J*, the algorithm meets the forward leg of its search engine. Therefore instead of creating a new patch it begins the process again with the revised *Vertex List*.

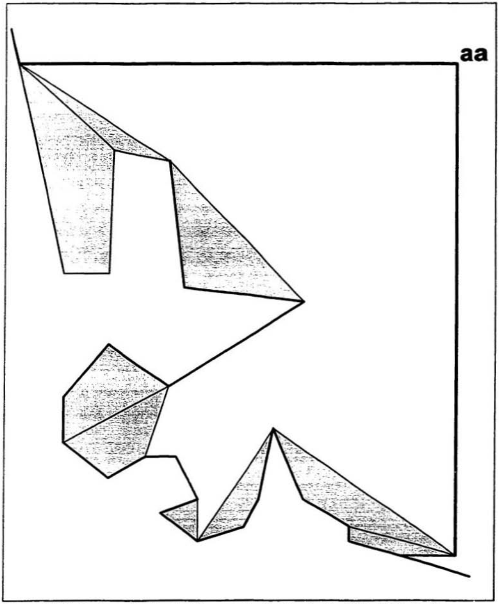


Figure 128 Beginning again, the algorithm sets the first item in the *Vertex List* to be the *Anchor aa.* Recall that vertex angles are updated to reflect the 'trapped' vertices of each of the new patches.

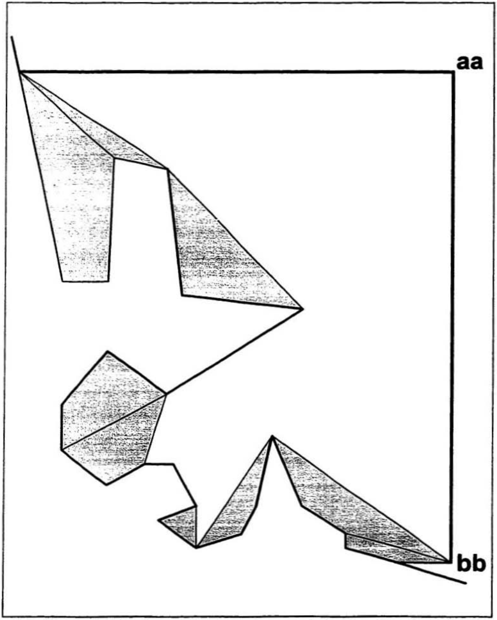


Figure 129 Switching sides the algorithm sets the last vertex in the *Vertex List* to be the *Kedge* vertex *bb*.

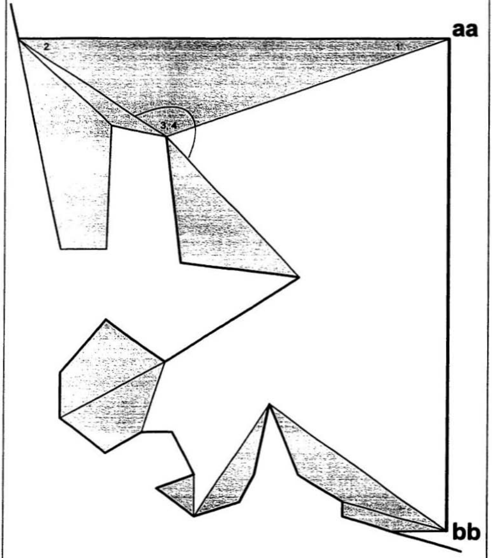


Figure 130 Returning to the *Anchor aa*, the algorithm creates a new patch. The patch is limited to three sides because of a potential concavity at *Vertex 3*.

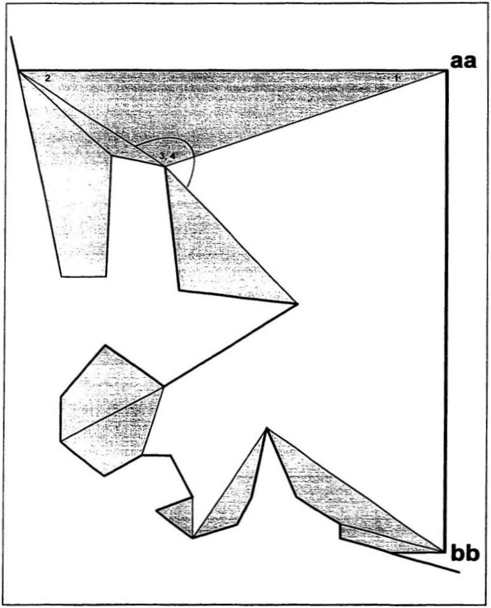


Figure 131 Jumping to *Kedge bb*, the algorithm unsuccessfully attempts to create a new patch, failing because of the exterior angle at what would be *Vertex 2* of the new patch.

