2 29
12

Intersection and Filleting of Non-Uniform B-Spline Surfaces

by

Robert W. Jones

thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Mechanical Engineering

APPROVED:

[illegible]

Dr. Arvid Myklebust, Chairman

Sankar Jayaram

Dr. S. Jayaram

Robert L West Jr

Dr. R. L. West

January 29, 1991

Blacksburg, Virginia

c.2

LD
5655
V855
1991
J[illegible]
c.2

Abstract

Preliminary aircraft design codes require a more complete and integrated geometry definition than that used by conceptual design codes. This thesis documents the design and creation of an interactive CAD system which converts the geometry descriptions commonly used in conceptual aircraft design codes to descriptions that meet the requirements of preliminary design systems. In particular, the conversion of ACSYNT Hermite surface data of aircraft models to the non-uniform bi-cubic B-Spline surface representation is addressed. The topics discussed in this thesis include the design and development of an interactive graphics user interface, the design and coding of an intersection method for non-uniform bi-cubic B-Spline surfaces utilizing subdivision techniques and the development of a one-dimensional filleting algorithm for blending surfaces along iso-parametric curves.

Acknowledgements

I would first like to thank my advisor Dr. Myklebust for giving me the opportunity to do research in the area of geometric modeling. I want to also express my sincere gratitude to Dr. Jayaram for his constant support and time devoted to helping me finish my thesis on time. Thanks are also extended to Dr. R. L. West for taking the time to be part of my graduate committee.

I want to thank the entire ACSYNT crew for putting up with me the last year and a half. Special thanks go to J. R. Gloudemans and Kris Kolady. Without them I would still be trying to log onto most of the computers in our lab.

Finally, I thank my Mother and Father for their constant love and support throughout my college career, one they sometimes thought would last forever.

Table of Contents

List of Illustrations

1.0 Introduction

Aircraft design is divided into the following three phases: conceptual, preliminary and final design. The geometry requirements of conceptual design codes are less demanding than those for preliminary design codes. Conceptual design codes, such as ACSYNT (AirCraft SYNThesis), use geometry descriptions which can be quickly displayed and easily modified [Wamp88b]. The sculptured surface representations are displayed as either wireframe or shaded image models. This geometry definition allows the designer to interactively change a design and easily visualize the effect on the screen. In most cases the surface definition need only guarantee tangent continuity between surface patches.

Preliminary design codes require a more complete and integrated geometry definition than that of conceptual design codes. The geometry definition in the preliminary design phase is used for detailed aerodynamic (CFD) and radar cross-section analysis. This type of analysis requires positional, gradient and curvature continuity (C^2 continuity) between surface patches. Additionally, intersecting components must be blended (filleted) to form a C^2 continuous surface. At present, automated procedures to convert the geometry from conceptual to preliminary design systems are not published in the literature.

1.1 Background

In the early '70's the conceptual design code called ACSYNT (AirCraft SYNThesis) was developed at the NASA Ames Research Center in Moffet Field, California [Greg73]. The goal of the ACSYNT project was to produce software to serve as an analysis tool that is flexible enough to handle a wide range of civil and military aircraft. The software is parameter based to allow designers to easily generate a design using general descriptions of an aircraft's wing, fuselage, nose, etc. ACSYNT was one of the first conceptual design codes robust enough to handle non-linear optimization.

In 1987 the CAD/CAM laboratory at Virginia Polytechnic Institute and State University (VPI&SU), under the direction of Dr. Arvid Myklebust, began work on a computer-aided design system for ACSYNT. The goal of the work at VPI&SU was to produce a highly interactive graphics interface and geometry data structure that would enhance the analysis and design capabilities of ACSYNT. This new enhancement utilized the 3D graphics standard PHIGS (Programmer's Hierarchical Interactive Graphics System) to produce visual design feedback while maintaining device independence [Wamp88a,Wamp88b]. The bi-cubic Hermite representation was used to describe component surfaces. This representation proved very successful for easy display and manipulation of wireframe and shaded images of the component surfaces.

The bi-cubic Hermite representation allows easy specification of both positional and tangent continuity between surface patches. However, the ACSYNT geometry description does not guarantee continuity between separate components. Conversion to one completely C^2 continuous model is a slow and tedious, manually-assisted task.

Previous research at VPI&SU created and proved the capability to form C^2 continuous fillets (blends) between any two intersecting bi-cubic B-Spline surfaces [Glou90]. This research was used to produce a C^2 continuous model of one aircraft using point data from an ACSYNT-based design. The objective was to verify these new concepts and methods. The process of converting the model was still based on manual input of data files.

1.2 Objectives

One of the objectives of the ACSYNT Institute at VPI&SU is to develop a comprehensive design system to convert conceptual aircraft design geometry to definitions which meet the requirements of preliminary design codes. The conversion procedure consists of three steps: defining the component geometry with C^2 continuous surfaces, computing the curves of intersection between all components and filleting (blending) the intersecting components to form one C^2 continuous surface description.

The objectives of this thesis work are divided into two main categories.

1. Design and develop a user interface and all necessary utilities to serve as a general platform to test new algorithms in surface and solid modeling. This interface will be used in the ACSYNT conceptual aircraft design code and will be referred to as the "ACSYNT B-Spline Module".

2. Create and code algorithms to convert point data for several intersecting components into one C^2 continuous surface definition.

The first goal of designing and coding the ACSYNT B-Spline Module has been shared by a fellow graduate student, Frederick W. Marcaly.

The surface description chosen is the non-uniform bi-cubic B-Spline surface. The cubic B-Spline definition guarantees positional, gradient and curvature continuity at patch boundaries. The C programming language and the 3D graphics standard, PHIGS, will be used to create the necessary software.

The process of forming a completely C^2 continuous model involves converting each component's geometry to the bi-cubic B-Spline definition, computing the curve of intersection between all intersecting components and finally, filleting (blending) all intersecting components to form one C^2 continuous surface. The conversion of each component to the bi-cubic B-Spline definition must allow the user to force tangent discontinuities between patches in special cases (e.g. wing end caps).

1.3 Thesis Organization

This thesis presents the various concepts and algorithms used to convert the Hermite point data from ACSYNT to a C^2 continuous bi-cubic B-Spline model. The overall design of the user interface and the programming formats are also discussed. The thesis is divided into the following sections:

1. Literature Survey
2. ACSYNT B-Spline Module
3. B-Splines
4. Intersection of B-Spline Surfaces
5. Filleting B-Spline Surfaces

This structure is used to allow different levels of readers to quickly locate areas of interest. The "B-Splines" section of this thesis gives a brief description of the B-Spline representation from a geometric modeling point of view as well as a detailed description of the steps necessary to convert point data to the non-uniform bi-cubic B-Spline representation. Readers intending to work with the B-Spline Module in the future should pay special attention to this section as well as the ACSYNT B-Spline Module section itself. A comprehensive manual documenting the B-Spline Module is included in Appendix A.

2.0 Literature Survey

2.1 ACSYNT B-Spline Module

Wampler [Wamp88a] published the first complete documentation of the ACSYNT interface in his thesis "Development of a CAD System for Automated Conceptual Design of Supersonic Aircraft". His thesis discusses the complete development of the graphical interface for the conceptual aircraft design code using the 3D graphics standard PHIGS. This thesis was followed by a paper which reviews the goals of the PHIGS-based interface and describes the bi-cubic Hermite surface representation chosen to visualize the aircraft geometry [Wamp88b].

Taylor's thesis titled "Specification of Mission Cycles for Aircraft Conceptual Design Using the PHIGS Standard" [Tayl88] was an extension of Wampler's thesis. This thesis discusses the use of PHIGS in creating an interactive graphical environment to allow users to specify aircraft mission cycles. A PhD. dissertation by Jayaram [Jaya89] investigated methods of developing machine independent tools to aid in the creation of

CAD/CAM software. Jayaram's dissertation provided ideas for producing tools that would create menus, receive input or provide feedback in a quick and efficient manner.

Grieshaber's thesis "Interactive Calculation of Cross-Sectional Areas for Aircraft Design and Analysis" [Grie88a] discusses the implementation of wave drag analysis into the ACSYNT design system. Grieshaber's research included displaying the aircraft in Hess format and restructuring the data to be suitable for CFD codes and the calculation of areas of intersection between the aircraft and a cutting plane. Grieshaber's work was later extended to include an automatic method of generating cross-sections and the development of area distribution plots along the aircraft axis [Grie88b].

Other ACSYNT related work included a thesis by Malon titled "Inlet Drag Prediction for Aircraft Conceptual Design" [Malo89]. Malon's thesis extended ACSYNT's inlet drag prediction capabilities. Malon developed methods to predict the inlet drag of subsonic and supersonic pitot inlets, fixed and translating spike conical inlets and two-dimensional supersonic inlets. Malon and Brown published a paper based on his thesis titled "Prediction of Inlet Drag for Aircraft Conceptual Design" [Mala90]. Gloudemans [Glou90] and Wong [Wong90] also completed thesis research related to ACSYNT. Their work will be discussed in the following sections.

2.2 *Intersection Techniques*

Most available literature for computing the intersection curves of free-form surfaces is based on subdivision techniques. At present, only recursive subdivision techniques have proven to be robust enough to be used in commercial CAD software. Cohen, et al. [Cohe80] were one of the first researchers to utilize subdivision techniques to manipulate and display B-Spline surfaces. Their research involved the utilization of subdivision techniques for the display, interference calculation, contouring, and rendering of non-uniform B-Spline surfaces. A second paper co-authored by Riesenfeld and Lane [Lane80] discusses the utilization of subdivision techniques for computing the intersection curve of two B-Spline surfaces. Their research also discusses the use of subdivision techniques for the display of both Bezier and B-Spline surfaces. Dokken [Dokk85] developed an algorithm which first converts several different complex free-form surface descriptions to the B-Spline representation and then utilizes subdivision techniques to compute the curve of intersection. After converting the surfaces, the type of intersection is categorized based on the degree of the curve or surface and the number of parametric directions used to define the curve or surface. A separate subdivision algorithm is then utilized to compute the curve of intersection.

In July of 1984 Peng [Peng84] developed a "Divide and Conquer" algorithm. This approach was an organized utilization of recursive subdivision techniques. Peng utilized the control polyhedron of the B-Spline surface instead of the actual surface points. The quad-tree organization was used to recursively subdivide two intersecting surfaces. In this manner the time to sort intersection data was greatly reduced. Two years later Lasser [Lass86] also developed a "Divide and Conquer" algorithm. Lasser's algorithm differed in that each planar intersection was computed using two triangles instead of the planar approximation. The triangle intersections resulted in a better approximation of the actual surface.

Chen and Ozsoy [Chen86] have discussed the use of a hunting algorithm that uses subdivision techniques to find the first point of intersection and then uses Newton's method to march along the curve of intersection. Their research includes a dedicated algorithm for predicting initial values for Newton's method to guarantee its convergence. Asteasu [Aste88] also describes an algorithm that uses the previous point to compute the next point of intersection. Asteasu's algorithm is successful for computing the intersection between any two algebraic surfaces.

At VPI&SU, Wong [Wong90] researched the feasibility of finding the intersection curve of two B-Spline surfaces analytically. His research was based on simplifying the B-Spline surface by approximating it with ruled surface subsets of B-Spline surfaces. Once the surface was simplified, elimination methods were utilized to obtain the polynomial which describes the intersection curve between two B-Spline surfaces. In one case this polynomial can be solved in closed form. For other cases, polynomials of degree 6,9,12 or 33, must be rooted.

2.3 Filleting of B-Spline Surfaces

Literature on filleting or manipulating the shape of B-Spline surfaces is sparse. Farin, et al. [Fari86] developed algorithms for the fairing or smoothing of B-Spline curves. The goal of their research was to modify the B-Spline curve to give a pleasant aesthetic appearance. Later work by Piegl [Pieg89] discusses ways of modifying the shape of rational B-Spline curves and surfaces. Piegl's algorithms are based on moving a control vertex or adjusting the weight associated with a particular control vertex to pull/push the curve toward/away from the control vertex. Warren [Warr89] researched methods of blending two algebraic surfaces into one smooth C^2 continuous surface. Warren's method was successful but this research was limited to a small set of algebraic surfaces such as cones and cylinders.

Gloudemans [Glou90] was one of the first to publish results of research in filleting (blending) two arbitrary free form surfaces. His work was based on the non-uniform bi-cubic B-Spline representation. Gloudemans used an intermediate surface to blend two intersecting surfaces. The excess on each intersecting surface was then removed to create one C^2 continuous surface. The intermediate surface matched the knot vector and control vertices at the point of blending on both intersecting surfaces.

3.0 ACSYNT B-Spline Module

The ACSYNT B-Spline Module was created from software designed to serve as a development platform for testing new algorithms in surface and solid modeling. The design and development of the ACSYNT B-Spline Module was shared between the author of this thesis and fellow graduate student, Fredrick W. Marcaly. The ACSYNT B-Spline Module will first be used in the creation of algorithms for converting geometry descriptions from conceptual design codes to those of preliminary design systems in particular, the conversion of bi-cubic Hermite geometry from ACSYNT-based designs to a completely C^2 continuous non-uniform bi-cubic B-Spline model.

This section presents a brief overview of the design and development of the ACSYNT B-Spline Module. Further details of the functionality and capabilities of this B-Spline Module are given in the ACSYNT B-Spline Module manual. For completeness, this manual has been included in Appendix A. Please note that this manual is updated whenever new modules are added. Therefore, the reader should refer to the most recent edition of the manual.

3.1 Design Considerations

There are three major considerations in the design of the ACSYNT B-Spline Module. The most important consideration is modularity. Since the ACSYNT B-Spline Module will serve as a platform for future work, it must be designed to allow new developers to easily understand and take advantage of the available functions. Furthermore, developers will need the ability to add new modules to the base platform once their research is complete. The second consideration is that the user interface and functionality be the same as those of ACSYNT. This is done to provide the ACSYNT user with easy transition between the two software systems. The last consideration is that the software must be portable to most UNIX workstations. For this reason, the software development will utilize the 3D graphics standard, PHIGS.

3.2 User Requirements

The list of user requirements was created prior to the development of the ACSYNT B-Spline Module. This list is a compilation of assumed needs of the end user of the B-Spline Module. As stated earlier, the initial purpose of the B-Spline Module was to convert ACSYNT geometry to the non-uniform bi-cubic B-Spline definition. Hence, some of the user requirements are based on the ACSYNT software. A list of the user requirements follows:

1. The interface must be identical to that of ACSYNT - i.e. the user should see no major differences in the visualization and functionality ("look" and "feel") of the software.

2. The user should be able to read geometry represented in the bi-cubic Hermite or the non-uniform bi-cubic B-Spline format.

3. The user should be able to write data files in either the bi-cubic Hermite or non-uniform bi-cubic B-Spline format.

4. The user should be able to change the attributes of the geometry represented in both the bi-cubic Hermite and non-uniform bi-cubic B-Spline definition (color, rendering, etc.).

5. All logical input devices should be utilized to provide the user with several methods of achieving the same goal.

6. Default values should be provided to help users who are unfamiliar with the system.

7. The user should be able to view the geometry in any of the four standard views (top, front, side and isometric).

8. The user should have the ability to manipulate the orientation of any of the above mentioned views in order to easily visualize various aspects of the model.

9. The user should be able to choose between a single view or multi-view windowing system.

10. The user should be able to shade the displayed geometry depending on the hardware shading capabilities of the workstation.

11. The user should be able to access the shading module (if one exists) at any time during program execution.

3.3 Functional Requirements

The functional requirements of the ACSYNT B-Spline Module refer to file handling, processing user input, geometry display and user feedback. These requirements also deal with the limitations of the software and the format for data storage and retrieval.

Functional Requirements

1. Software will be written using the 3D graphics standard, PHIGS, and the C programming language.

2. Utility routines for the display and manipulation of bi-cubic Hermite and non-uniform bi-cubic B-Spline surface representations should be available.

3. A linked list data structure will be utilized for the storage of component data in both the bi-cubic Hermite and non-uniform bi-cubic B-Spline representations (color, rendering, points per cross section, cross sections, etc.).

4. Utility routines should be used to get and process the input from all five logical input devices (Pick, Locator, String, Choice and Valuator).

5. Utility routines should be used for the display of menus and manipulation of a menu tree.

6. Utility routines should be used for displaying messages in the message area.

7. The interface should provide an easy method for rotating, translating and scaling three-dimensional geometry.

8. Output from the model data structure should be available in formats suitable for input to the IBM 6090 DAP DEMO and the Initial Graphics Exchange Specification (IGES).

9. The system should be able to read input data in the IBM 6090 DAP DEMO or IGES format.

3.4 *Software Description*

3.4.1 Screen Layout

The screen layout of the ACSYNT B-Spline Module is similar to the ACSYNT screen layout. The B-Spline Module screen is divided into the following sections:

- Software Title Area
- Regular Menu Input Area
- Standard Menu Input Area
- Message Scroll Area
- String Input Echo Area
- Geometry Display Area

Figure 1 on page 17 shows the location of each of these sections on the screen. The software title and standard menu are the only static images shown on the screen. The standard menu is a list of options that are available to the user at any time during execution of the software. The options can be selected via the pick, choice or string input devices. The regular menu is a dynamic menu on the right side of the screen. The regular menu display allows room for the title of the menu module and up to thirteen possible options for the user to choose from. Regular menu items can be selected using the pick device only. Utility routines exist for the display and manipulation of menus.

The message scroll area allocates space for up to five messages to give feedback to the user or prompt the user for input. Utility routines are available to allow the software developer to display the messages whenever needed. The string input echo area is used to display whatever the user types on the keyboard. Also, default input will appear in this area whenever string input is needed. Finally, the geometry display allows for the visualization of the design in one or four views. Again, utility routines are available for controlling the display area. These routines are available to both the code developer and the user during execution.

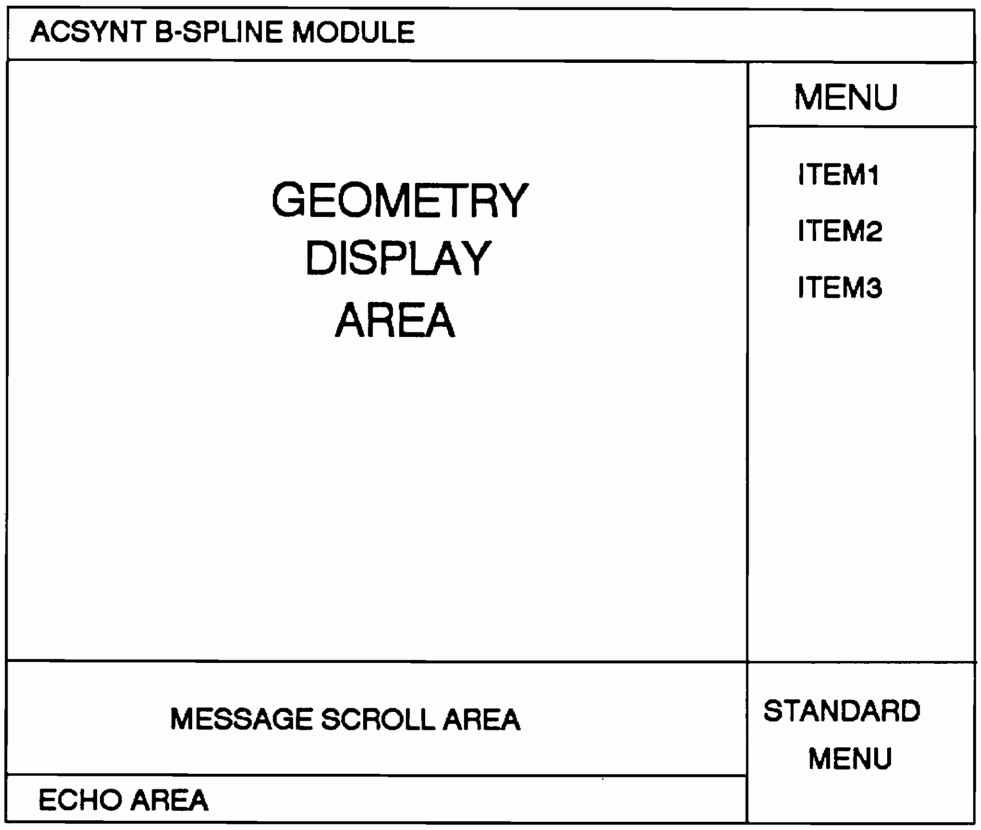


Figure 1. ACSYNT B-Spline Module Screen Layout

3.5 Data Storage

The B-Spline Module utilizes a linked list format using the C programming language for storing all component data in a centralized format [Koch89]. The component data structure contains all data needed to display a component in either the bi-cubic Hermite or non-uniform bi-cubic B-Spline formats. Additionally, the component color, number, name and rendering information are also stored in the component data structure. Refer to Figure 2 on page 19 for an example of the component data structure.

A separate model data structure is utilized to serve as the entry position to the component data. The model data structure contains the root structure identification numbers (id's) for all the possible types of geometry that can be displayed. The root id's are initially set to '-1' whenever a new data file is read. In this manner, the root id's can serve as flags for determining the types of available data. The model and component data structure relationships were set up to limit the passing of variables between modules to only the pointer to the model data structure. An example of the model data structure is shown in Figure 3 on page 20.

Component Data Structure

```
typedef struct compdata_type {
        int comp_number;              /* component number */
        char comp_name[20];           /* component name */
        int acs_id;                   /* structure id    */
        int nubs_id;
        int *hull_id;
        int fillet_id;
        int open[2];                  /* open flag 1 closed 0 open */
        int color;                    /* component color */
        int existence;                /* 1 exists 0 does not exist */
        int nu;                       /* rendering in u */
        int nw;                       /* rendering in w */
        int acs_ncross;               /* number of cross sections */
        int acs_npts;                 /* number of pts per xsection */
        float ***acs_pts;             /* pointer to component pts */
        float ***acs_utan;            /* pointer to tangents in u dir */
        float ***acs_wtan;            /* pointer to tangents in w dir */
        int nu_knots;                 /* number of u knots */
        int nw_knots;                 /* number of w knots */
        float *u_knot;                /* u knot array */
        float *w_knot;                /* w knot array */
        float ***hull;                /* control hull */
        struct compdata_type *next;   /* pointer to next component */
                                      }comp_data;
```

Figure 2. Example Component Data Structure

Model Data Structure

```
typedef struct {
        int num_comp;                          /* number of components in model */
        int acs_root;                          /* root structure id */
        int nubs_root;                         /* Non-Uniform B-Spline root id */
        int int_root;                          /* Structure id for intersection data */
        int fillet_root;
        comp_data *comp;                       /* pointer to beginning of linked list */
        struct intersection_type *intlist;     /* list of intersections */
                                               }MODEL;
```

Figure 3. Example Model Data Structure

4.0 Spline Curves and Surfaces

4.1 Introduction

The spline curve was first introduced as a drafting tool for the aircraft and shipbuilding industries. The spline was a strip of flexible material (plastic, wood, etc.) which could be flexed to pass through a series of design points drawn on a drafting board. Weights were positioned at the design points to force the spline to pass thru the required points. Once the spline curve was properly flexed, draftsmen would use the spline as a guide to draw a smooth curve passing through the specific design points [Mort85].

A key property of the spline is that it can pass through any number of points resulting in a gradual change in curvature without any kinks. This is accomplished as long as the spacing of the design points and the stiffness of the spline material are such that the spline is deformed within its elastic range. Thus, the spline behaves structurally as a beam under bending loads and the bending equations for a beam can be used to derive the equation for a plane cubic spline [Mort85].

Two popular curves are the Bezier and B-Spline curves. Both curve types are described with a set of control vertices that are approximated by the resulting curve. Together, the control vertices form the control hull or control polygon. The general shape of the Bezier or B-Spline curve is taken from the shape of the control polygon.

4.2 B-Spline Curves

In general, a B-Spline curve is an approximation curve in which the actual curve points approximate the control vertices but do not actually pass through them. However, it is possible to use the B-Spline representation to interpolate points as will be discussed in later sections. An example of the relationship between a control polygon and the corresponding B-Spline curve is shown in Figure 4 on page 23. Any point on the curve represents a weighted average of a specific number of control vertices. In the case of a cubic curve, any point is a weighted average of at most four surrounding control vertices.

The effect of a control vertex on a point on a curve is controlled by a set of blending functions. For a cubic curve, there will be at most four blending functions associated with each point on the curve. These blending functions can be determined in a few different ways which will be discussed in the following sections. At present it is sufficient to understand that the shape of a B-Spline curve is defined by a control polygon while the effect of each control vertex is determined by the blending functions. The following sections discuss the formulation of the relationships between the blending functions, control vertices and curve points.

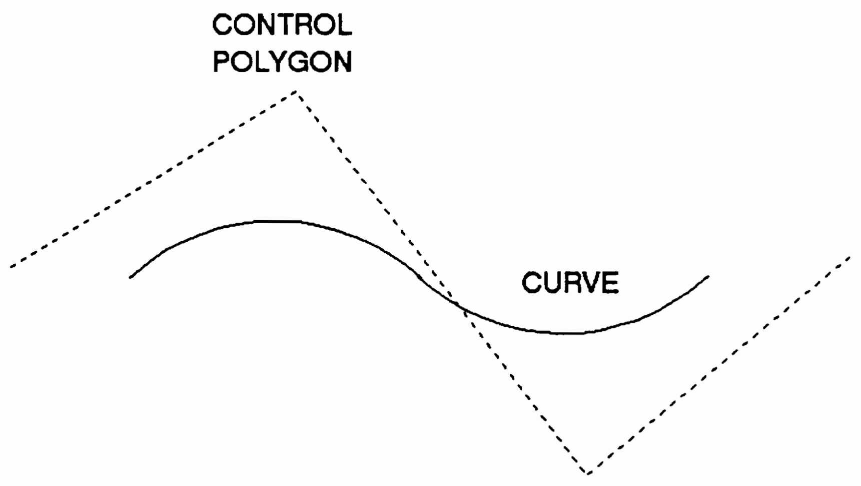


Figure 4. Relationship Between a B-Spline Curve and Control Polygon

4.2.1 B-Spline Curve Definition

The B-Spline curve definition has been well explained by a number of authors [Mort85], [Yama88], [Bart87] and [Glou90]. Yamaguchi's format for the display and rendering of B-Splines has been adopted for the research in this thesis. This section defines the equations for computing the points along a B-Spline curve. All equations given are valid for the cubic B-Spline representation. The following nomenclature will be used in this thesis.

B-Spline Nomenclature

k	degree of curve (cubic $k=3$)
M	order of curve ($k+1$, for a cubic curve $M = 4$)
p	point on B-Spline curve
q	control vertex
Q	control polygon or polyhedron
u, w, t	knot value along curve
T	knot vector for entire B-Spline curve
N	blending function

4.2.2 Uniform Cubic B-Spline Curve

The mathematical formulation for a B-Spline curve with $n+1$ control points is as follows:

$$p(u) = \sum_{i=0}^{n} q_i N_{i,M}(u)$$

This equation states that any point on the curve is a weighted average of the surrounding control vertices. The $N_{i,M}$ refers to the blending functions for an M^{th} order curve. An interesting characteristic of the B-Spline definition is the manner in which the blending functions are computed. All but k+1 blending functions will be zero for any point on the B-Spline curve. Hence, any point on the curve is affected by at most k+1 surrounding control vertices. This feature results in giving the B-Spline curve local shape control. For the cubic B-Spline curve (k = 3), any point is a weighted average of at most four surrounding control vertices.

The uniform B-Spline curve has the additional characteristic of having the same blending function coefficients throughout the entire curve. Thus, the equation for the uniform B-Spline curve can be simplified to the following format.

$$P_i(u) = UMQ_i$$

Where:

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \quad M = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \quad Q_i = \begin{bmatrix} q_{i-1} \\ q_i \\ q_{i+1} \\ q_{i+2} \end{bmatrix}$$

The "M" matrix contains the coefficients for all blending functions throughout the curve. This matrix is often referred to as the "Universal Transformation Matrix" for B-Spline curves and surfaces. The multiplication of the U and M matrices gives the blending functions for any point along the uniform B-Spline curve.

The B-Spline curve is a composite of curve segments. In the uniform case, each curve segment starts at a parametric value (u) of 0 and ends at a value of 1. All points within a cubic curve segment are affected by the same four control vertices. The blending functions for each point are varied according to the parametric value u. As the parameter u changes from 0 to 1 the blending functions will shift the effect of each control vertex.

It is important to note that when the parameter u is equal to 0, the curve segment is only affected by the first three control vertices. Likewise, when the parameter u is equal to 1, the curve segment is affected by the last three control vertices. These two limits of the parametric value represent the transition from one curve segment to another. This transition is defined as a curve segment breakpoint. The importance of this transition property will become more apparent in later discussions of the filleting algorithm.

4.2.3 Non-uniform Cubic B-Spline Curve

The non-uniform B-Spline curve allows more shape control than the uniform B-Spline curve. This additional control is accomplished through a knot sequence or knot vector. The knot sequence is a set of nondecreasing numerical values. The cubic B-Spline curves used in this thesis have one knot value for each control vertex with two additional knot values at each end of the control polygon [Yama88].

The knot spacing (difference between two subsequent knot values) controls the effect of a control vertex. Hence, a knot vector of (1, 3, 7, 9) has the same effect as (4, 6, 10, 12). Figure 5 on page 28 illustrates the relationships between the knot sequence, control polygon, and points along a non-uniform B-Spline curve.

The blending functions for the non-uniform B-Spline curve are determined through a sum of divided differences of the knot sequence. The following recursive formula can be used to calculate the blending functions:

$$N_{i,1}(u) \begin{array}{ll} = 1 & \textit{if } t_i \leq u < t_{i+1} \\ = 0 & \textit{otherwise} \end{array}$$

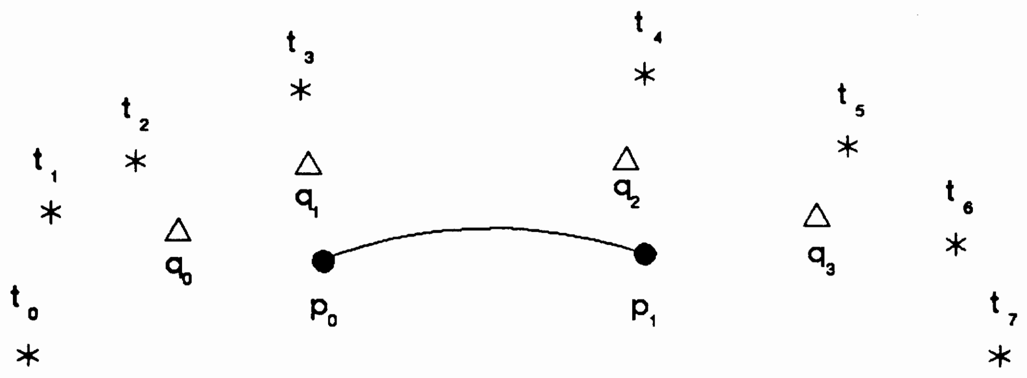
$$N_{i,M} = \frac{(u - t_i)\, N_{i,M-1}(u)}{t_{i+M-1} - t_i} + \frac{(t_{i+M} - (u))\, N_{i+1,M-1}(u)}{t_{i+M} - t_{i+1}}$$

Again, all but k+1 blending functions will go to zero allowing at most k+1 control vertices to affect any point along the B-Spline curve.

4.2.4 Important Properties of the B-Spline Representation

The following list reviews the important properties of the B-Spline representation which led to the selection of the B-Spline definition for this research [Yama88].

- C^{k-1} continuity between curve segments. i.e. a cubic B-Spline curve guarantees C^2 continuity at curve segment breakpoints.



● Number of break points = n (2)

△ Number of control vertices = n + 2 (4)

✳ Number of knots = n + 6 (8)

CORRESPONDENCE

$t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7$

$q_0 \quad q_1 \quad q_2 \quad q_3$

$p_0 \quad p_1$

Figure 5. Relationships of Points, Control Vertices and Knots

- Variation Diminishing Property: A B-Spline curve never intersects any arbitrary straight line more times than its control polygon does. Thus, the shape of a B-Spline curve is reflected by the shape of its control polygon.

- Local Shape Control: The effect of any control vertex is limited to k+1 curve segments. Any point along a cubic B-Spline curve is affected by at most four control vertices.

- Curve Degree Control: The degree of a B-Spline curve is independent of the number of control vertices. The degree of a Bezier curve is dependent upon the number of control vertices.

4.3 Bezier Curves

The B-Spline and Bezier curves are related in that the Bezier control polygon is related to the B-Spline control polygon. The development of the Bezier curve was published in 1972 at the Renault car company by P. Bezier [Mort85]. Bezier defined the following four properties prior to developing his spline representation:

1. The curve must interpolate the first and last control points.

2. The curve tangent at p_0 is defined by the first two control vertices Q_1 - Q_0. The curve tangent at the last segment p_n is defined by the last two control vertices Q_n - Q_{n-1}.

3. The second requirement is generalized for the higher order derivatives. For example the second derivative at p_0 is defined by the control vertices Q_0, Q_1 and Q_2. This feature results in unlimited control of the continuity between curve segments.

4. The curve functions must be symmetric with respect to the parametric value u and (1 - u). This feature allows for the reversal of the direction of parameterization.

Figure 6 on page 31 demonstrates the relationship between the control polygon and the Bezier curve. The importance of these relationships will become more apparent when the intersection algorithm is discussed.

4.4 B-Spline Surfaces

The formulation of the B-Spline surface is a direct extension of the B-Spline curve formulation. The B-Spline surface has the same distinct characteristics as the B-Spline curve. The B-Spline surface is defined in terms of its control polyhedron as shown in Figure 7 on page 32. The following equation represents a point on a general B-Spline surface:

$$p(u) = \sum_{i=0}^{n} \sum_{j=0}^{m} q_{ij} \, N_{i,M}(u) \, N_{j,L}(w)$$

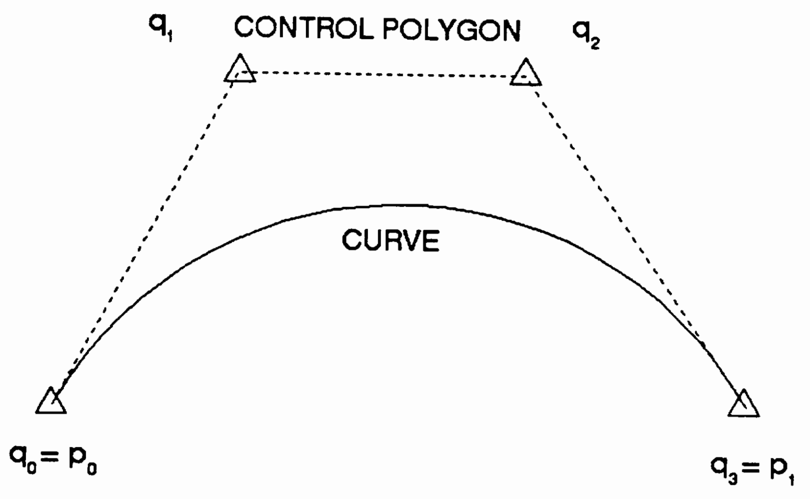


Figure 6. Bezier Control Polygon

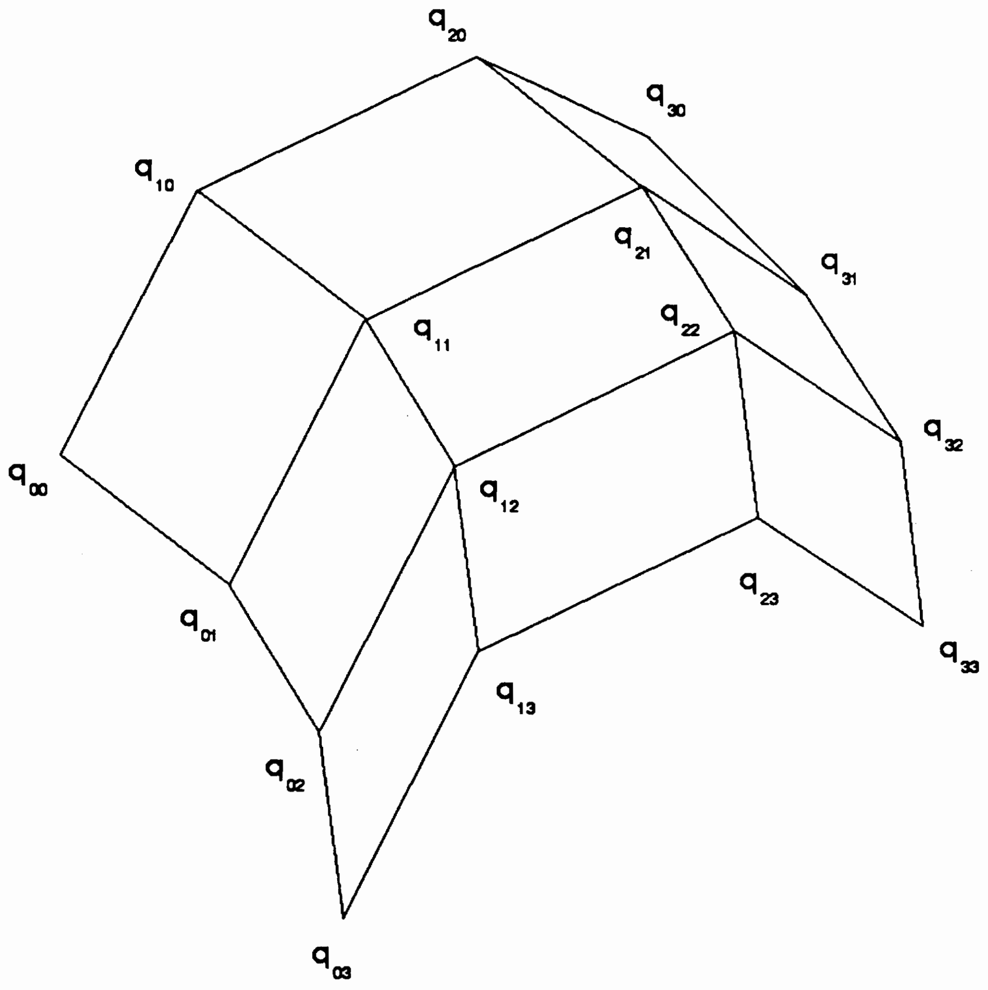


Figure 7. B-Spline Surface Control Polyhedron

This equation is similar to the B-Spline curve formulation with the addition of the blending function in another parametric direction. As was the case with B-Spline curves, the uniform B-Spline surface formulation can be simplified as follows:

$$p_{i,j}(u,w) \;=\; U\,M\,Q\,M^T\,W^T$$

where:

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \quad W = \begin{bmatrix} w^3 & w^2 & w & 1 \end{bmatrix}$$

$$M = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \qquad Q = \begin{bmatrix} q_{i-1,j-1} & q_{i-1,j} & q_{i-1,j+1} & q_{i-1,j+2} \\ q_{i,j-1} & q_{i,j} & q_{i,j+1} & q_{i,j+2} \\ q_{i+1,j-1} & q_{i+1,j} & q_{i+1,j+1} & q_{i+1,j+2} \\ q_{i+2,j-1} & q_{i+2,j} & q_{i+2,j+1} & q_{i+2,j+2} \end{bmatrix}$$

The blending functions are calculated recursively in the same manner as those in the B-Spline curve formulation. The non-uniform bi-cubic B-Spline representation is presented in matrix form below.

$$p_{i,j}(u,w) \;=\; \left[N_{0,4}(u)\,,\,N_{1,4}\,,\,N_{2,4}\,,\,N_{3,4}\right] \begin{bmatrix} q_{i-1,j-1} & q_{i-1,j} & q_{i-1,j+1} & q_{i-1,j+2} \\ q_{i,j-1} & q_{i,j} & q_{i,j+1} & q_{i,j+2} \\ q_{i+1,j-1} & q_{i+1,j} & q_{i+1,j+1} & q_{i+1,j+2} \\ q_{i+2,j-1} & q_{i+2,j} & q_{i+2,j+1} & q_{i+2,j+2} \end{bmatrix} \begin{bmatrix} N_{0,4}(w) \\ N_{1,4}(w) \\ N_{2,4}(w) \\ N_{3,4}(w) \end{bmatrix}$$

The variable N(u) represents the blending functions in one parametric direction while the variable N(w) represents the blending functions in the other parametric direction.

5.0 Knot Insertion

As stated earlier, the break point on a cubic B-Spline curve is a weighted average of three control vertices. The non-uniform B-Spline curve has a break point at every parametric value that is equal to a knot value in the knot vector ($u = t_i$). It is sometimes advantageous to add a knot value into the knot sequence and force a break point in the curve. These advantages will become apparent in the discussions of the intersection and filleting algorithms.

De Boor [DeBo72] was the first to publish an algorithm for inserting knots into B-Spline curves. This algorithm has been refined over the years by Boehm [Boeh80,Boeh85] and is described by Yamaguchi as follows [Yama88]:

$$Q_j = (1 - \alpha_j)q_{j-1} + \alpha_j q_j$$

With:

$$\alpha_j = \begin{cases} 1 & (j \le i - M + 1) \\ \dfrac{t - t_j}{t_{j+M-1} - t_j} & (i - M + 2 \le j \le i) \\ 0 & (j \ge i + 1) \end{cases}$$

Where:

Q	new control vertex
q	original control vertex
t	knot values (t_j are original knots while t is the inserted knot)
i	interval of inserted knot in the knot sequence
j	index into array
M	order of the B-Spline curve

For the cubic B-Spline curve (M = 4), the α equation can be simplified to:

$$\alpha_j = \begin{cases} 1 & (j \le i - 3) \\ \dfrac{t - t_j}{t_{j+3} - t_j} & (i - 2 \le j \le i) \\ 0 & (j \ge i + 1) \end{cases}$$

The non-uniform cubic B-Spline curve shown in Figure 8 on page 37 has a knot vector T_i (i = 0,1,2,3,4,5,6,7) with t_i = (0,1,2,4,5,8,9,11) and control vertices Q_j (j = 0,1,2,3). Inserting a knot at t = 4.5 (interval of insertion = 3) results in the following insertion equations:

$$Q_j = (1 - \alpha_j)q_{j-1} + \alpha_j q_j$$

$$\alpha_j = \begin{cases} 1 & (j \le 0) \\ \dfrac{t - t_j}{t_{j+3} - t_j} & (1 \le j \le 3) \\ 0 & (j \ge 4) \end{cases}$$

Solving for each new control vertex gives:

$$\begin{aligned}
Q_0 &= q_0 \\
Q_1 &= (1 - 0.875)q_0 + 0.875q_1 \\
Q_2 &= (1 - 0.4167)q_1 + 0.4167q_2 \\
Q_3 &= (1 - 0.1)q_2 + 0.1q_3 \\
Q_4 &= (1 - 0.0)q_3
\end{aligned}$$

Thus, the control vertex q_0 will remain the same. Control vertices q_1 and q_2 are shifted and a new control vertex is added at q_3. Subsequently, the remaining control vertex indices are shifted (e.g. $Q_4 = q_3$).

The new knot vector T = (0,1,2,4,4.5,5,8,9,11) and control polygon Q_i (i = 0,1,2,3,4). are shown in Figure 8 on page 37.

When inserting a knot into a B-Spline surface the same insertion equations are carried out for each row or column of the control vertex matrix. If a knot is to be inserted into the u knot sequence, the insertion equations will be used on each column of the control matrix. Likewise, if a knot is inserted into the w knot sequence, the insertion equations will be used on each row of the control matrix.

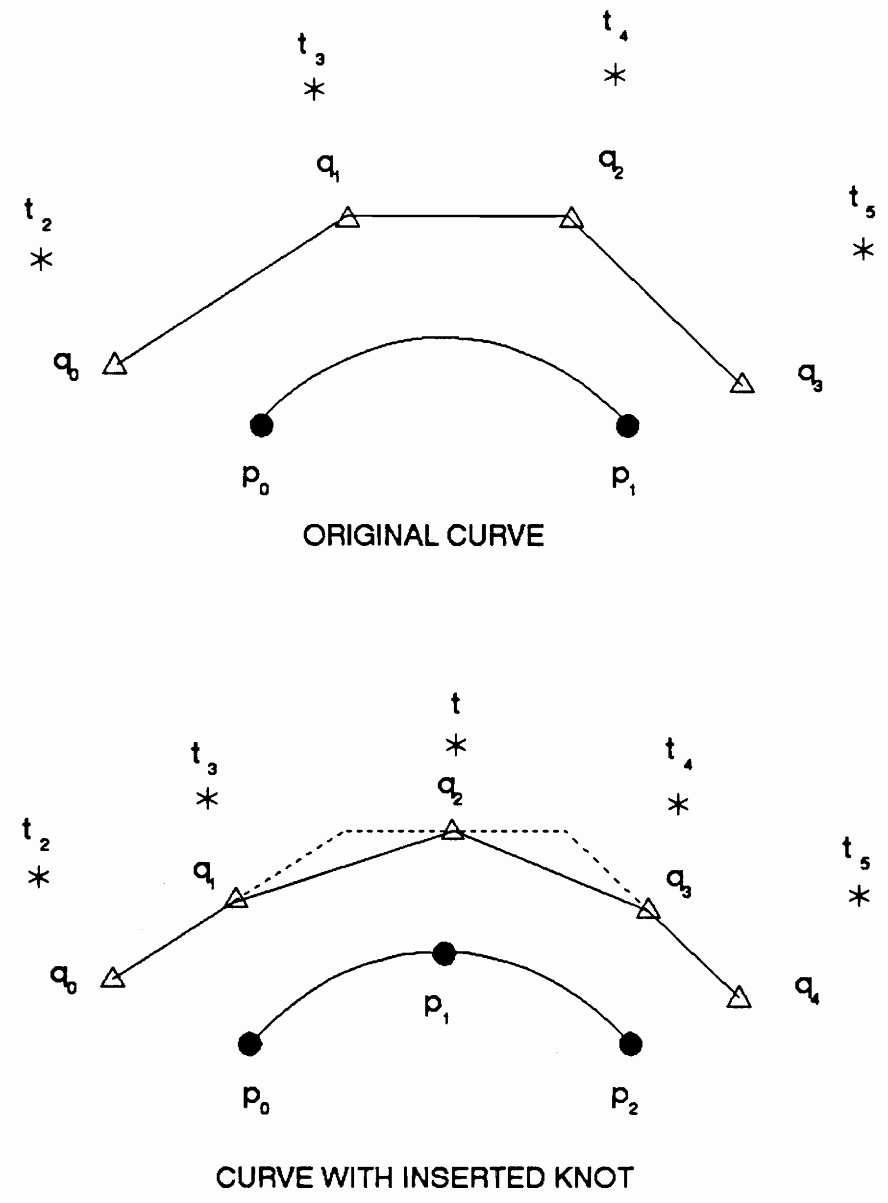


Figure 8. Knot Insertion in a Cubic B-Spline Curve

6.0 Cubic B-Spline Inversion

6.1 Uniform Cubic B-Spline Inversion

B-Spline inversion is the process of determining the control polygon for a B-Spline curve that passes through a given set of points. Yamaguchi [Yama88] defines "The Inverse Transformation" method for determining the control polygon for a uniform B-Spline curve.