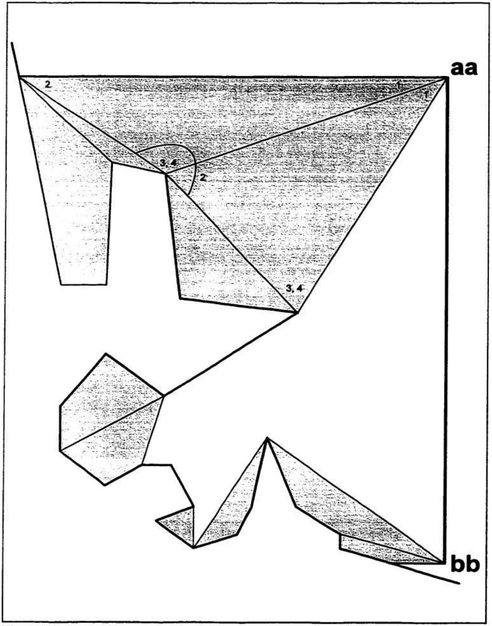


Figure 132 The algorithm now successfully creates a second triangular patch from *Anchor aa.*

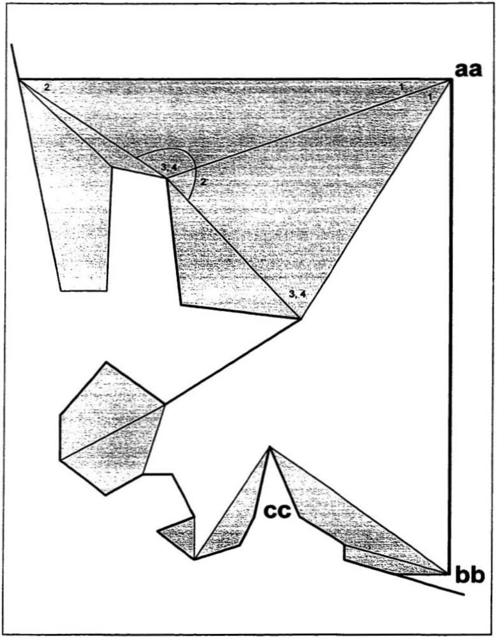


Figure 133 Switching ends, the algorithm now moves the *Kedge* from *bb* to *cc*.

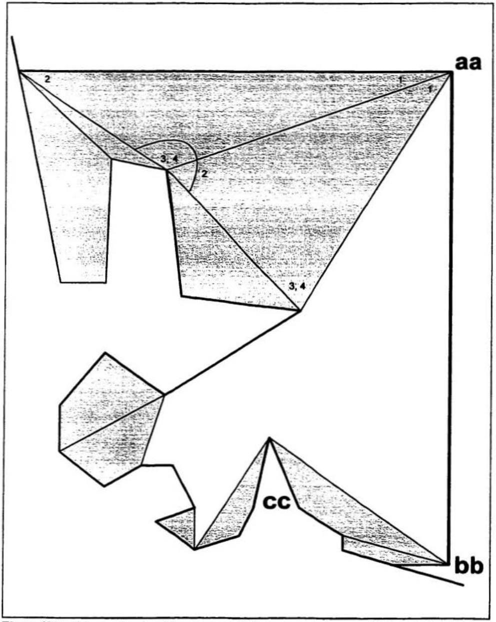


Figure 134 In this step the algorithm unsuccessfully attempts to create a third patch from the *Anchor aa*. Instead, it notes that the *Anchor* must be moved in order to continue.

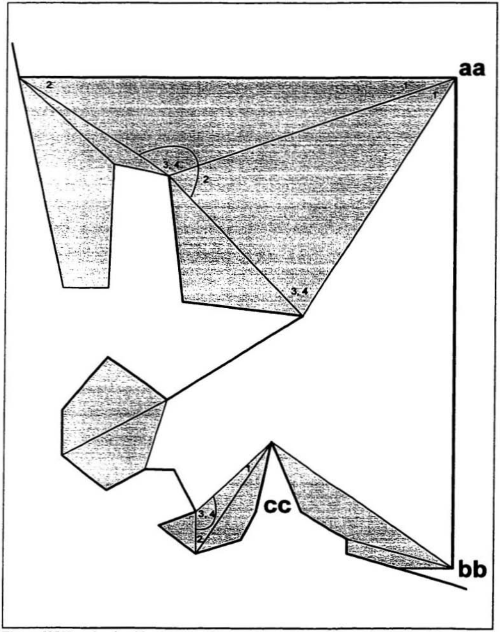


Figure 135 Here the algorithm builds a three-sided patch from *Kedge cc*.