The inverse transformation process is an iterative technique for finding the control vertices Q_j (j = 0,1,2,...,n) for the set of points P_i (i = 1,2,...,n-1) on a uniform cubic B-Spline curve.

When the parametric variable is equal to zero the point on a uniform cubic B-Spline curve is affected by only three control vertices as given in the following equation:

$$\frac{1}{6}Q_{i-1} + \frac{2}{3}Q_i + \frac{1}{6}Q_{i+1} = P_i \quad \begin{pmatrix} i = 1,2,\ldots,n-1 & (\textit{open curve}) \\ i = 0,1,\ldots n & (\textit{closed curve}) \end{pmatrix}$$

Because there are two fewer equations than unknowns (extra control vertex at each end of the B-Spline curve), the following end conditions are defined:

$$Q_0 = Q_1, \quad Q_n = Q_{n-1} \quad \textit{open curve}$$

$$Q_{-1} = Q_n, \quad Q_{n+1} = Q_0 \quad \textit{closed curve}$$

Figure 9 on page 40 illustrates these end conditions for both the open and closed curves. With the addition of these end conditions the number of equations is equal to the number of unknowns allowing the set of simultaneous equations to be solved. Yamaguchi defines an iterative technique for solving these equations by initially setting the control vertices equal to the points to be interpolated. Once the control vertices have been initialized, the following equation can be used to iteratively determine the control vertices:

$$\frac{1}{6} Q_{i-1}^{k} + \frac{2}{3} Q_i^{k} + \frac{1}{6} Q_{i+1}^{k-1} = P_i$$

Where k stands for the k^{th} iteration. This equation can be re-written in terms of Q_i^k as follows:

$$Q_i^k = P_i + \frac{1}{2} \left\{ P_i - \frac{1}{2} (Q_{i-1}^{k} + Q_{i+1}^{k-1}) \right\}$$

This equation solves for Q_i^k in terms of the previously computed control vertices. When the difference between the k^{th} and the (k^{th} - 1) iterations are less than some allowable error, the equation is solved.

The iterative technique for solving the simultaneous equations involves the following steps:

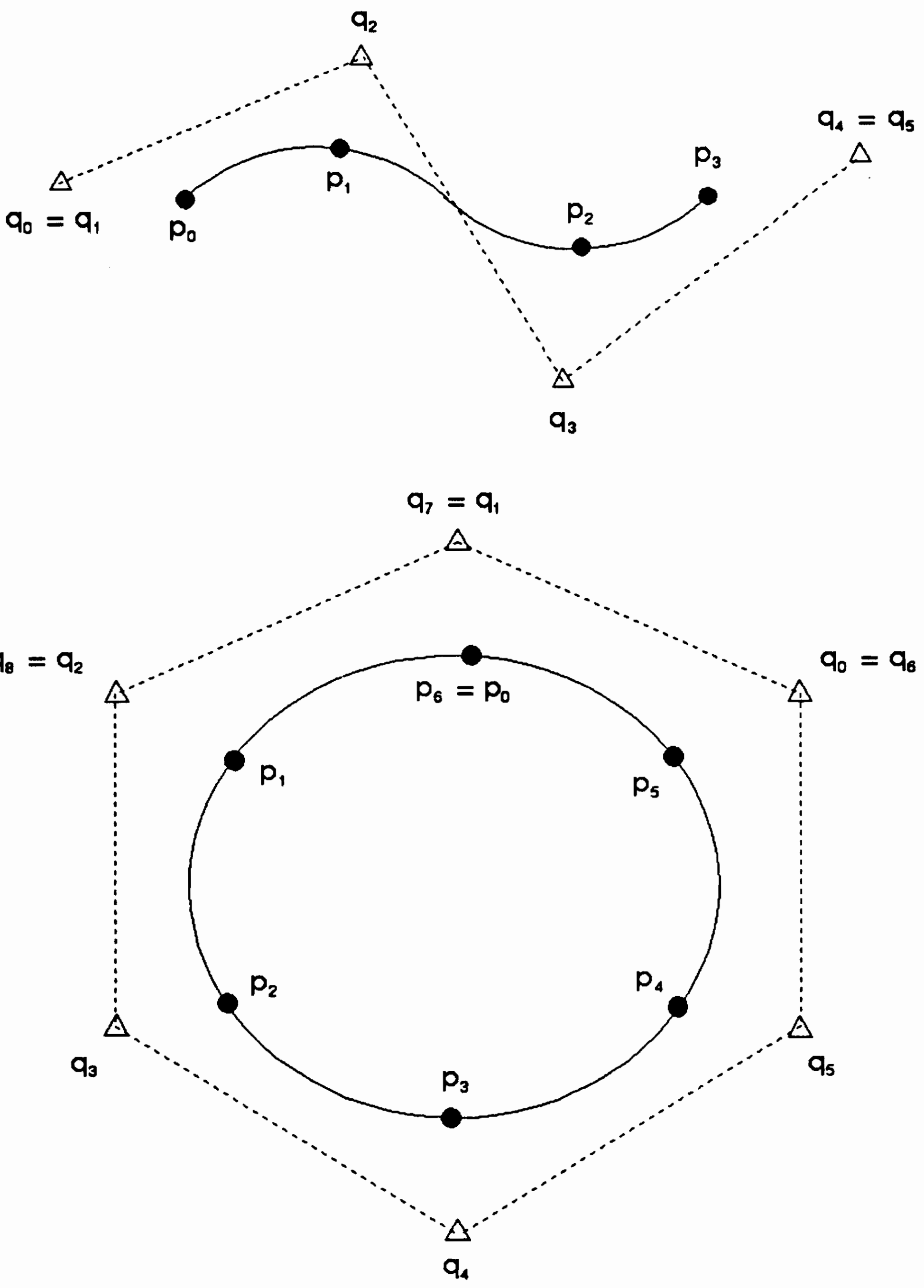


Figure 9. End Conditions for Open and Closed Curves

1. Initially set each control vertex equal to the points on the B-Spline curve. Specify the end conditions corresponding to the open or closed curve.

2. Solve for the next iteration of control vertices using the equation previously stated. Specify the end conditions corresponding to the open or closed curve.

3. Compute the difference between the k^{th} and the (k^{th} - 1) iterations

$$\delta_i^k = Q_i^k - Q_i^{k-1}$$

4. If the maximum difference between the k^{th} and (k^{th} - 1) iteration is greater than some allowable error, repeat steps two and three.

6.2 *Non-Uniform Cubic B-Spline Inversion*

Gloudemans [Glou90] extended Yamagughi's inversion technique for uniform curves to the non-uniform case. The equation for a general B-Spline curve is re-stated and expanded for a non-uniform cubic curve.

$$p(u) = \sum_{i=0}^{n} q_i N_{i,4}(u)$$

$$p(u) = N_{0,4}(t_i)q_{i-1} + N_{1,4}(t_i)q_i + N_{2,4}(t_i)q_{i+1} + N_{3,4}(t_i)q_{i+2}$$

It is known that a curve break point is affected by only three control vertices for a cubic B-Spline curve. Gloudemans applied this characteristic by computing the blending functions for a parametric value u = t_i and found that the fourth blending function goes to zero. Hence, the equation for a point along a non-uniform cubic B-Spline curve for every u = t_i is:

$$p_i = N_{0,4}(t_i)q_{i-1} + N_{1,4}(t_i)q_i + N_{2,4}(t_i)q_{i+1} \quad (i = 1,2,\ldots,n)$$

Similar to the formulation for inverting a uniform cubic B-Spline curve, the above equation can be rewritten to solve for q_i

$$q_i^k = \frac{1}{N_{1,4}(t_i)} \{p_i - (N_{0,4}(t_i)q_{i-1}^k + N_{2,4}(t_i)q_{i+1}^{k-1})\}$$

Before the non-uniform inversion process can be initiated, a knot sequence or knot vector must be determined for the curve points. The selection of a knot sequence is referred to as the parameterization of a curve. The three most common parameterization techniques are Uniform, Chord length and Centripetal. Each of these are defined below:

Uniform The spacing between subsequent knot values is uniform (i.e. T_{i+1} - T_i = T_{i+2} - T_{i+1}). This process can cause unwanted fluctuations when the points to be interpolated are unevenly spaced.

Chord length The knot spacing is proportional to the distance between two subsequent points.

Centripetal The knot spacing is proportional to the square root of the distance between points. This parameterization technique allows tighter corners than those allowed by chord length parameterization.

The chord length parameterization technique is used for the inversion of points in this thesis. Chord length was chosen for reasons which will be discussed in the filleting algorithm.

6.2.1 Non-Uniform Cubic B-Spline Inversion Process

The non-uniform cubic B-Spline inversion process is composed of the following five steps:

1. Parameterize the points to be interpolated (in this case the chord length parameterization technique is used).

2. Initially set each control vertex equal to the point on the B-Spline curve. Specify the end conditions corresponding to the open or closed curve.

3. Solve for the next iteration of control vertices using the following equation. Again specify the end conditions corresponding to the open or closed curve.

$$q_i^k = \frac{1}{N_{1,4}(t_i)} \{p_i - (N_{0,4}(t_i)q_{i-1}^k + N_{2,4}(t_i)q_{i+1}^{k-1})\}$$

4. Compute the difference between the k^{th} and the (k^{th} - 1) iterations.

$$\delta_i^k = Q_i^k - Q_i^{k-1}$$

5. If the maximum difference between the k^{th} and (k^{th} - 1) iterations is greater than some allowable error, repeat steps three and four.

6.3 *Forcing a Tangent Discontinuity*

The ability to force a tangent discontinuity in a B-Spline curve gives the designer freedom to produce sharp corners without inverting two separate sets of points. A good example of this is the inversion of a square with tangent discontinuities at each corner.

The steps taken to force a tangent discontinuity are described below. Figure 10 on page 45 illustrates the curve to be inverted with a tangent discontinuity and the resulting knot sequence and control polygon.

Assume we are given a set of points P_i (i = 0,1,2,3,4,5) to be interpolated with a non-uniform cubic B-Spline curve. Additionally, set $p_2 = p_3$. The resulting parameterization will be T_j (j = 0,1,2,3,4,5,6,7,8,9,10) with $t_5 = t_6$.

In attempting to invert this set of points with the knot sequence computed, a blending function of zero will result for $N_{1,4}$, $N_{2,4}$ and $N_{3,4}$ at j = 5. Restating the inversion equation:

$$q_i^k = \frac{1}{N_{1,4}(t_i)} \{p_i - (N_{0,4}(t_i)q_{i-1}^k + N_{2,4}(t_i)q_{i+1}^{k-1})\}$$

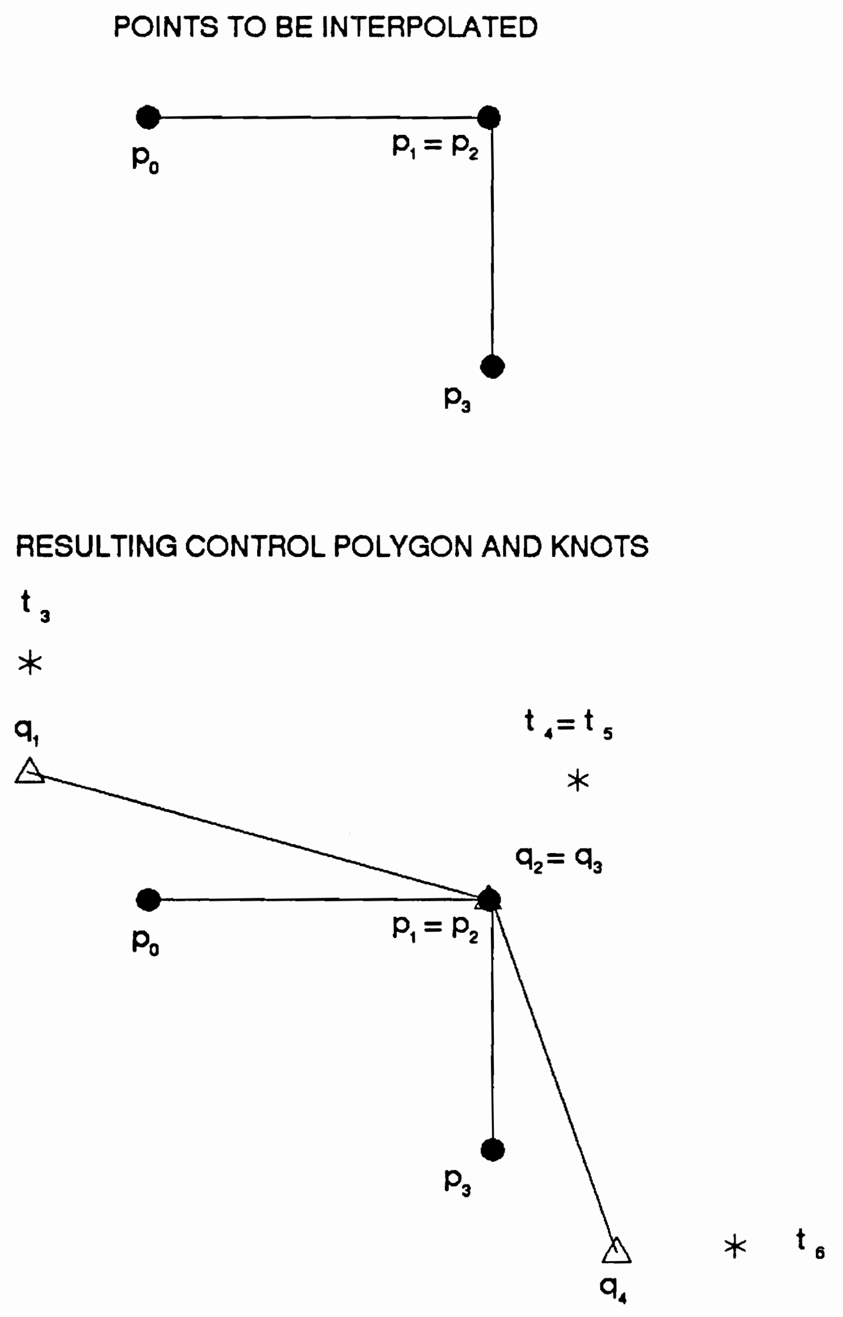


Figure 10. Forcing a Tangent Discontinuity

It is apparent that this will cause erroneous results when an attempt is made to divide by zero. This situation can be corrected by setting the control vertex equal to the point to be interpolated whenever a zero is found for the blending function $N_{1,4}$.

When computing the next control vertex (j = 6) the blending functions $N_{2,4}$ and $N_{3,4}$ will be equal to zero resulting in $q_i = q_{i-1}$. The resulting control polygon Q_l (l = 0,1,2,3,4,5,6) will have coincident control vertices for l = 3 and l = 4 ($q_3 = q_4$).

Again, the equation for computing a point along a non-uniform cubic B-Spline curve is:

$$p(u) = N_{0,4}(t_i)q_{i-1} + N_{1,4}(t_i)q_i + N_{2,4}(t_i)q_{i+1} + N_{3,4}(t_i)q_{i+2}$$

However, when the parameter value u is equal to any one of the knot values (curve segment break point) the fourth blending function goes to zero giving:

$$p(u) = N_{0,4}(t_i)q_{i-1} + N_{1,4}(t_i)q_i + N_{2,4}(t_i)q_{i+1}$$

Furthermore, at the parameter value u $= t_6$ the third blending function also goes to zero resulting in:

$$p(u) = N_{0,4}(t_i)q_{i-1} + N_{1,4}(t_i)q_i$$

Finally, the blending functions $N_{0,4}$ and $N_{1,4}$ equal 0.5. More importantly, the two control vertices equal each other and the computed point equals the interpolated point. The curve has a tangent discontinuity by definition because it is no longer piecewise linearly continuous over more than one point.

6.4 Surface Inversion

The surface inversion algorithm used in this thesis is a direct extension of the non-uniform cubic B-Spline curve inversion process previously stated. The surface inversion process is repeated for both parametric directions as explained below.

The surface is inverted in the first parametric direction as independent curves. The control polyhedron from the first inversion is used as the points to interpolate for the second parametric direction. In this manner the final control polyhedron is a tensor product of both parametric directions.

7.0 Intersection of B-Spline Surfaces

Computing the curve of intersection between two free form surfaces has remained a difficult problem. Knowledge of the curve of intersection is needed for computing a model's volume, trimming surfaces, NC tool path determination, etc.. Both Peng [Peng84] and Lasser [Lass86] have developed "Divide and Conquer" algorithms for computing the curve of intersection between B-Spline or Bezier Surfaces. The following section discusses the combined implementation of both algorithms to produce a robust intersection method for non-uniform bi-cubic B-Spline surfaces. The first section defines the subdivision algorithm in a generic format. The latter sections discuss how this algorithm is implemented for the non-uniform bi-cubic B-Spline definition.

7.1 The Subdivision Algorithm

The subdivision algorithm is comprised of the following eight steps:

1. Build bounding boxes for each separate surface (component) by determining the minimum and maximum X, Y, and Z Cartesian coordinates for the surfaces (Figure 11 on page 51).

2. Compare two bounding boxes for a possible intersection. If the check is successful, go to step three, otherwise test a different set of bounding boxes (Figure 11 on page 51).

3. Any two components that pass the component bounding box test are compared on a patch by patch basis. A bounding box check is performed for each patch on one component against every patch on the other component.

4. The larger of the two patches which pass the bounding box check is split into four sub-patches. The sub-patches inherit the controversy (possibility of intersection) from their parent patch.

5. New bounding box checks are performed on each of the sub-patches against their parent's adversary.

6. Steps four and five are repeated until both sub-patches can be approximated by a plane. At this time the intersection between any two sub-patches can be determined by calculating the intersection between two planes.

7. Steps three, four, five and six are repeated until all possible intersection checks have been exhausted between two components.

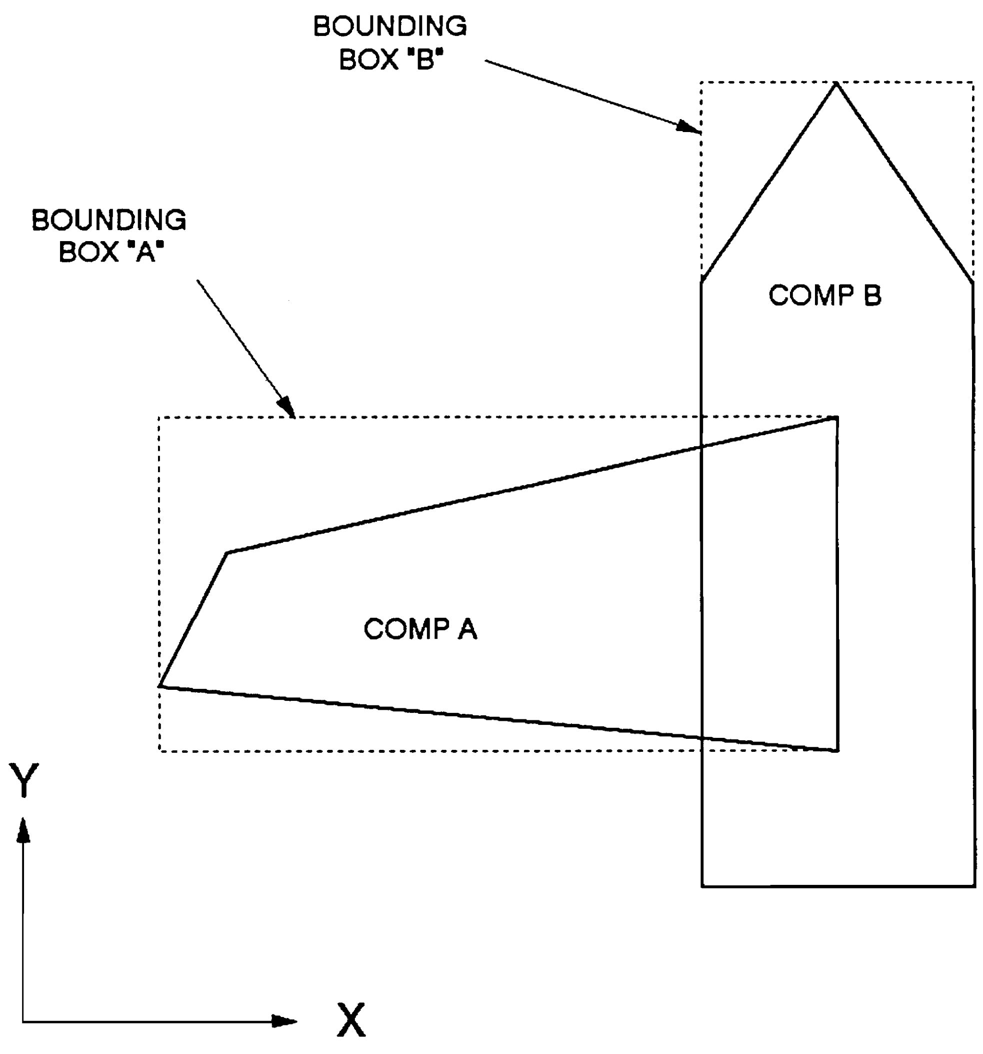
8. The intersection data is sorted to form one or more continuous curves of intersection.

Figure 12 on page 52 illustrates the sub-divisions on a patch along with the curve of intersection.

7.2 Bounding Boxes

Two types of bounding boxes are needed for the subdivision algorithm: component bounding boxes and patch bounding boxes. Both bounding boxes are computed by determining the minimum and maximum X, Y, and Z values of the control polyhedron. This type of of bounding box is larger than one computed from the actual surface points. However, building a bounding box from actual surface points requires a large number of computations.

Both the B-Spline and Bezier control polyhedrons approximate the corresponding surface. However, the Bezier control polyhedron interpolates the surface patch at each of the corner points. This characteristic of the Bezier control polyhedron results in a smaller bounding box which reduces the number of subdivisions performed. Also, a better approximation of the surface is made when the control polyhedron is assumed to be a plane. Figure 13 on page 53 illustrates the difference between the B-Spline and Bezier control polygons.



BOUNDING BOX TEST

IF $A_{MIN} < B_{MAX}$ AND $A_{MAX} > B_{MIN}$ THEN
POSSIBLE INTERSECTION EXISTS

Figure 11. Building a Bounding Box

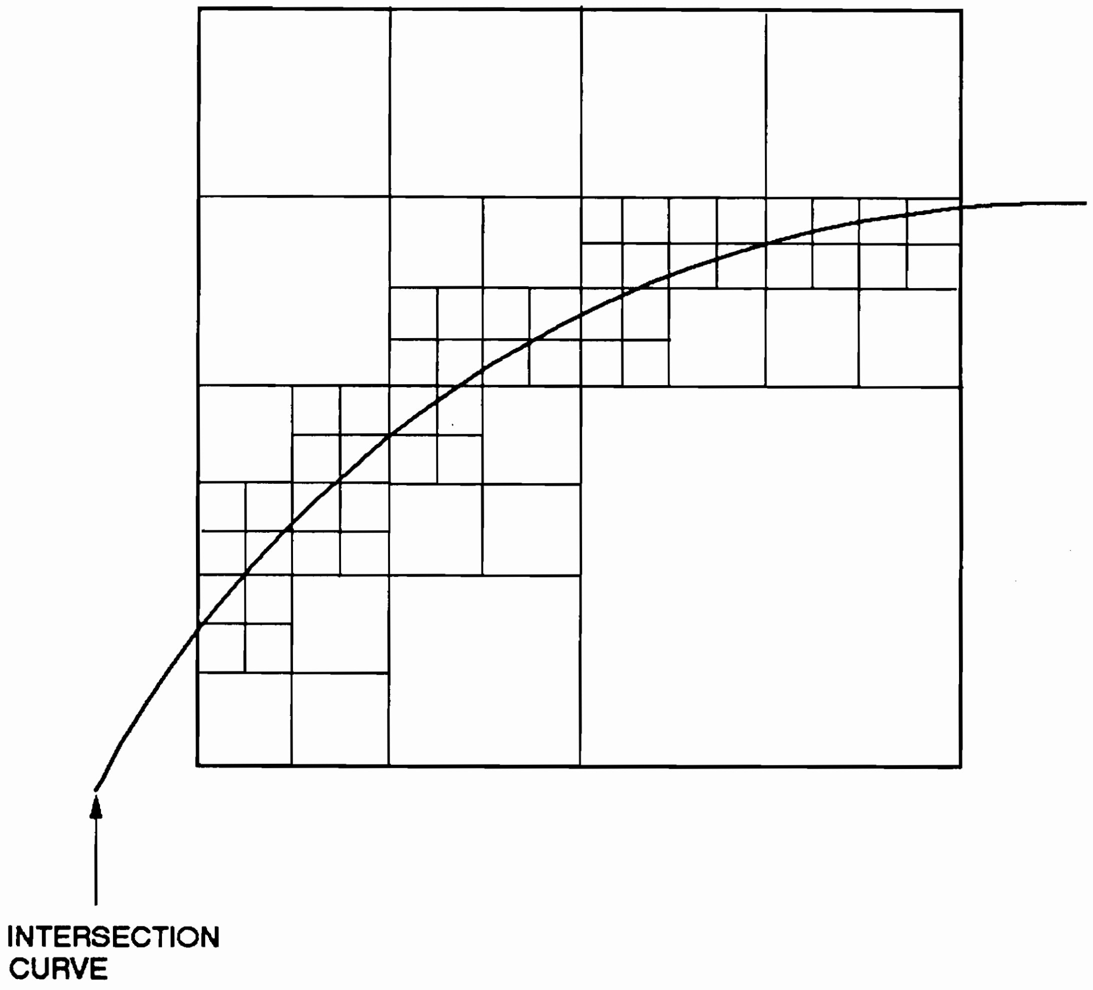


Figure 12. Subdivided Patch and Intersection Curve

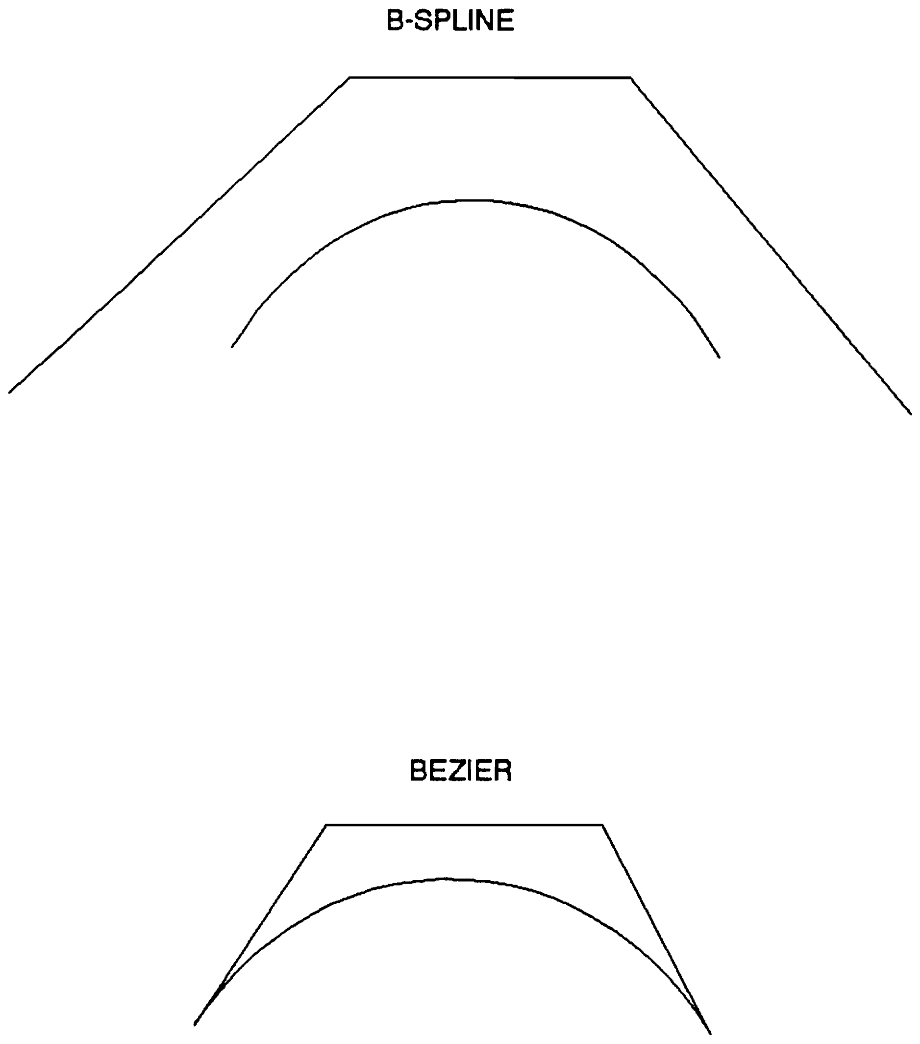


Figure 13. B-Spline and Bezier Control Polygons

7.2.1 Converting to Bezier Format

A knot multiplicity of three (three coincident knots) for a bi-cubic B-Spline curve results in the surface interpolating the control vertex. Hence, forcing a knot multiplicity of three at each patch corner point will result in a Bezier type control polyhedron. Because a knot value already exists at a patch corner point, only two knot insertions are needed to obtain a knot multiplicity of three.

An illustration of the conversion to the Bezier type control polygon is shown in Figure 14 on page 56. The figure displays one curve segment of a non-uniform cubic B-Spline curve with a knot vector T_i (i = 0,1,2,3,4,5,6,7) and control polygon Q_j (j = 0,1,2,3). Inserting a knot a t = t_3 gives the following set of insertion equations:

$$q_0^1 = q_0$$

$$q_1^1 = \left(1 - \frac{t_3 - t_1}{t_4 - t_1}\right)q_0 + \left(\frac{t_3 - t_1}{t_4 - t_1}\right)q_1$$

$$q_2^1 = \left(1 - \frac{t_3 - t_2}{t_5 - t_2}\right)q_1 + \left(\frac{t_3 - t_2}{t_5 - t_2}\right)q_2$$

$$q_3^1 = \left(1 - \frac{t_3 - t_3}{t_6 - t_3}\right)q_2 + \left(\frac{t_3 - t_3}{t_6 - t_3}\right)q_3 = q_2$$

$$q_4^1 = q_3$$

It should be noted that only two control vertices are affected when a knot is inserted on top of an existing knot. The new knot vector T_i (i = 0,1,2,3,3,4,5,6,7) and control vertices Q_j (j = 0,1,2,3,4) are also shown in Figure 14 on page 56. Inserting another knot at t = t_3 will result with the following set of insertion equations:

$$q_0^{11} = q_0^1 = q_0$$

$$q_1^{11} = q_1^1$$

$$q_2^{11} = \left(1 - \frac{t_3 - t_2}{t_4 - t_2}\right)q_1^1 + \left(\frac{t_3 - t_2}{t_4 - t_2}\right)q_2^1$$

$$q_3^{11} = \left(1 - \frac{t_3 - t_3}{t_5 - t_3}\right)q_2^1 + \left(\frac{t_3 - t_3}{t_5 - t_3}\right)q_3^1 = q_2^1$$

$$q_4^{11} = \left(1 - \frac{t_3 - t_3}{t_6 - t_3}\right)q_3^1 + \left(\frac{t_3 - t_3}{t_6 - t_3}\right)q_4^1 = q_3^1 = q_2$$

$$q_5^{11} = q_4^1 = q_3$$

This time, only one control vertex was affected by inserting the knot on top of two coincident knots. This knot insertion is also shown in Figure 14 on page 56. The control vertex q_2 interpolates the B-Spline curve at the break point associated with the original knot t_3.

Inserting two knots at the other break point of the curve will force the control vertex q_5 to interpolate the other break point which was originally associated with knot t_4. This process is used to force the control polyhedron of a non-uniform bi-cubic B-Spline patch to be interpolated by the patch corner points. Two knots are inserted on each side of the patch in both parametric directions for a total of eight knot insertions.

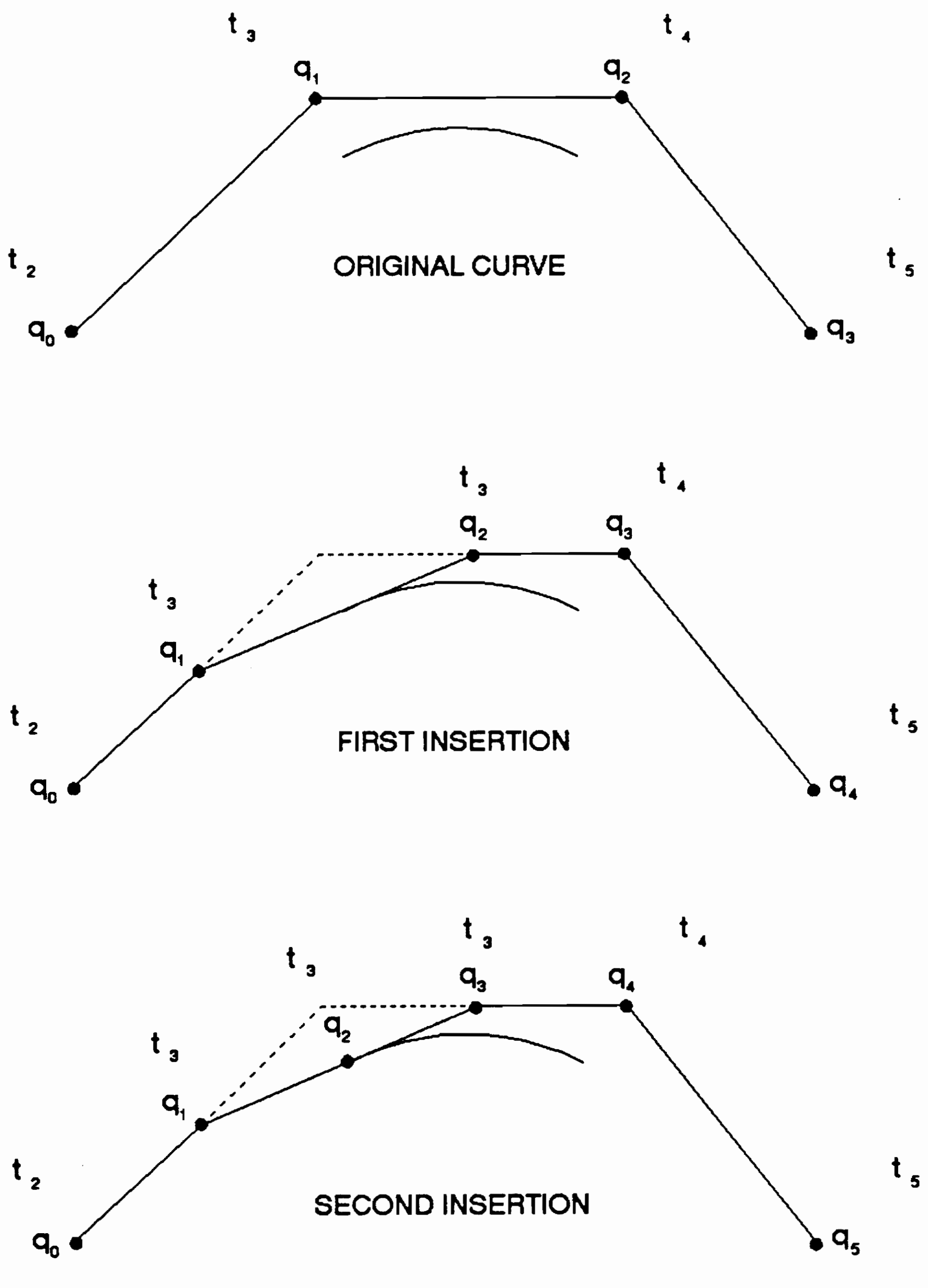


Figure 14. Conversion from B-Spline to Bezier Format

7.3 Subdividing a Patch

Subdividing a patch is the process of splitting the control polyhedron of one bi-cubic patch into four separate control polyhedrons. This is performed by forcing a knot multiplicity of three in both parametric directions equally spaced between the patch break points. This process will be demonstrated using the converted control polygon described in the preceding section.

The curve segment shown in Figure 15 on page 60 has a knot vector T = (t_3, t_3, t_3, t_4, t_4, t_4) and control polygon Q = (q_2, q_3, q_4, q_5). It is desired that the control polygon be split to obtain two equally spaced curve segments. In order to accomplish this, a knot multiplicity of three must be inserted into the knot sequence at t = (t_3 + t_4)/2

Inserting the first knot t = (t_3 + t_4)/2 into the knot vector gives the following α values for all three affected control vertices.

$$\alpha_{3,4,5} = \frac{t - t_3}{t_4 - t_3}$$

Substituting (t_3 + t_4)/2 for t gives:

$$\alpha_{3,4,5} = \frac{\frac{t_4 + t_3}{2} - t_3}{t_4 - t_3} = \frac{t_4 - t_3}{2(t_4 - t_3)} = \frac{1}{2}$$

Thus, $\alpha = 0.5$ for all cases. Knowing this α value saves a number of computations while performing the subdivision algorithm resulting in less time required to compute the intersection curve. Using the α value of 0.5 for the knot insertion gives the following equations:

$$\begin{aligned}
q_2^1 &= q_2 \\
q_3^1 &= (1 - 0.5)q_2 + 0.5\, q_3 \\
q_4^1 &= (1 - 0.5)q_3 + 0.5\, q_4 \\
q_5^1 &= (1 - 0.5)q_4 + 0.5\, q_5 \\
q_6^1 &= q_5
\end{aligned}$$

Inserting a second knot t = (t_3 + t_4)/2 into the knot sequence gives α values of

$$\alpha_{4,5} = \frac{t - t_3}{t_4 - t_3} = 0.5 \qquad \alpha_6 = \frac{t - t}{t_4 - t} = 0.0$$

Which gives the insertion equations:

$$\begin{aligned}
q_2^{11} &= q_2^1 = q_2 \\
q_3^{11} &= q_3^1 \\
q_4^{11} &= (1 - 0.5)q_3^1 + 0.5\, q_4^1 \\
q_5^{11} &= (1 - 0.5)q_4^1 + 0.5\, q_5^1 \\
q_6^{11} &= q_5^1 \\
q_7^{11} &= q_6^1 = q_5
\end{aligned}$$

Finally, inserting the knot t = (t_3 + t_4)/2 into the sequence a third time gives α values of:

$$\alpha_5 = \frac{t - t_3}{t_4 - t_3} = 0.5 \qquad \alpha_{6,7} = \frac{t - t}{t_4 - t} = 0.0$$

Resulting in the insertion equations:

$$\begin{aligned}
q_2^{111} &= q_2^{11} = q_2^{1} = q_2 \\
q_3^{111} &= q_3^{11} = q_3^{1} \\
q_4^{111} &= q_4^{11} \\
q_5^{111} &= (1 - 0.5)q_4^{11} + 0.5\, q_5^{11} \\
q_6^{111} &= q_5^{11} \\
q_7^{111} &= q_6^{11} = q_5^{1} \\
q_8^{111} &= q_7^{11} = q_6^{1} = q_5
\end{aligned}$$

The curve segment has now been split into two curve segments and the control polygon has been adjusted to interpolate both curve segments at their end points. Figure 15 on page 60 displays the control polygon after each of the three knot insertions. The knot vector and control polygon for the first curve segment are $T^A = (t_3, t_3, t_3, t, t, t)$ and $Q^A = (q_2, q_3, q_4, q_5)$. The knot vector and control polygon for the second curve segment are $T^B = (t, t, t, t_4, t_4, t_4)$ and $Q^B = (q_5, q_6, q_7, q_8)$.

The same process is performed for splitting a bi-cubic patch. By inserting three knots in one parametric direction and then inserting three knots in the other parametric direction the patch can be split into four sub-patches.

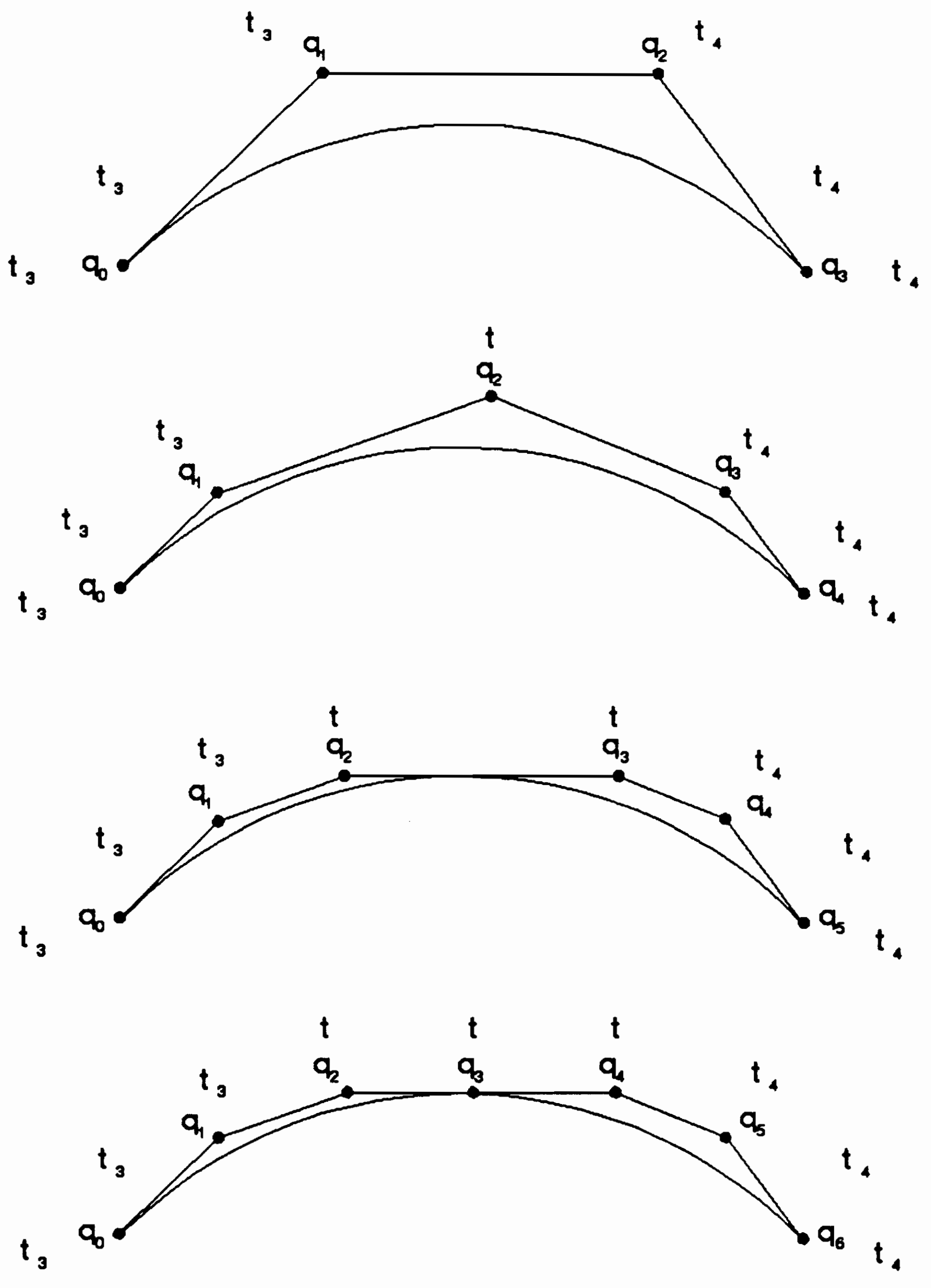


Figure 15. Splitting a Control Polygon

7.4 Plane Approximations

The intersection algorithm recursively subdivides a patch until either a plane approximation can be made or the possibility of an intersection ceases to exist (bounding box check fails). Two types of checks are made to determine if the patch can be approximated by a plane. The first of these is to calculate the distance of the interior four control vertices q_{11}, q_{12} , q_{21} and q_{22} from the plane described by control vertices q_{00} , q_{30} and q_{03}. The distance of control vertex q_{33} from the plane is also computed. The second check is the distance from the interior edge control vertices and a line drawn between the corresponding corner vertices. For example, the distance from control vertices q_{10} and q_{20} to a line drawn between control vertices q_{00} and q_{30} is computed. The largest value returned from all twelve tests is compared against a specified tolerance. If the largest value is less than the specified tolerance the patch is approximated by a plane. Refer back to Figure 7 on page 32 for an illustration of a B-Spline patch control polygon.

7.4.1 Minimum Distance from a Point to a Plane

Hill [Hill90] defines a method which is used in this thesis for finding the minimum distance from a point to a plane described with three points. The first obstacle of this algorithm is to define the plane in the form Ax + By + Cz = D, given the three points P_1, P_2 and P_3.

The coefficients A, B and C of the plane equation represent the plane normal vector, while the value D is the projection of any vector from the origin to a point lying in the plane, onto the plane normal vector. An example of this description is shown in Figure 16 on page 63. The plane normal vector is found by computing the cross product of the two vectors defined by points P_1, P_2 and P_3. The value D is then found by computing the dot product of the plane normal vector and the point P_2. This sequence of computations is sometimes called the scalar triple product.