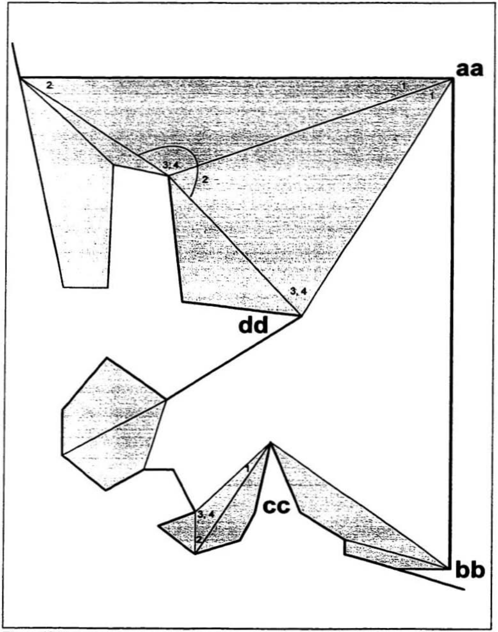


Figure 136 Switching sides again, the algorithm now shifts the *Anchor* from *aa* to *dd*.

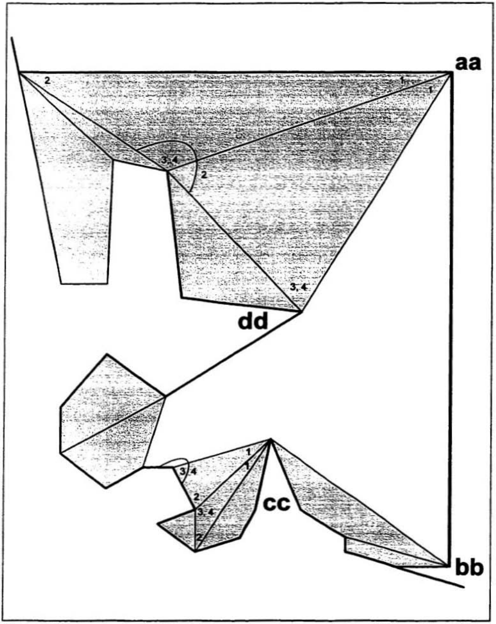


Figure 137 In this step the algorithm successfully creates a second three-sided patch from *Kedge cc*.

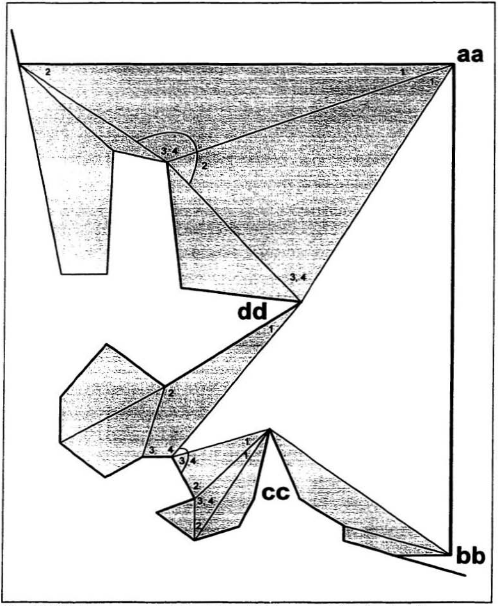


Figure 138 Having once more had the *Anchor* and *Kedge* meet such that there is no longer a sufficient number of vertices between the two to form a patch, the algorithm resets the anchor vertices and begins again.