Assume the points shown in Figure 16 on page 63 are $P_1 = (1, 0, 2)$, $P_2 = (2, 3, 0)$ and $P_3 = (1, 2, 4)$. The vectors a and b are computed as follows:

$$\vec{a} = [2\ \ 3\ \ 0] - [1\ \ 0\ \ 2] = [1\ \ 3\ \ -2]$$
$$\vec{b} = [1\ \ 2\ \ 4] - [1\ \ 0\ \ 2] = [0\ \ 2\ \ 2]$$

The cross product of vectors $\vec{a}$ and $\vec{b}$ is:

$$\vec{n} = \begin{bmatrix} i & j & k \\ 1 & 3 & -2 \\ 0 & 2 & 2 \end{bmatrix} = [10\ \ -2\ \ 2]$$

Finally, the value D is obtained by the dot product of $\vec{r}$ and $\vec{n}$:

$$D = [1\ \ 0\ \ 2] \bullet [10\ \ -2\ \ 2] = 14$$

Thus, the equation for a plane passing thru the points (1,0,2), (2,3,0) and (1,2,4) is:

$$10x - 2y + 2z = 14$$

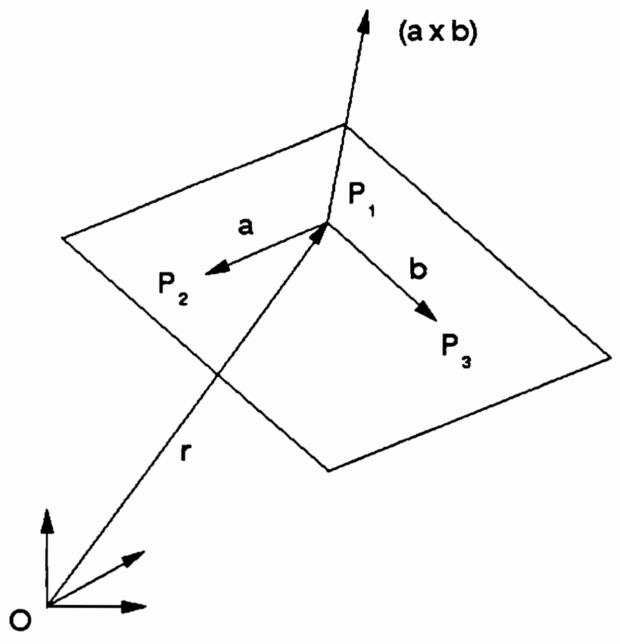


Figure 16. Computing Equation of a Plane Given Three Points

Once the equation of the plane is computed, the minimum distance from a point (x_1, y_1, z_1) to a plane Ax + By + Cz = D is:

$$d_{\min} = \left| \frac{Ax_1 + By_1 + Cz_1 - D}{\sqrt{A^2 + B^2 + C^2}} \right|$$

7.4.2 Minimum Distance from a Point to a Line

The cross product is utilized to compute the minimum distance from a point to line [Lin91]. The minimum distance between the point P_1, shown in Figure 17 on page 65 and the line defined by the points P_2 and P_3 can computed using the equation:

$$d = |\vec{v}| \sin\theta$$

However, the angle theta is unknown. The cross product of the unit vector $\vec{u}$ and the vector $\vec{v}$ is:

$$|\vec{u} \times \vec{v}| = |\vec{u}|\,|\vec{v}| \sin\theta$$

The magnitude of a unit vector = 1. Therefore, the equation to compute the distance from a point to a line is:

$$d = |\vec{u} \times \vec{v}|$$

Where:

d = minimum distance between a point and a line

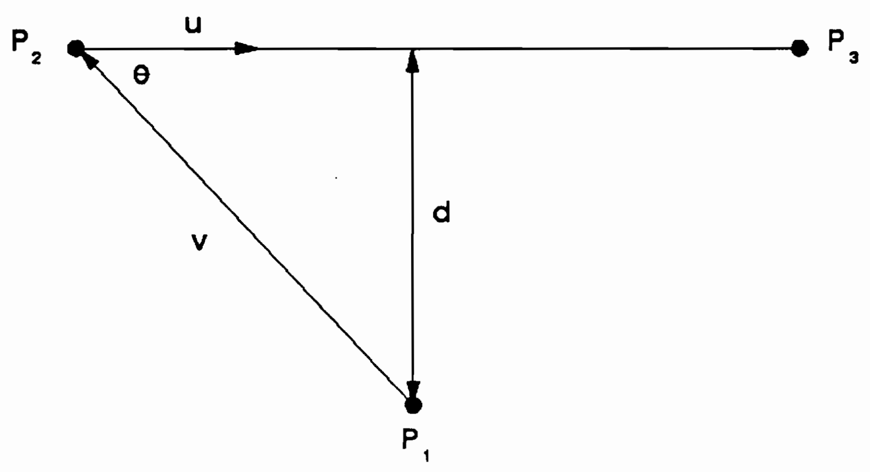


Figure 17. Computing the Minimum Distance Between a Point and a Line

$$\vec{u} = \frac{(x_3 - x_2)}{|\overline{P_2P_3}|}\vec{i} + \frac{(y_3 - y_2)}{|\overline{P_2P_3}|}\vec{j} + \frac{(z_3 - z_2)}{|\overline{P_2P_3}|}\vec{k}$$

$$\vec{v} = (x_2 - x_1)\vec{i} + (y_2 - y_1)\vec{j} + (z_2 - z_1)\vec{k}$$

7.5 Computing Plane-Plane Intersections

Approximating the B-Spline patch by a bounded plane can cause large errors in calculating the intersection of that patch with another surface. Figure 18 on page 67 displays two of the problems associated with this approximation. All four corner control vertices (q_{00} , q_{30} , q_{03} and q_{33}) may not lie in the same plane. Therefore, approximating the patch by a plane described by control vertices q_{00}, q_{30} and q_{03} may result in the inaccurate calculation of an intersection point. Secondly, if the patch is degenerate as shown in the second part of Figure 18 on page 67, a bounded plane approximation could again give an erroneous intersection value.

To correct these problems, the plane-plane intersection is actually computed using triangle approximations for each patch. Thus, all four corner control vertices are involved in the intersection calculation. The intersection is computed by testing a series of line-triangle combinations for intersections.

Figure 19 on page 69 represents two intersecting polygons. The test sequence consists of first testing for an intersection between triangle 1 of polygon A against any of the four edges of polygon B. Next, triangle 2 of polygon A is compared with the same four edges

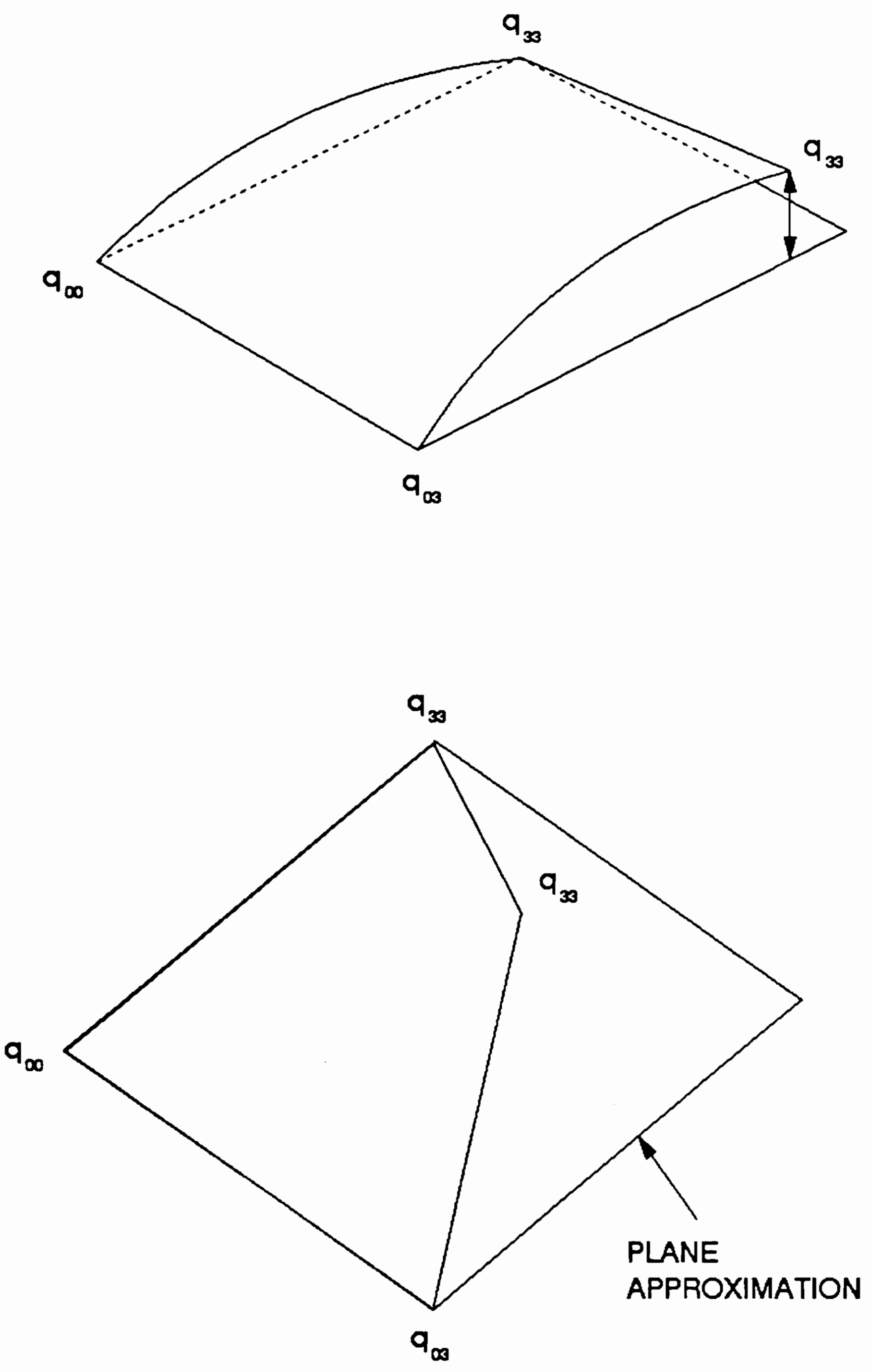


Figure 18. Error in Bounded Plane Approximation.

of polygon B. The same sequence of tests are performed using the triangles on polygon B and edges of polygon A. Once two distinct points of intersection are found, the line-triangle intersection test is terminated. Both intersection points are stored along with the corresponding parametric values on both surfaces into a 'line' data structure. This line data structure is then added to a linked list for later sorting. The format of the 'line' data structure will be discussed later. If two points of intersection are not found, the test is terminated and no intersection data is returned.

7.5.1 Computing the Intersection between a Triangle and a Line

Computing the intersection for a triangle and a line is a simple extension of the line-plane intersection discussed in Mortenson [Mort85]. Figure 20 on page 71 illustrates an intersection between a line and a plane. The point of intersection can be found by setting the equation of the plane equal to the equation of the line.

$$\vec{A} + u\vec{B} + w\vec{C} = \vec{D} + s\vec{E}$$

All three parametric values u,w and c are unknown in the above equation. Multiplying both sides of the equation by the cross product of vectors B and C gives:

$$(\vec{B} \times \vec{C}) \bullet \vec{A} + (\vec{B} \times \vec{C}) \bullet u\vec{B} + (\vec{B} \times \vec{C}) \bullet w\vec{C} = (\vec{B} \times \vec{C}) \bullet \vec{D} + (\vec{B} \times \vec{C}) \bullet s\vec{E}$$

Since the cross product of vectors $\vec{B}$ and $\vec{C}$ is perpendicular to both $\vec{B}$ and $\vec{C}$, the above equation can be reduced to:

$$(\vec{B} \times \vec{C}) \bullet \vec{A} = (\vec{B} \times \vec{C}) \bullet \vec{D} + (\vec{B} \times \vec{C}) \bullet s\vec{E}$$

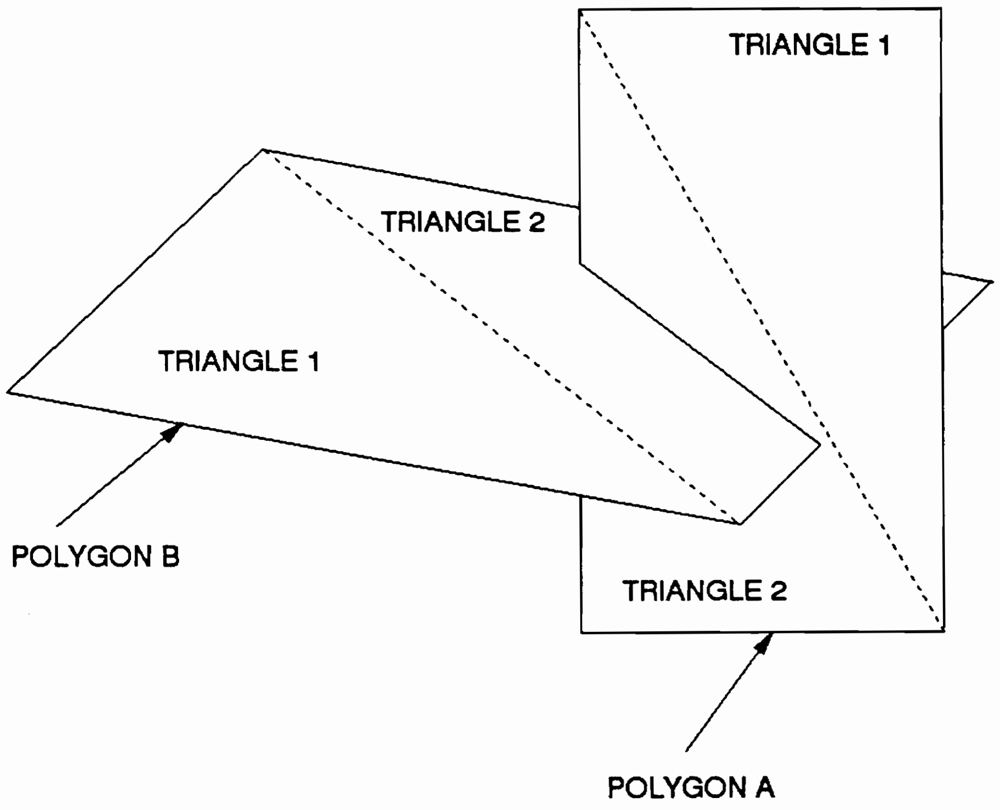


Figure 19. Computing the Intersection Using Triangle Approximations

Rearranging the equation gives:

$$s = \frac{(\vec{B} \times \vec{C}) \bullet \vec{A} - (\vec{B} \times \vec{C}) \bullet \vec{D}}{(\vec{B} \times \vec{C}) \bullet \vec{E}}$$

The same method can be applied to solve for parametric values u and w.

$$u = \frac{(\vec{C} \times \vec{E}) \bullet \vec{D} - (\vec{C} \times \vec{E}) \bullet \vec{A}}{(\vec{C} \times \vec{E}) \bullet \vec{B}}$$

$$w = \frac{(\vec{B} \times \vec{E}) \bullet \vec{D} - (\vec{B} \times \vec{E}) \bullet \vec{A}}{(\vec{B} \times \vec{E}) \bullet \vec{C}}$$

An intersection between the line and the plane exists if the parametric values u, w and s all fall in the range of 0 and 1 (bounded line and plane). To extend this formulation to a triangle-line intersection an additional constraint of $u + w \leq 1$ must be met. If $u + w$ is greater than 1, an intersection does not exist between the triangle and the line.

7.6 Sorting the Intersection Data

As mentioned earlier, the intersection points of two planes are loaded into a 'line' data structure. Each line data structure contains both points of intersection as well as the parametric values on both surfaces corresponding to the points. Once all possible intersections have been exhausted between two surface patches, a 'patch' data structure is made. The patch data structure contains pointers to the beginning and end of the linked list of line data structures.

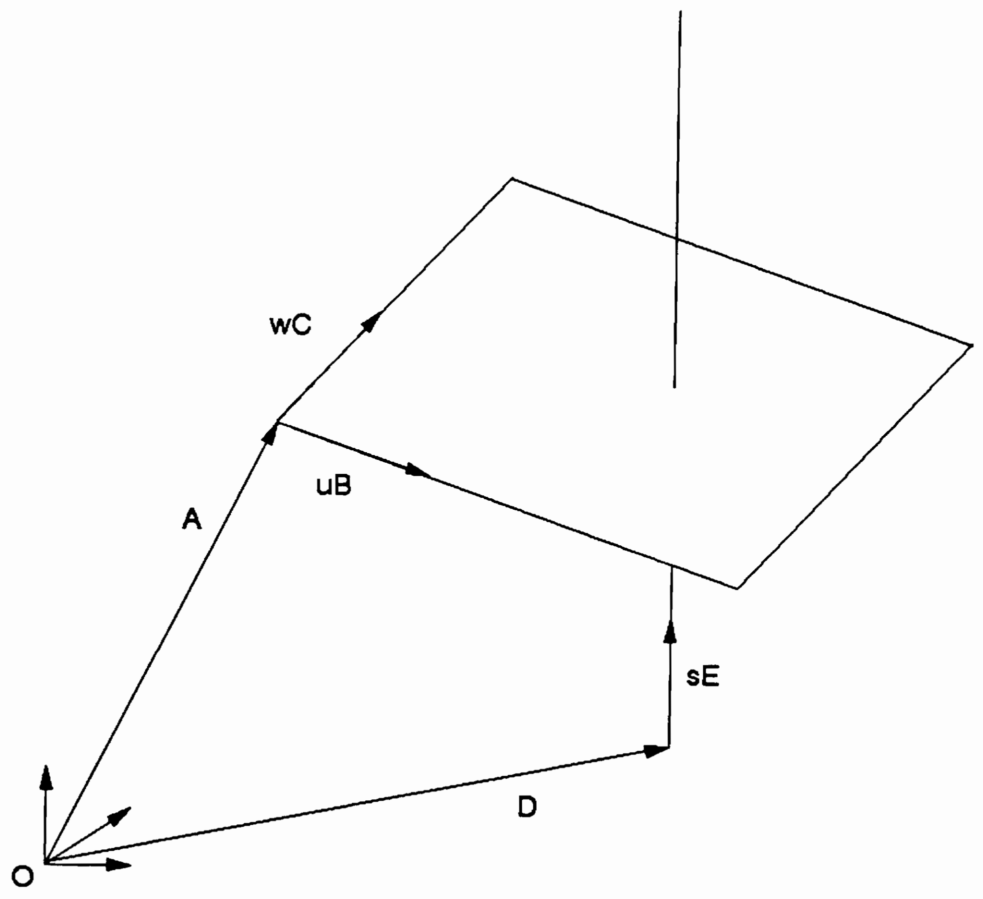


Figure 20. Intersection Between a Line and Plane

Once all possible patch intersections have been exhausted between two components, an intersection curve data structure is loaded. The intersection curve data structure contains the pointer to the beginning of the intersection list. This data structure is a part of another set of linked lists. The intersection linked list is a list of intersection curves between components of a model. A schematic of the three data structures is shown in Figure 21 on page 73

These three levels of data are utilized to reduce the number of sorting computations which reduces the sorting time. The same sorting algorithm is applied to both the line and patch levels of data storage. The line linked lists are each sorted independently. Once all line linked lists have been sorted, the algorithm is applied to the patch linked lists.

7.6.1 The Sorting Algorithm

The algorithm for sorting the intersection data makes no assumptions about the type of data or relationships between data. The sort is based on the distance between parametric values of two separate sets of intersection data. The algorithm is similar to the standard bubble sort algorithm and consists of the following sequence of events [Aho83].

1. Take the first element from the unsorted list and make it the first element in the sorted list.

2. Compare the first number of each of the unsorted elements against the first and last number of the sorted list.

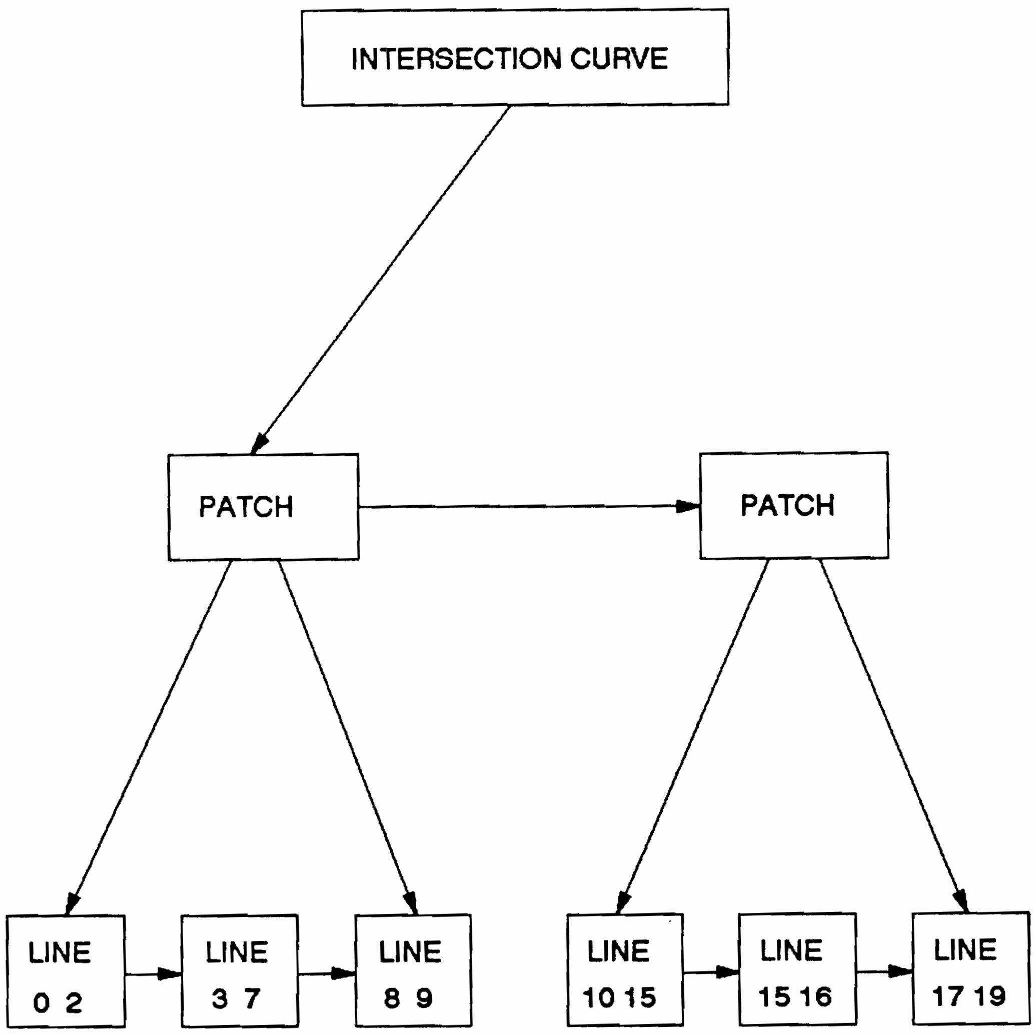


Figure 21. Intersection Data Storage Schematic

3. Compare the second number of each of the unsorted elements against the first and last number of the sorted list.

4. Add the closest number to the list before or after the sorted elements based on the comparisons. If the closest number is not within a specified tolerance, start a new list.

5. Repeat steps two, three and four until the unsorted list is exhausted.

Figure 22 on page 75 gives an illustration of the sorting algorithm at either the line or patch level.

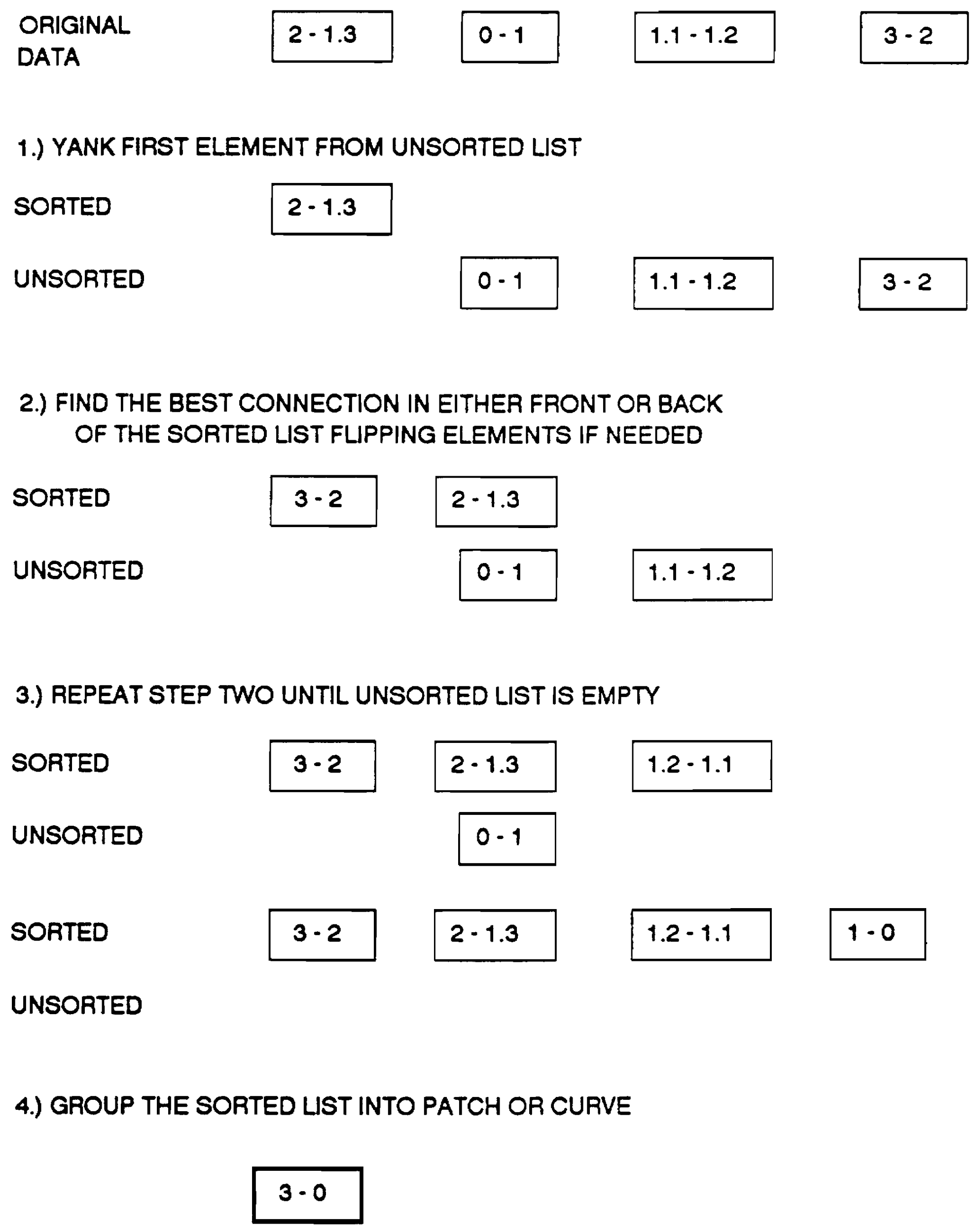

ORIGINAL DATA
2 - 1.3
0 - 1
1.1 - 1.2
3 - 2
1.) YANK FIRST ELEMENT FROM UNSORTED LIST
SORTED
2 - 1.3
UNSORTED
0 - 1
1.1 - 1.2
3 - 2
2.) FIND THE BEST CONNECTION IN EITHER FRONT OR BACK OF THE SORTED LIST FLIPPING ELEMENTS IF NEEDED
SORTED
3 - 2
2 - 1.3
UNSORTED
0 - 1
1.1 - 1.2
3.) REPEAT STEP TWO UNTIL UNSORTED LIST IS EMPTY
SORTED
3 - 2
2 - 1.3
1.2 - 1.1
UNSORTED
0 - 1
SORTED
3 - 2
2 - 1.3
1.2 - 1.1
1 - 0
UNSORTED
4.) GROUP THE SORTED LIST INTO PATCH OR CURVE
3 - 0

Figure 22. The Sorting Algorithm

8.0 Filleting

A fillet is an intermediate surface which blends two intersecting surfaces to form one continuous surface. Preliminary aircraft design codes require fillets to blend adjoining surfaces with C^2 continuity. Gloudemans [Glou90] proved the feasibility of joining two non-uniform bi-cubic B-Spline surfaces with a fillet surface that guarantees C^2 continuity. The following sections discuss the filleting algorithm developed in this thesis. First, the algorithm will be defined for filleting two curves. C^2 continuity at the curve blend point will also be verified. Secondly, an extension to this algorithm for filleting two surfaces along iso-parametric (one-dimensional filleting) curves will be discussed. A design concept combining the work of Gloudemans and the author of this thesis for parametric corner filleting is defined in the recommendations section of this thesis.

8.1 Curve Filleting

The equation for computing a point along a uniform cubic B-Spline curve is:

$$P_i(u) = UMQ_i$$

Where:

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \quad M = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \quad Q_i = \begin{bmatrix} q_{i-1} \\ q_i \\ q_{i+1} \\ q_{i+2} \end{bmatrix}$$

It has been shown that any break point along the curve is a weighted average of only three control vertices. Evaluating the first and second derivative of this equation with respect to u gives:

$$\begin{aligned} p(0)_i &= \frac{1}{6} q_{i-1} + \frac{2}{3} q_i + \frac{1}{6} q_{i+1} && \textit{position} \\ p_u(0)_i &= -\frac{1}{2} q_{i-1} + \frac{1}{2} q_{i+1} && \textit{slope} \\ p_{uu}(0)_i &= q_{i-1} - 2q_i + q_{i+1} && \textit{curvature} \end{aligned}$$

The above equations verify that only the control vertices q_{i-1}, q_i and q_{i+1} affect the position, slope and curvature of a cubic B-Spline curve at the curve segment break points. This demonstrates that two curves can be joined to form a C^2 continuous blend providing they share the same three control vertices at the blend point. Figure 23 on page 78 illustrates the blending of two intersecting curves with an intermediate curve.

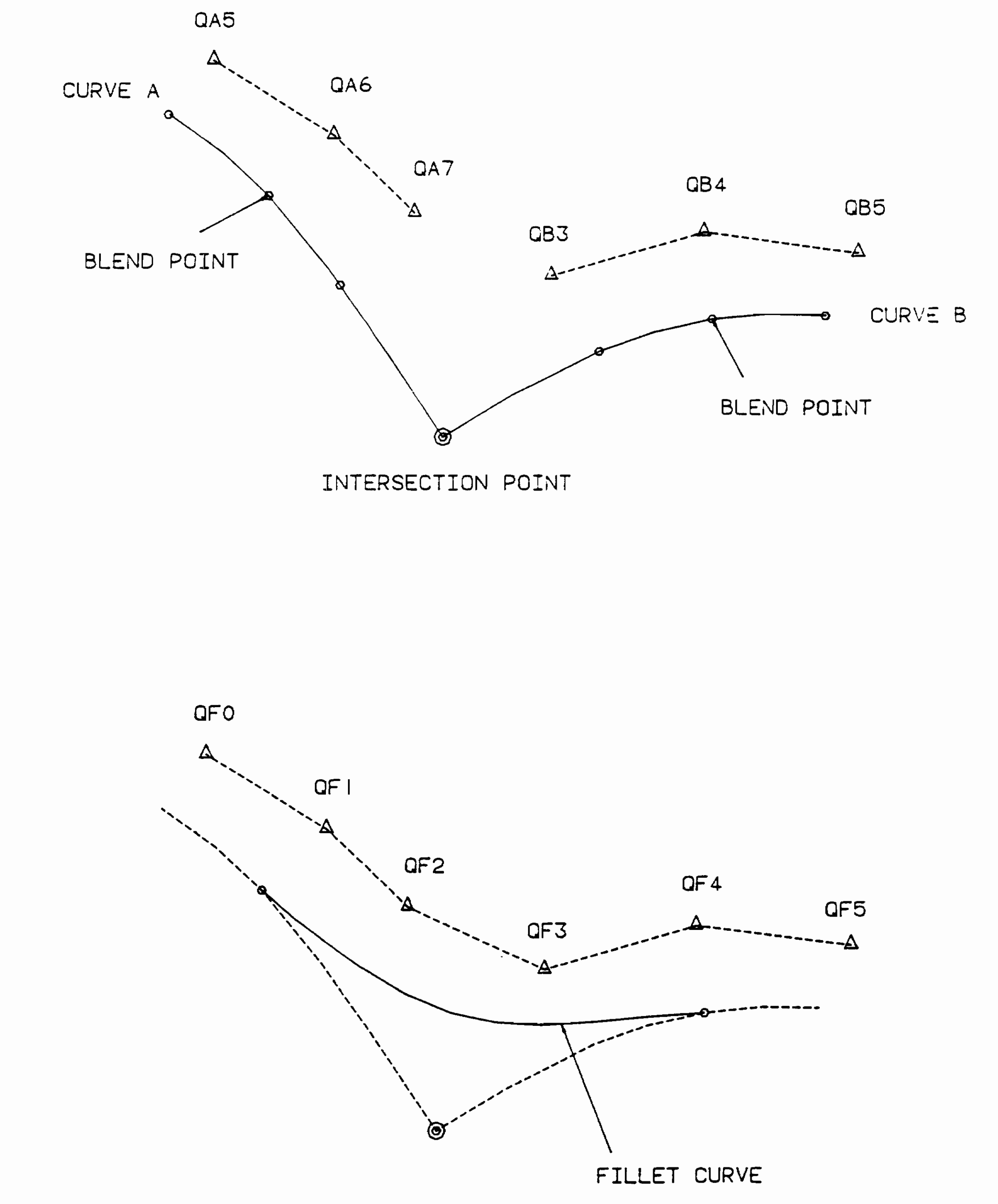


Figure 23. Simple Curve Fillet

8.1.1 Forcing a Break Point

The algorithm described above is limited to blending two intersecting curves at a curve segment break point. Most often, it will be desired to blend two curves at a location other than a break point. The use of knot insertion allows a curve segment break point to be created at any location along the curve. Thus, knot insertion will allow the curves to be blended at any location.

The curve filleting algorithm was described using uniform cubic B-Spline curves. Inserting a knot into the knot sequence results in a non-uniform cubic B-Spline curve. The position, slope and curvature of a non-uniform cubic B-Spline curve are a function of four control vertices and six knot values. However, similar to the uniform curve, the position, slope and curvature at a curve segment break point are a function of only three control vertices and five knot values.

Blending two non-uniform cubic B-Spline curves involves matching the knot spacing of five knot values while sharing the same three control vertices at the blend point. The best way of achieving this condition is to insert two knots on either side of the forced break point. The spacing of the knots is irrelevant as long as it matches the spacing of the other curve being blended with it. The algorithms used in this research force a uniform knot spacing on both sides of the forced break point.

8.1.2 Shape Control

The objective of this section is to discuss how a conic definition can be used to control the shape of the intermediate fillet curve. An example of the conic definition is shown in Figure 24 on page 81. The conic is defined using three points (A, B, C) and a parametric value u. A parameter value of zero corresponds to a straight line between points A and B, while a parameter value of one results in the tightest corner possible.

The conic representation is incorporated for shape control by using the point of intersection as point C in the conic definition. Points A and B are set equal to the break point at which the blend is to be made. The parameter value is now used to control the shape of the conic between the two intersecting curves. Points along the conic are used as interpolation points along with the three control vertices on each curve as input for "Fixed Control Vertex Inversion" [Glou90].

Fixed control vertex inversion is an extension to the inversion algorithm discussed earlier. In fixed control vertex inversion, actual control vertices are flagged for protection against inversion. Thus, known control vertices can be used along with actual interpolation points to control the shape of the curve while guaranteeing a C^2 continuous blend at the end points.

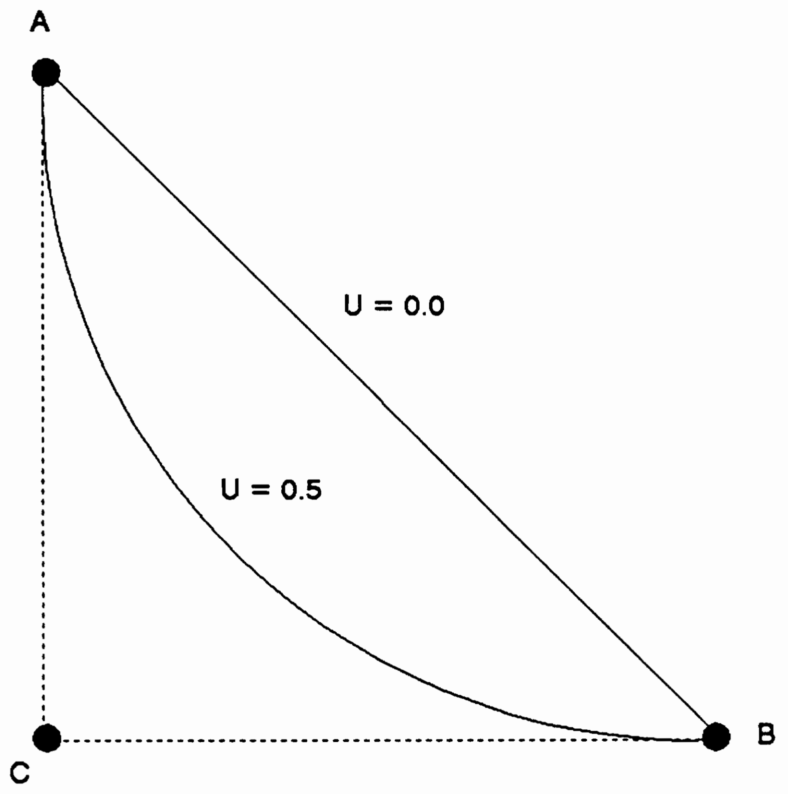


Figure 24. Example Conic Definition

8.1.3 Point Selection

A poor selection of points along the conic can result in unwanted inflections in the fillet curve. Each end of the fillet curve is controlled by three control vertices and five uniformly spaced knots. The interior knot spacing is determined based on the chord length between the points to be interpolated. However, knots t_1 thru t_5 must be uniformly spaced. The relationships between knots, control vertices and points along the curve (Figure 5 on page 28) show that the break points p_0 thru p_2 must be uniformly spaced to coincide with the chord length parameterization for determining knot values. Furthermore, the break point p_2 is the first point to be interpolated because the control points q_1 and q_2 are taken from the curve with which the fillet curve is to be blended. Figure 25 on page 83 illustrates this concept.

To ensure a smooth blend, the point spacing around the break point on the original curve is determined. The first point to be interpolated on the conic is at a distance equal to twice this point spacing from the blend point. The resulting intermediate break point p_1 will be the mid-point of the blend point and the first point to interpolate. Hence, the first three break points are uniformly spaced.

8.1.4 Curve Filleting Process

The following is a list of steps necessary to fillet (blend) two curves with a non-uniform cubic B-Spline curve:

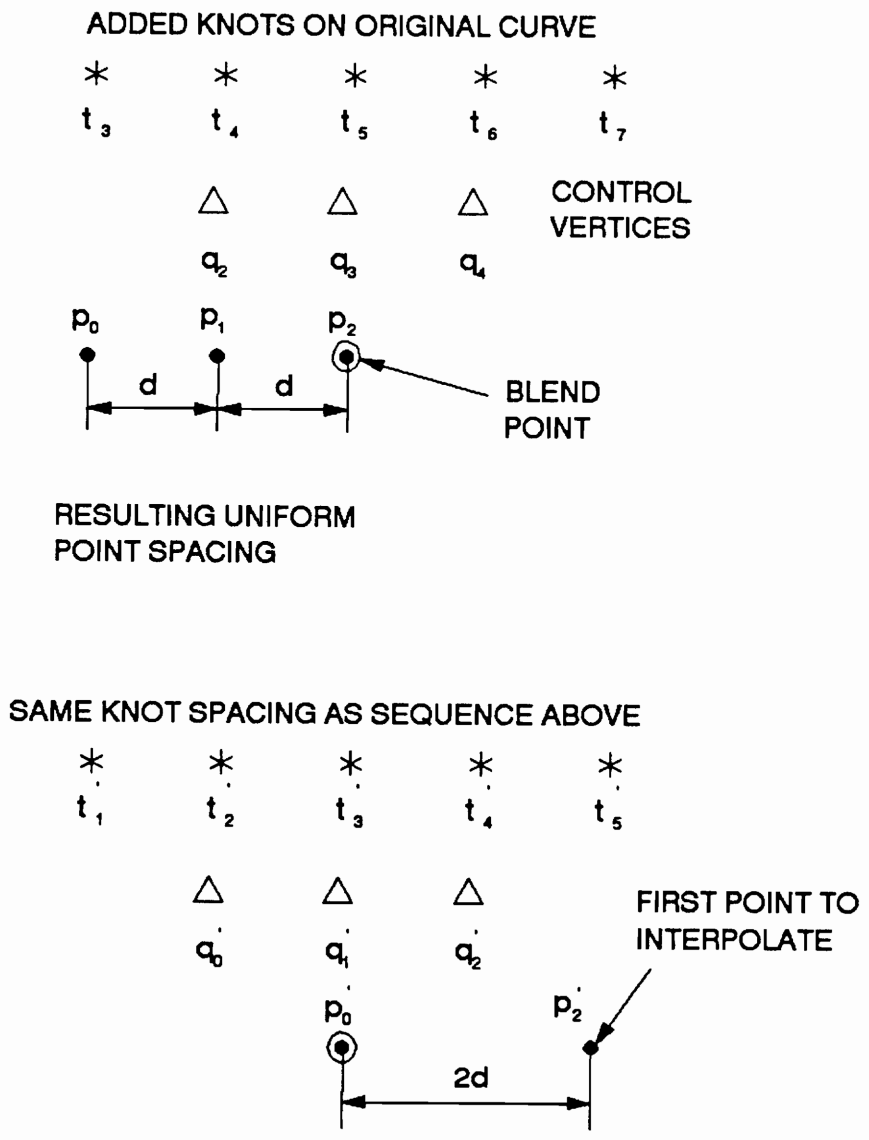

ADDED KNOTS ON ORIGINAL CURVE
t_3 t_4 t_5 t_6 t_7
CONTROL VERTICES
q_2 q_3 q_4
p_0 p_1 p_2
d
d
BLEND POINT
RESULTING UNIFORM POINT SPACING
SAME KNOT SPACING AS SEQUENCE ABOVE
$\dot{t}_1$ $\dot{t}_2$ $\dot{t}_3$ $\dot{t}_4$ $\dot{t}_5$
$\dot{q}_0$ $\dot{q}_1$ $\dot{q}_2$
FIRST POINT TO INTERPOLATE
$\dot{p}_0$ $\dot{p}_2$
2d

Figure 25. Point Selection for Curve Filleting

1. Use the subdivision algorithm discussed earlier to compute the point of intersection between both curves.

2. Define an offset distance from the point of intersection for the fillet curve to join both intersecting curves.

3. Insert knots at the blend point, if needed, on both curves to force a curve segment break point.

4. Define a conic between the intersecting curves using the point of intersection as point C. Set points A and B equal to the blend points of the fillet.

5. Vary the parameter value u for the conic until the desired shape is achieved between the curves.

6. Select points along the conic as interpolation points for the B-Spline curve.

7. Use the interpolation points along with the interior three control vertices at each blend point as input for the fixed control vertex inversion process discussed above.

8.2 One-Dimensional Filleting

One-dimensional filleting refers to blending two intersecting surfaces in one parametric direction along iso-parametric curves. Figure 26 on page 86 illustrates the concept of an iso-parametric curve. An iso-parametric curve is a curve produced on a surface patch for a given constant value of the parameter u or w. In this thesis the term iso-parametric curve will refer to the curve produced at a constant u or w parameter that is equal to a knot value in the knot vector. Two intersecting surfaces can be blended along iso-parametric curves in the same manner as the curve filleting algorithm defined earlier.

The position, slope and curvature of any point on a bi-cubic B-Spline surface are a function of nine control vertices and five knot values in both parametric directions. Hence, guaranteeing a C^2 continuous blend requires that the surfaces share nine control vertices for each blend point. Additionally, the knot spacing over a span of five knots in both parametric directions must be proportional.

One-dimensional filleting requires a blend point and conic definition to be computed along each iso-parametric curve involved in the filleting. Each blend point is calculated independent of the other iso-parametric curves. The farthest parametric value is then used as the base for computing the break point for each surface. Similar to the curve fillet, a uniform knot spacing is forced around the break point. The uniform knot spacing is computed based on the knot spacing around the farthest blend point.

8.2.1 Alignment of Iso-Parametric Curves

Blending two surfaces along iso-parametric curves requires the surfaces to have the iso-parametric curves in the direction of the fillet aligned with each other to ensure against unwanted inflections in the fillet surface. More importantly, the knot spacing in the other parametric direction must be proportional.

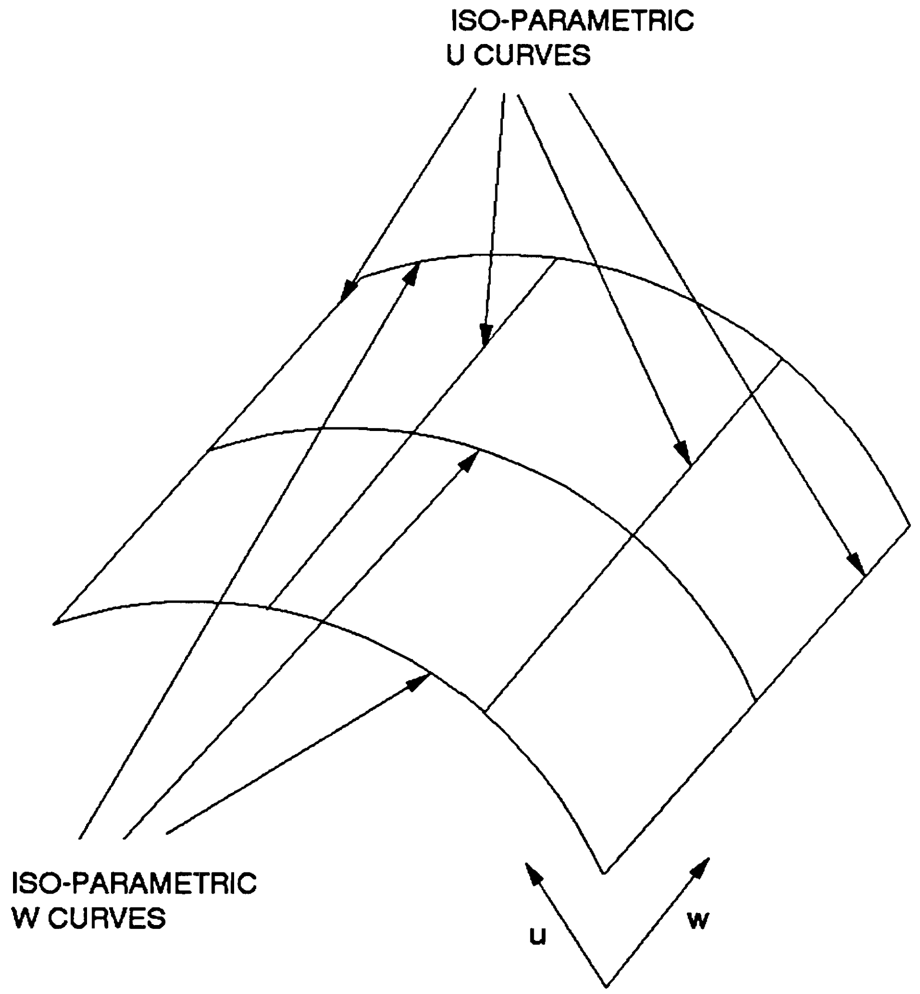


Figure 26. Iso-Parametric Curves

It is often necessary to insert knots into one or both of the intersecting surfaces to prepare them for filleting. Shown in Figure 27 on page 88 are two intersecting surfaces that do not have aligned iso-parametric curves for filleting. It can be easily realized that two knots must be inserted on the first surface while one knot must be inserted on the second surface.