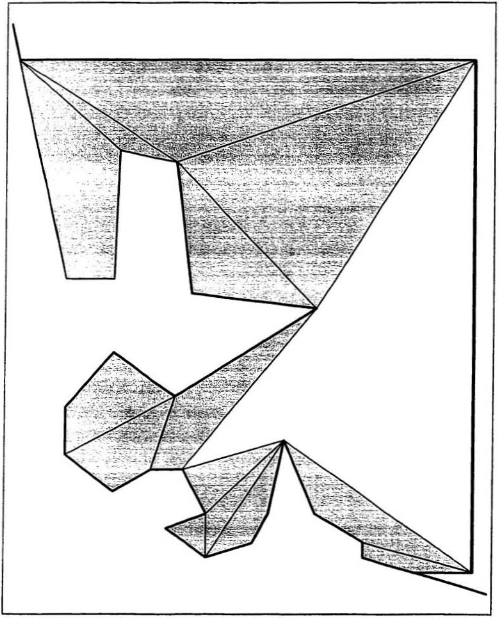


Figure 139 As can be seen, each iteration of the algorithm reduces the number of vertices to be placed into patches until no more are required.

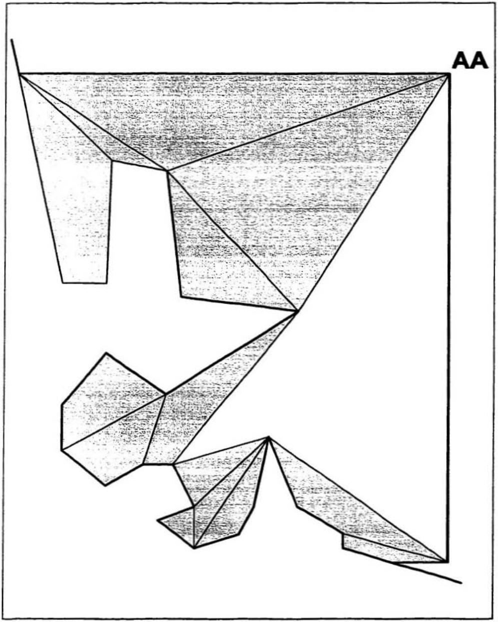


Figure 140 Once more the algorithm sets the *Anchor*, this time *AA* in the figure, to the first item in the *Vertex List*.

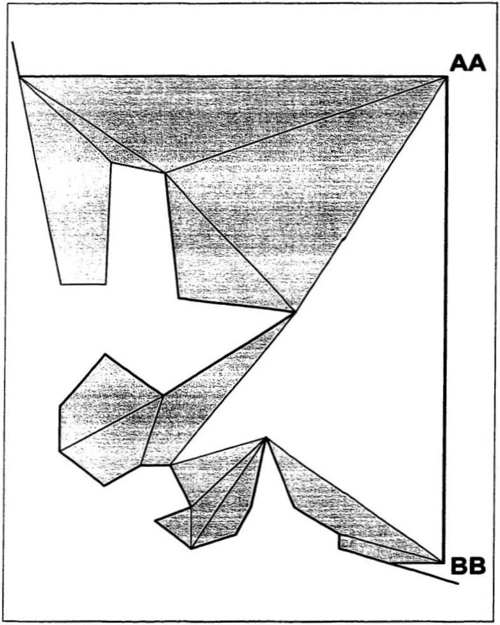


Figure 141 Switching ends, the algorithm then sets the *Kedge BB* equal to the last vertex in the *Vertex List*.

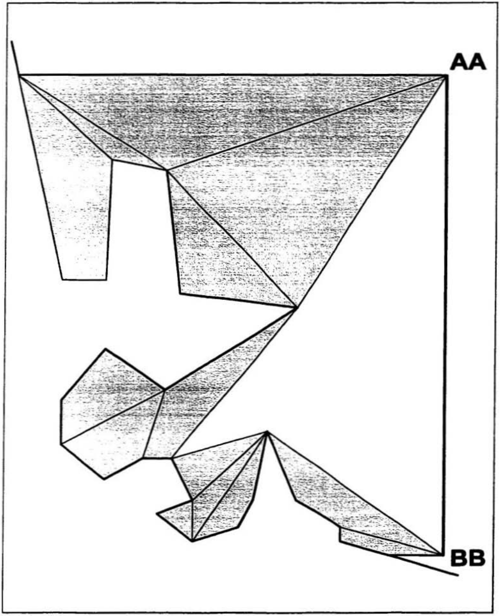


Figure 142 In this step the algorithm unsuccessfully attempts to create a new patch from the *Anchor AA*. The failure is due to the exterior angle at the next vertex in the list.