It was mentioned earlier that chord length parameterization was used for inverting the surface geometry. If both surfaces have been parameterized using chord length parameterization, matching the ratio of individual knot spacing to total knot spacing between the surfaces will result in aligned iso-parametric curves. This process will meet the knot spacing requirements for a C^2 continuous blend while also aligning the iso-parametric curves to ensure against unwanted inflections in the blend surface.

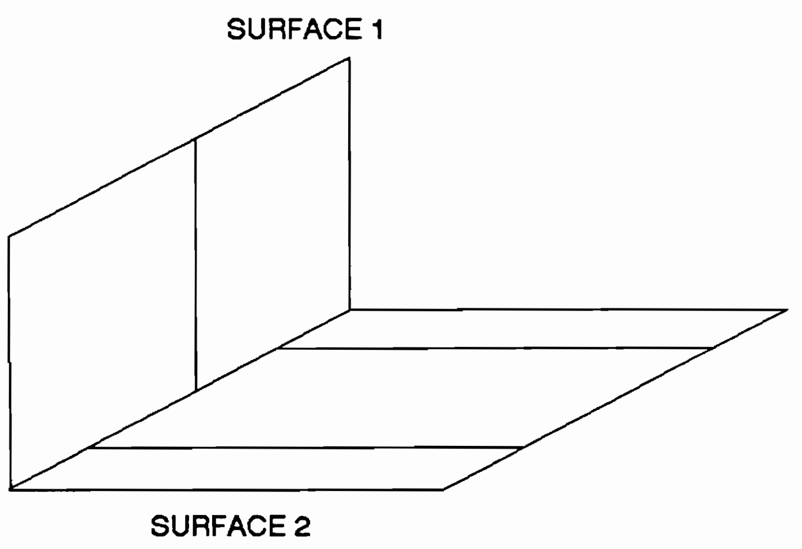
To illustrate the importance of chord length parameterization, refer to Figure 28 on page 89. The figure displays the same two intersecting surfaces, once with chord length parameterization and once with centripetal parameterization. Using the chord length parameterization gives a total knot spacing of 3 for both surfaces. The centripetal method has a span of 3 for the surface A and a span of 2.414 for surface B.

Comparing the ratio of an individual knot span to the total knot span using the chord length parameterization technique gives:

$$A_1 = \frac{t_4^A - t_3^A}{t_5^A - t_3^A} = \frac{1}{3} = B_1 = \frac{t_4^B - t_3^B}{t_6^B - t_3^B} = \frac{1}{3}$$

$$A_2 = \frac{t_5^A - t_4^A}{t_5^A - t_3^A} = \frac{2}{3} \neq B_2 = \frac{t_5^B - t_4^B}{t_6^B - t_3^B} = \frac{1}{3}$$

A knot must be inserted on surface A at:



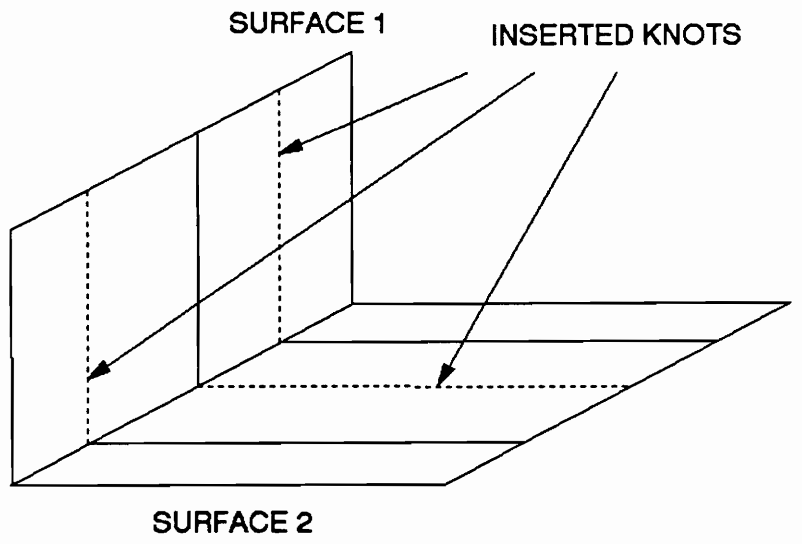
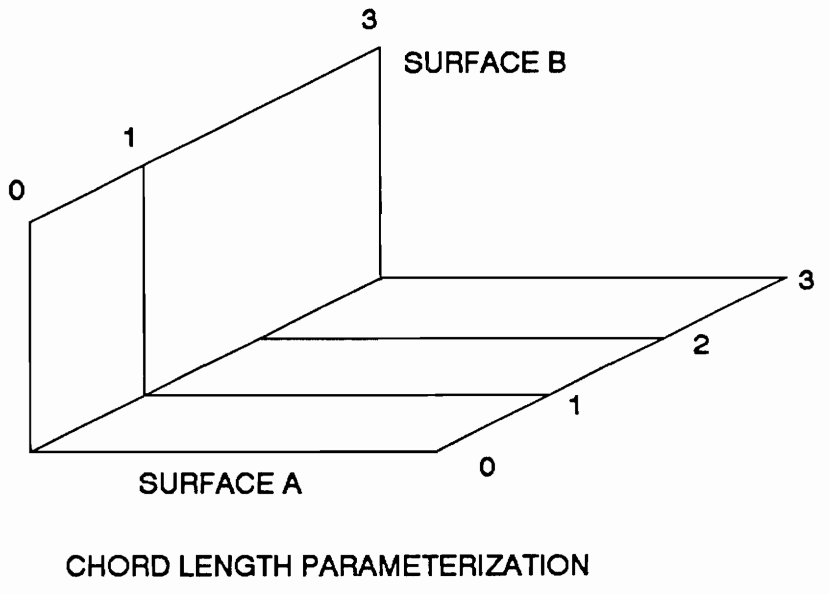


Figure 27. Misaligned Iso-Parametric Curves



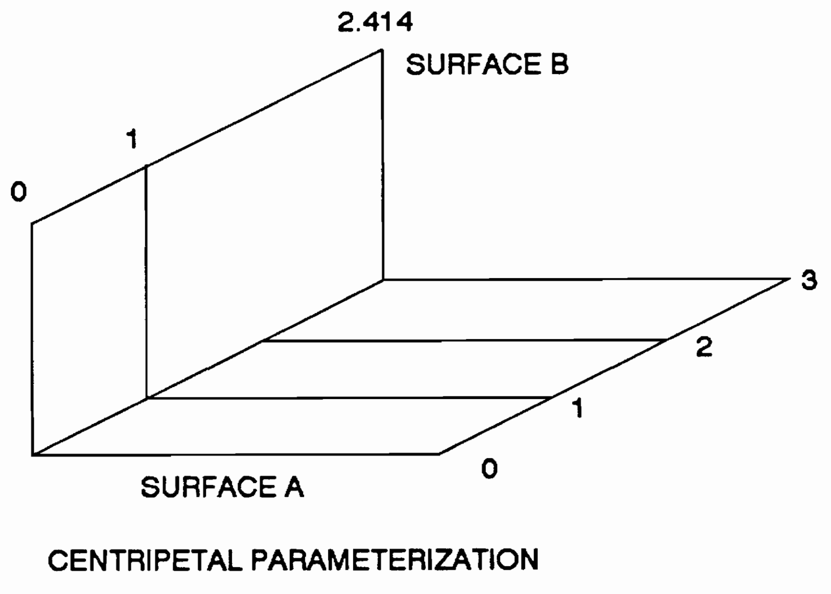


Figure 28. Chord versus Centripetal Parameterization

$$t_{new}^{A} = (t_5^A - t_3^A) \bullet \frac{t_5^B - t_4^B}{t_6^B - t_3^B} + t_4^A = 3 \bullet \frac{1}{3} + 1 = 2$$

The knot sequences for both surfaces are matched. The inserted knot in surface A aligns the iso-parametric curves and the surfaces are now ready for filleting.

Performing the same comparison for the centripetal parameterization gives:

$$A_1 = \frac{t_4^A - t_3^A}{t_5^A - t_3^A} = \frac{1}{2.414} \neq B_1 = \frac{t_4^B - t_3^B}{t_6^B - t_3^B} = \frac{1}{3}$$

According to the above calculation, a knot must be inserted on surface B between t_3 and t_4. This clearly adds an extra iso-parametric curve. Carrying out the ratio matching for the centripetal parameterization gives a knot sequence on surface A of (0, 1.0, 1.24, 2, 3) and a knot sequence on surface B of (0, 0.8, 1.0, 1.61, 2.41). These two knot vectors meet the proportional knot spacing requirements. However, the resulting iso-parametric curves will not be aligned which can cause shape problems on the fillet surface. Figure 29 on page 91 illustrates the resulting iso-parametric curves for the ratio matching routine with centripetal parameterization.

8.2.2 The One-Dimensional Filleting Process

The process of filleting (blending) two surfaces along iso-parametric curves can be divided into the steps listed below. An example one-dimensional fillet is also included. The surfaces are filleted in the parametric w direction.

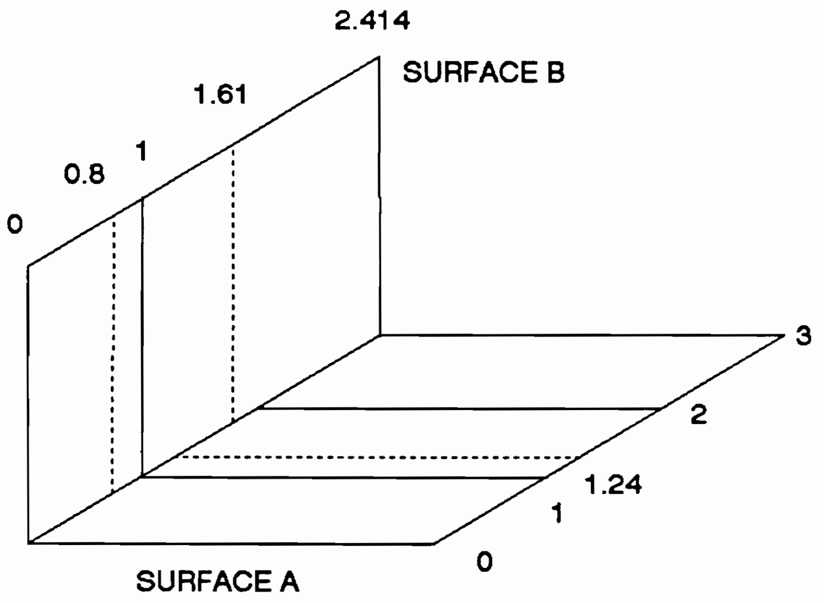


Figure 29. Ratio Matching with Centripetal Parameterization

1. Compute the curve of intersection using the subdivision method described earlier.

2. Define an offset distance from the curve of intersection for the fillet surface to join both intersecting surfaces.

3. Use the offset distance to compute the blend point on both surfaces for each iso-parametric curve involved in the fillet (Figure 30 on page 93).

4. Insert five knots uniformly spaced from the largest or smallest parametric w value on both surfaces (Figure 31 on page 94).

5. Define a conic between the blend points along all iso-parametric curves involved in the fillet (Figure 32 on page 95).

6. Select points along each of the conics as interpolation points for the B-Spline surface.

7. Use the interpolation points and the three control vertices on both surfaces along each iso-parametric curve as interpolation points for the fixed control vertex inversion. Additionally, three extra control vertices on the outside edges of both surfaces must be added as inputs for the inversion process (Figure 33 on page 96).

The completed fillet is shown in Figure 34 on page 97. The fillet surface is shown in solid lines while the original surfaces are shown in dashed lines. The original surfaces have also been trimmed to remove the sections which have been replaced with the fillet surface.