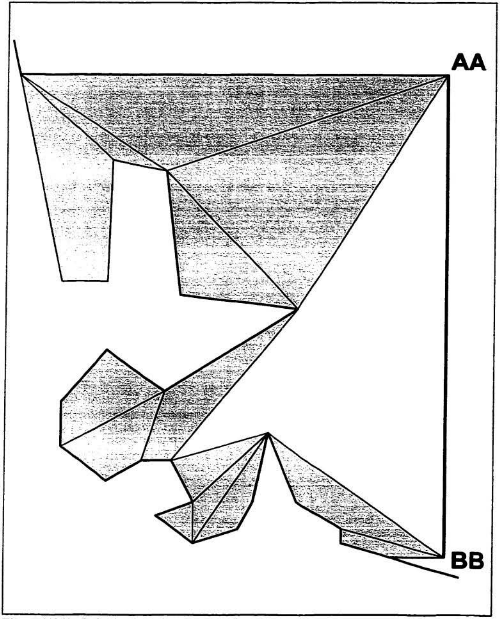


Figure 143 Similarly, the algorithm unsuccessfully attempts to create a new patch from the *Kedge BB*. Instead, the need to change the anchor is noted.

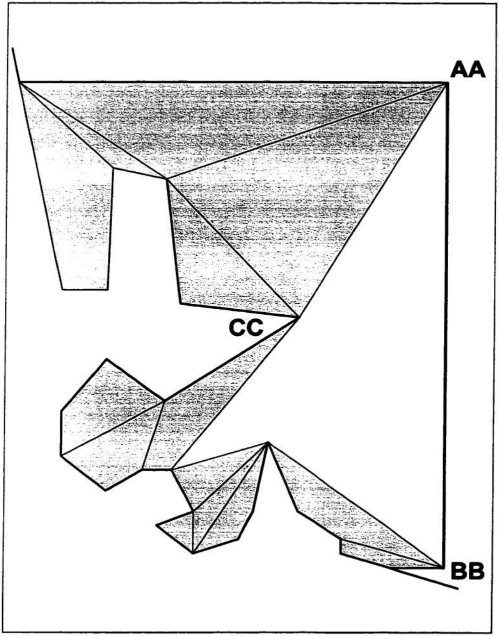


Figure 144 In this step the *Anchor* is moved to *CC*.

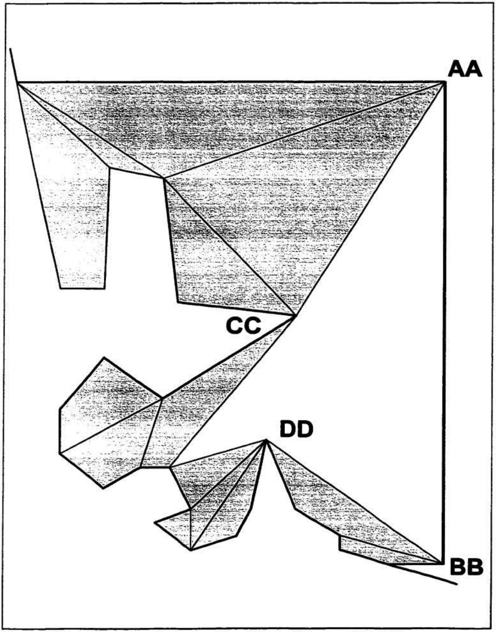


Figure 145 Switching ends again, the algorithm shifts the *Kedge* from *BB* to *DD*.

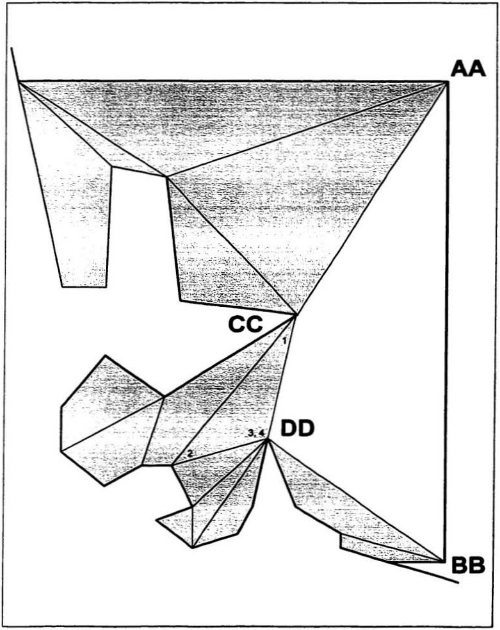


Figure 146 In this step a new three-sided patch is created from *Anchor CC*.

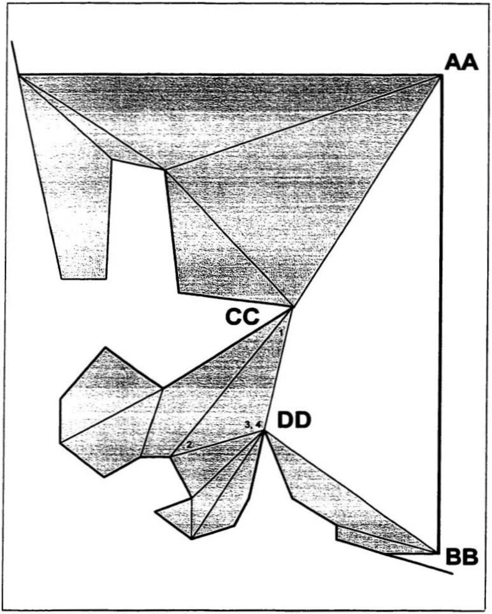


Figure 147 The completion of the new patch also brings the two ends of the list together again. Hence the algorithm resets for the last time.

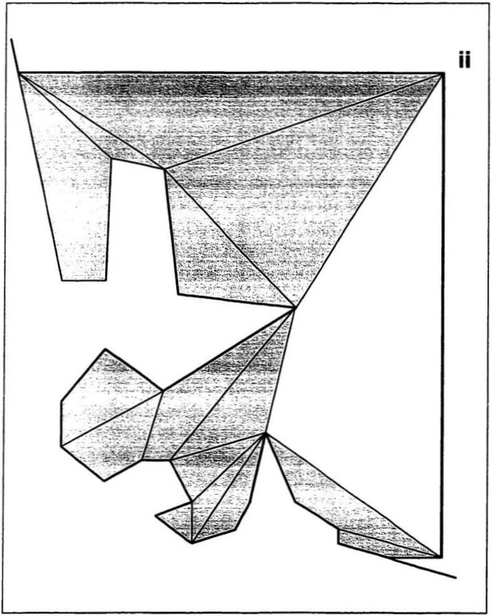


Figure 148 Beginning again at the start of the *Vertex List*, the algorithm sets the first item to be the *Anchor ii.*

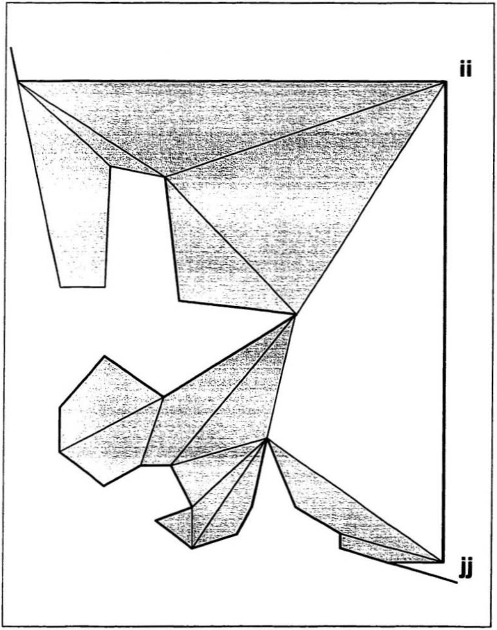


Figure 149 Switching ends, the algorithm also establishes a *Kedge* at *jj*.

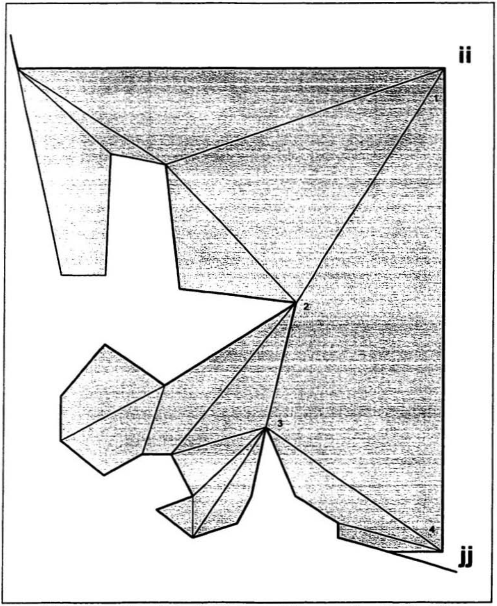


Figure 150 Switching ends again, the algorithm successfully creates a four-sided figure from *Anchor ii.* And with only two vertices remaining, the algorithm has also successfully completed the new mesh.