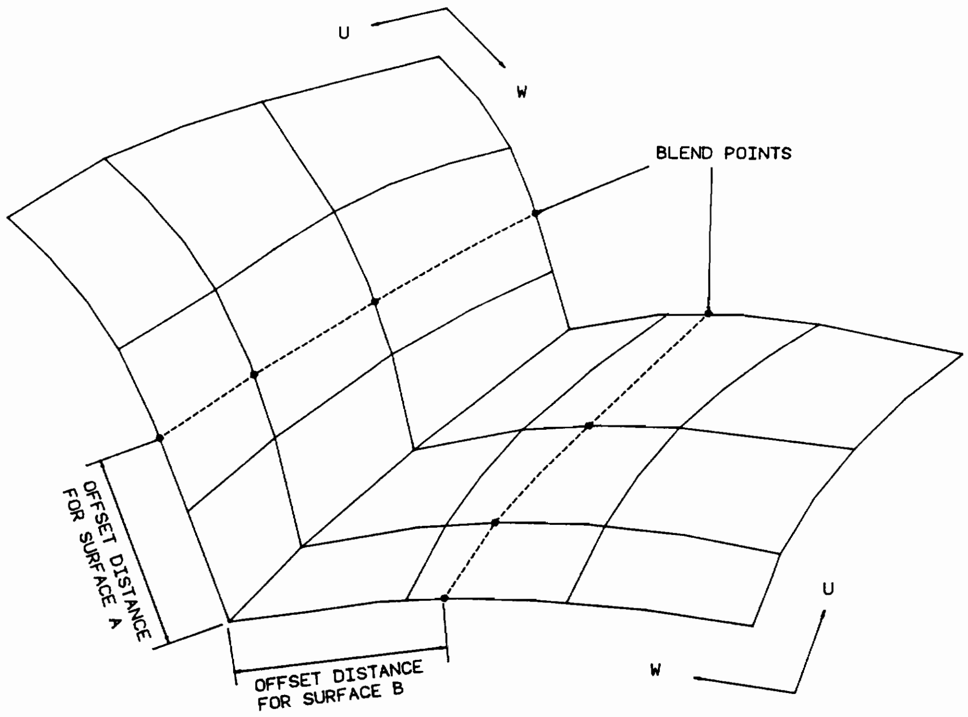


Figure 30. Computing the Blend Points for a 1-D Fillet

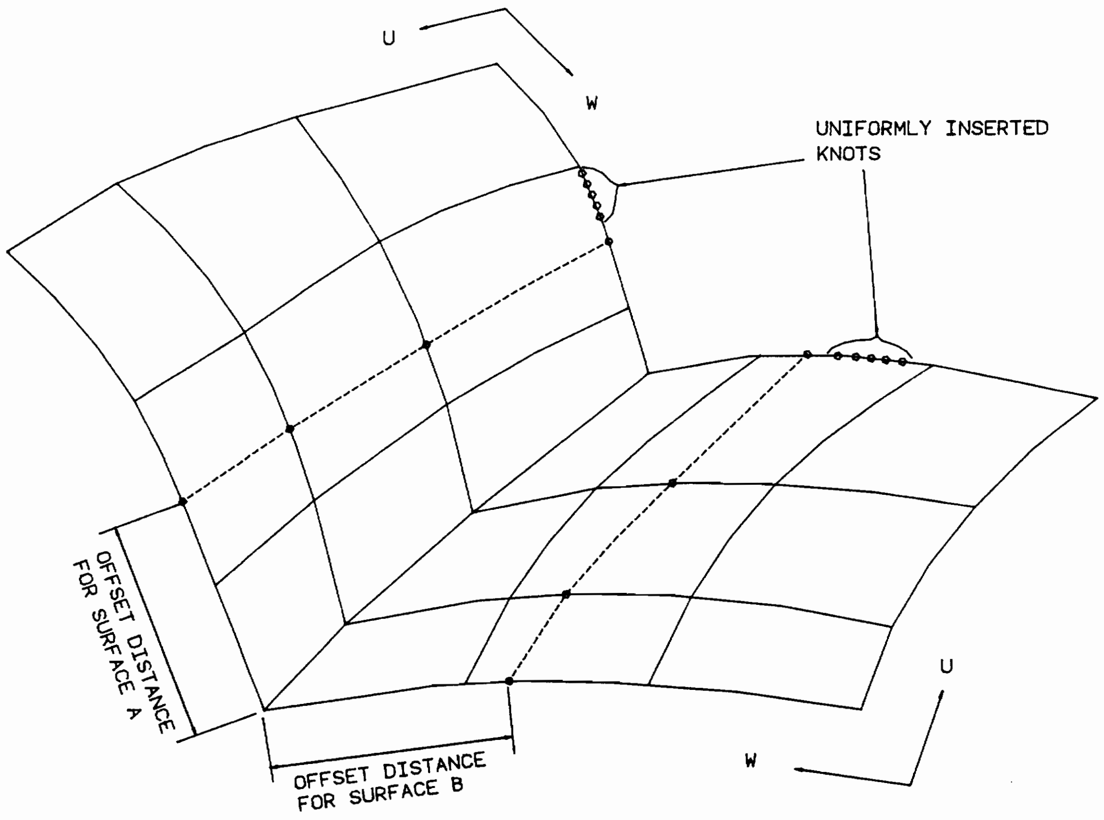


Figure 31. Forcing Uniform Knot Spacing at Blend

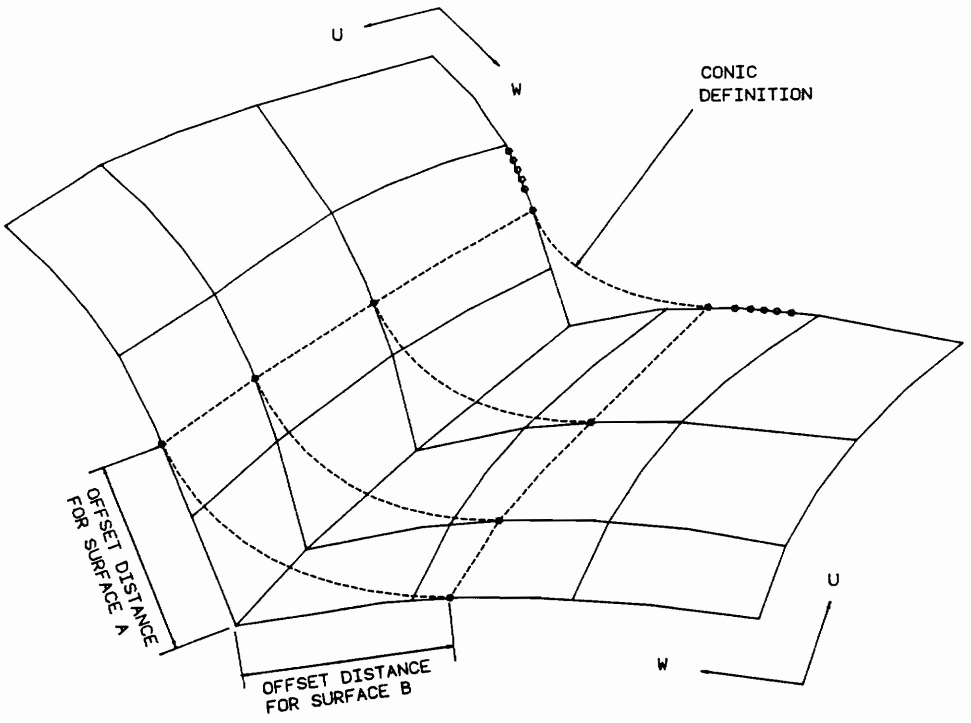


Figure 32. Defining a Conic for 1-D Filleting

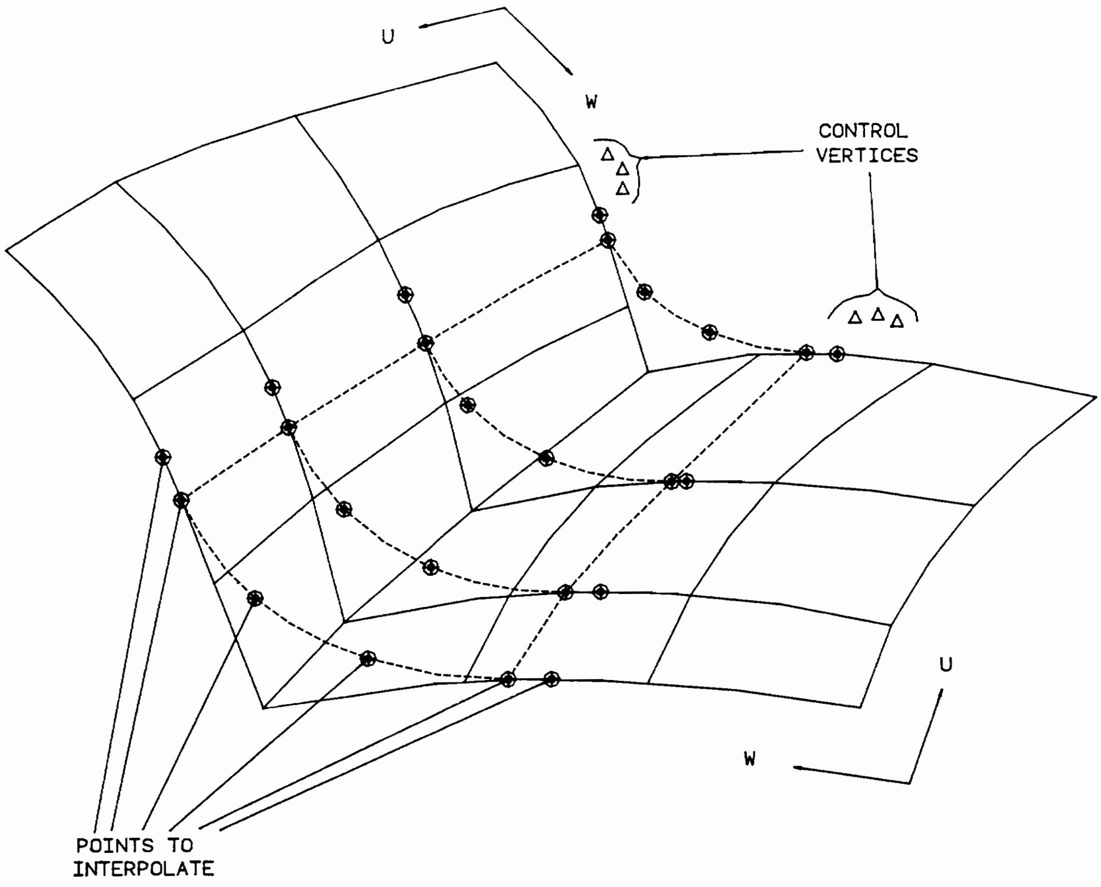


Figure 33. Points to Interpolate

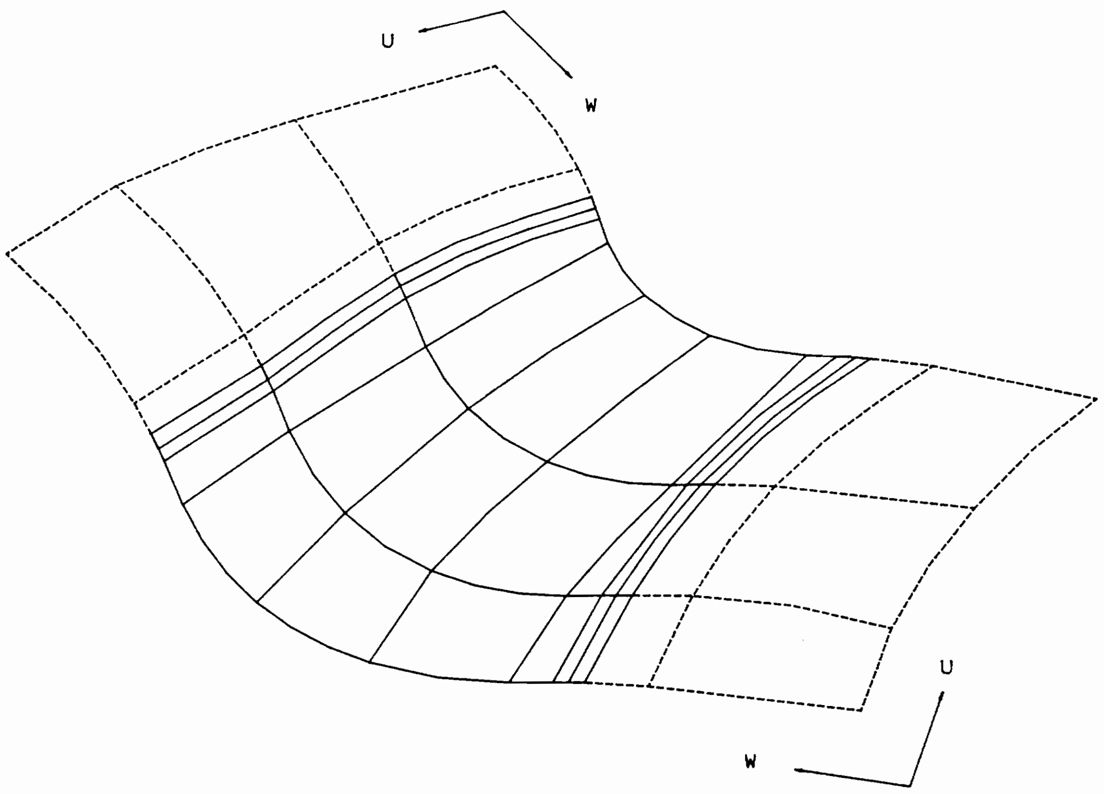


Figure 34. Completed One-Dimensional Fillet

9.0 Results

9.1 ACSYNT B-Spline Module

The design and coding of the ACSYNT B-Spline Module has been completed. The B-Spline Module can read files containing point data from ACSYNT and display the geometry using the bi-cubic Hermite or non-uniform bi-cubic B-Spline surfaces. The B-Spline Module has been released to all members of the ACSYNT Institute with the V1.1.0 release of ACSYNT on January 25, 1991. A utility function has also been included as part of the ACSYNT release to write out the ACSYNT geometry data in a form which can be read by the B-Spline Module.

The ACSYNT B-Spline Module is also being used by several graduate students at Virginia Tech as a development platform for their research in geometric modeling. A few of the research projects currently using the B-Spline Module include: the determination of parametric data for aircraft components (wing span, fuselage length, etc.) from arbitrary B-Spline surfaces and the testing of modeling algorithms.

9.2 Intersection of B-Spline Surfaces

The creation of the methods and software for computing the intersection between any two arbitrary non-uniform bi-cubic B-Spline surfaces has been completed. The intersection method has been successfully tested on several arbitrary B-Spline surfaces. Once the intersections have been computed, a linked list data structure is used to store the data for later display or for use in other modules (e.g. filleting).

The intersection method has been incorporated as part of the ACSYNT B-Spline Module. The intersection method can be accessed through menu items in the B-Spline Module (MAIN (GEOMETRY (B-SPLINE (INTERSECT)))). The user has the option of computing the intersection between all components in the model or selecting individual components to be used for the intersection algorithm. The user also has the option of displaying existing intersection data if it is not already on the screen.

Figure 35 on page 101 thru Figure 37 on page 103 illustrate the results of the intersection method. Each figure includes the maximum error in the intersection, the tolerance specified for making a planar approximation and the number of subdivisions performed. The real time to compute the intersection curve on an IBM RISC System 6000 Model 530 is also given.

The maximum error is determined by comparing each intersection point found on the first surface against the corresponding point on the second surface. The largest distance between two corresponding points is then reported as the maximum error of the intersection method. The actual point of intersection falls somewhere between the points on both surfaces, therefore, this is a conservative method of computing the error. The error value is a function of the specified tolerance for making a planar approximation and does not depend on the size of the surfaces involved in the intersection.

9.3 One-Dimensional Filleting

The creation of the one-dimensional filleting method has been completed. The algorithm is based on extensive user interaction. The user must specify the parametric direction of the fillet along both surfaces, the offset distance for the blend surface to join both surfaces and the parametric value for defining the conic. The parametric direction specifies the direction in which the filleting algorithm should move on both surfaces from the intersection curve. If a wrong direction is given, the routine will report errors to the user and the filleting process must be re-initiated. The offset distance and parametric value for defining the conic are defined in the discussion on the filleting technique.

Figure 38 on page 105 thru Figure 43 on page 110 illustrate the results of the one-dimensional filleting method. The results of each fillet are presented in two separate figures. A gray scale shaded image of the two intersecting surfaces is shown in the first

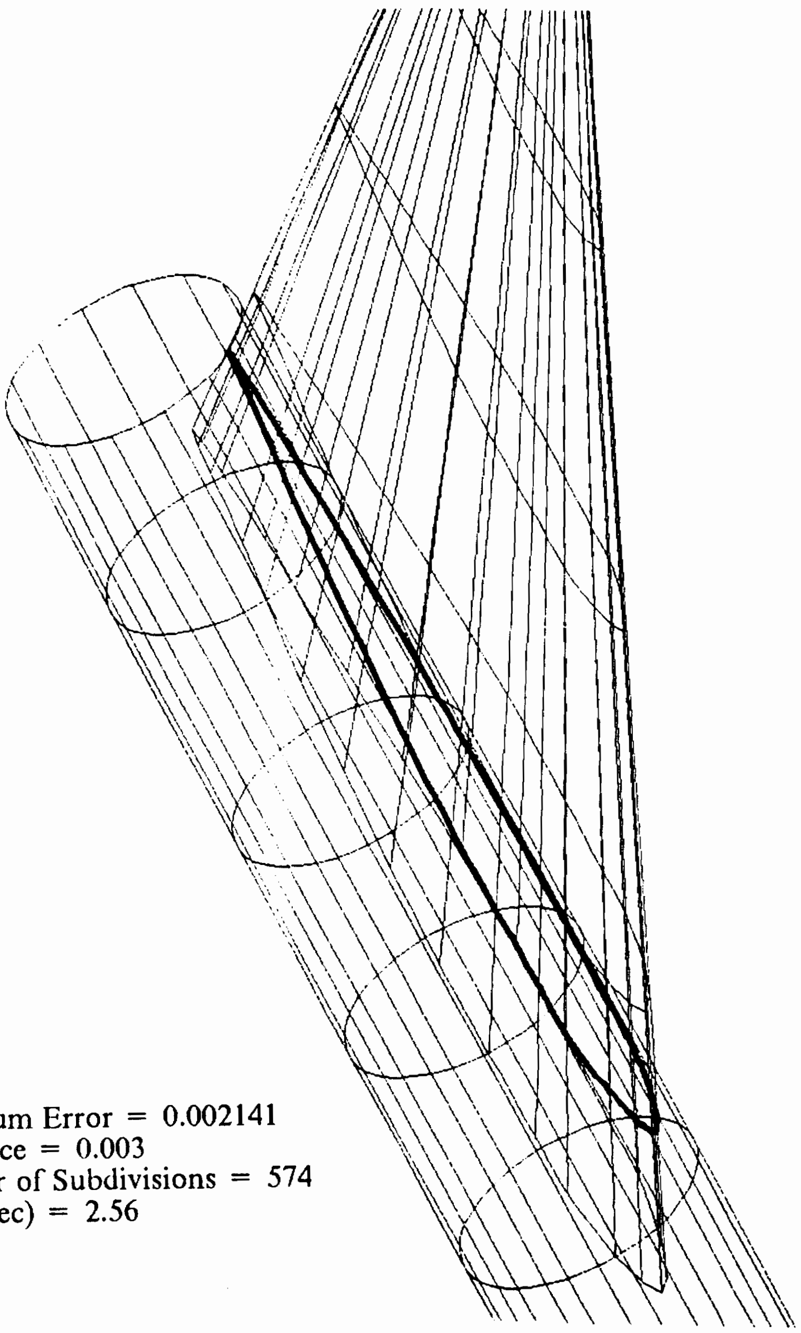
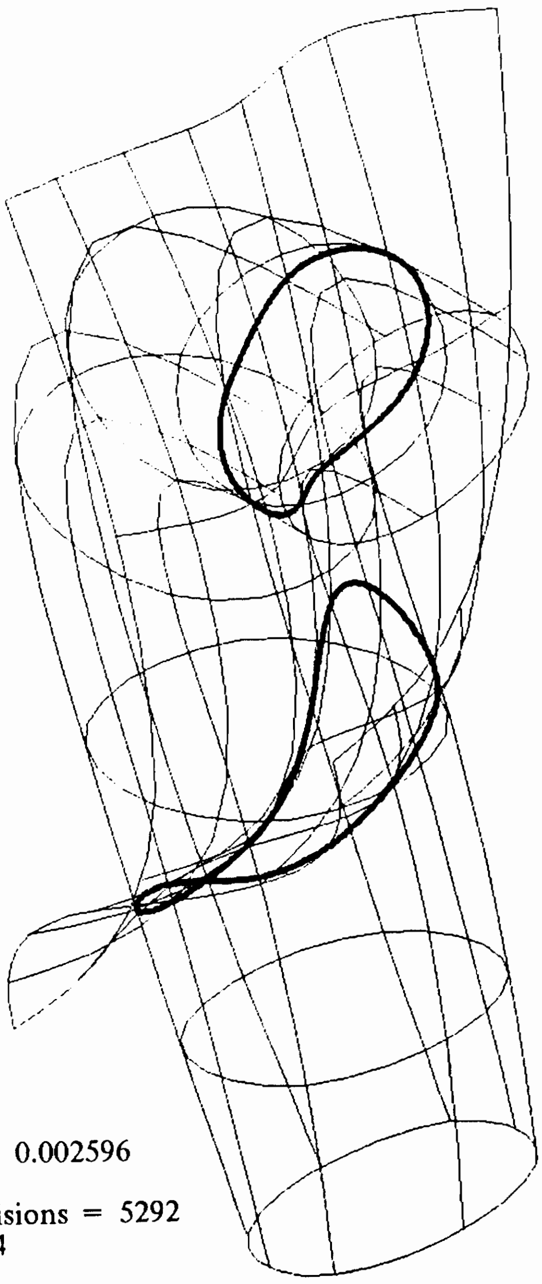
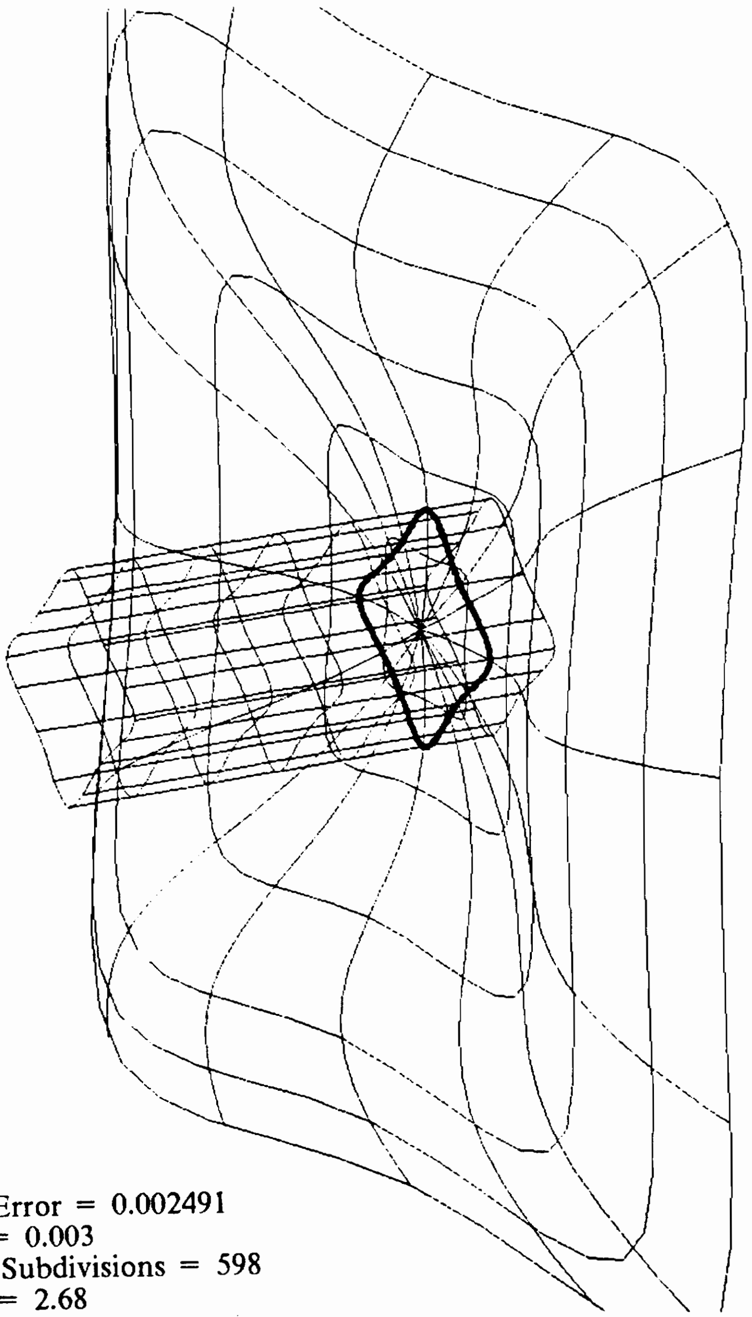

Maximum Error = 0.002141
Tolerance = 0.003
Number of Subdivisions = 574
Time (sec) = 2.56

Figure 35. Wing-Fuselage Intersection



Maximum Error = 0.002596
Tolerance = 0.003
Number of Subdivisions = 5292
Time (sec) = 16.54

Figure 36. Intersection of Two Arbitrary Surfaces



Maximum Error = 0.002491
Tolerance = 0.003
Number of Subdivisions = 598
Time (sec) = 2.68

Figure 37. Plate-Cylinder Intersection

of each set of figures. The resulting fillet is then shown as a shaded image in the following figure. The input parameters to produce each of the fillets shown are given in the B-Spline Module tutorial in Appendix A.

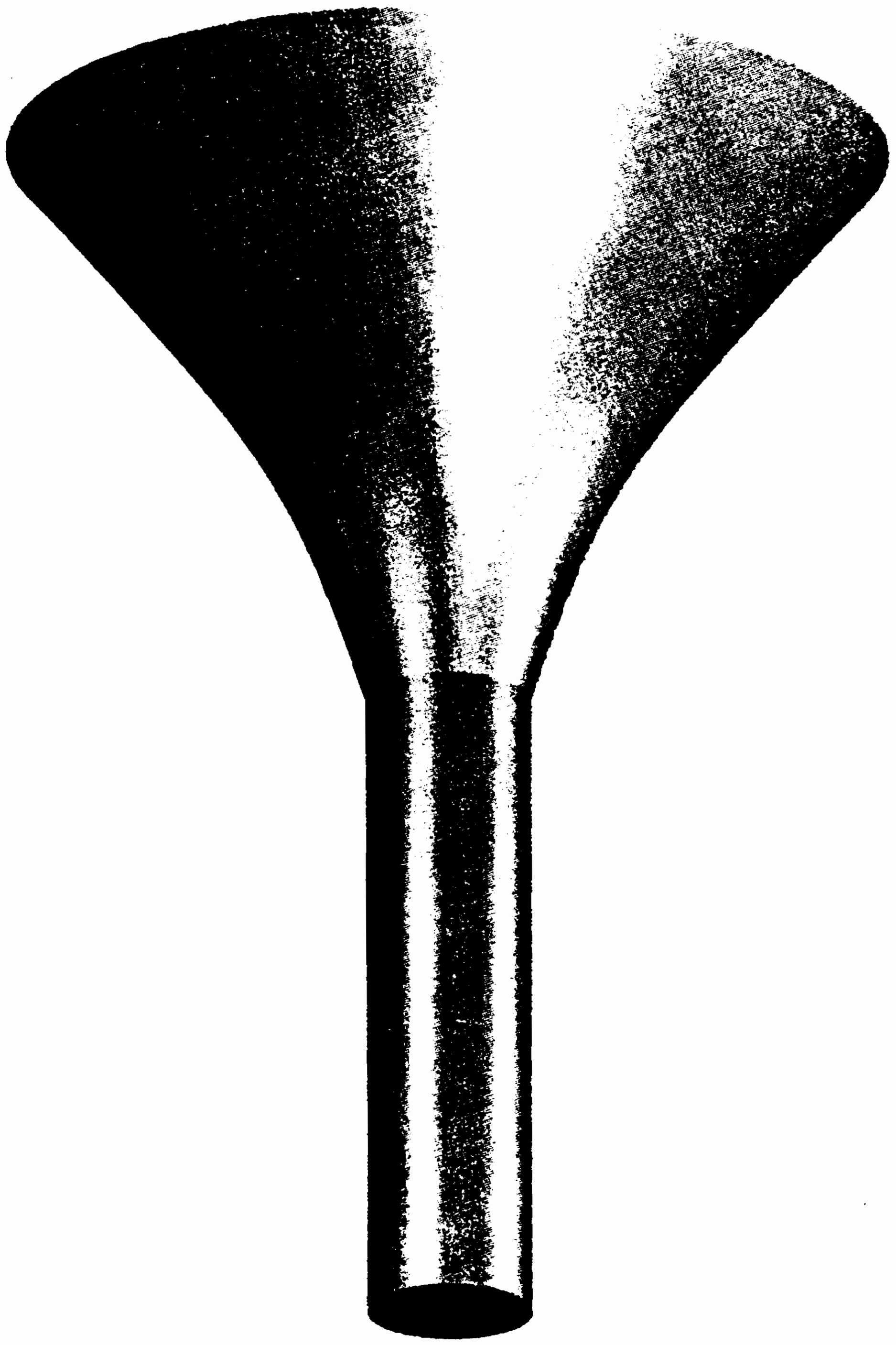


Figure 38. Cone and Cylinder Before Filleting

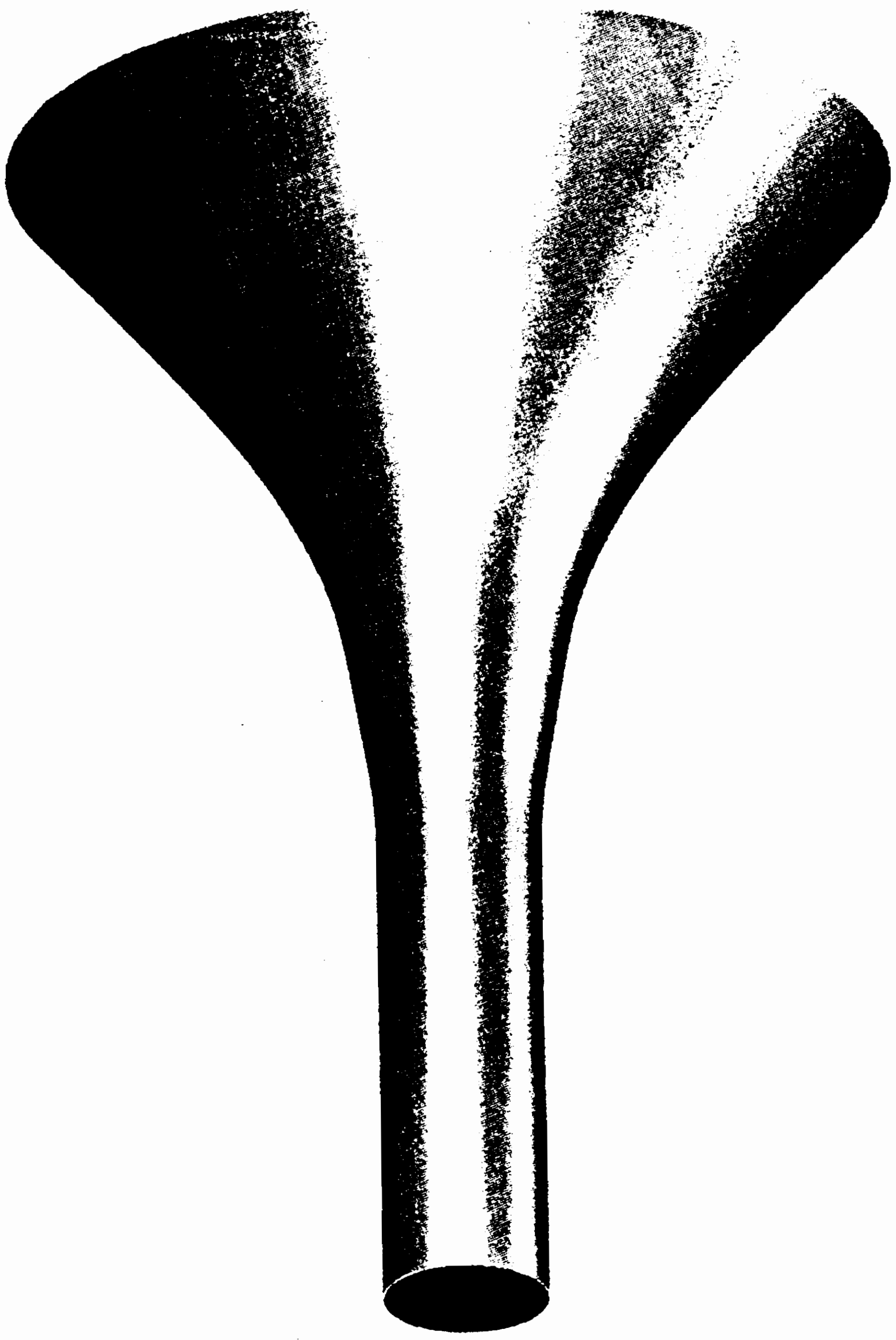


Figure 39. Filleted Cone and Cylinder

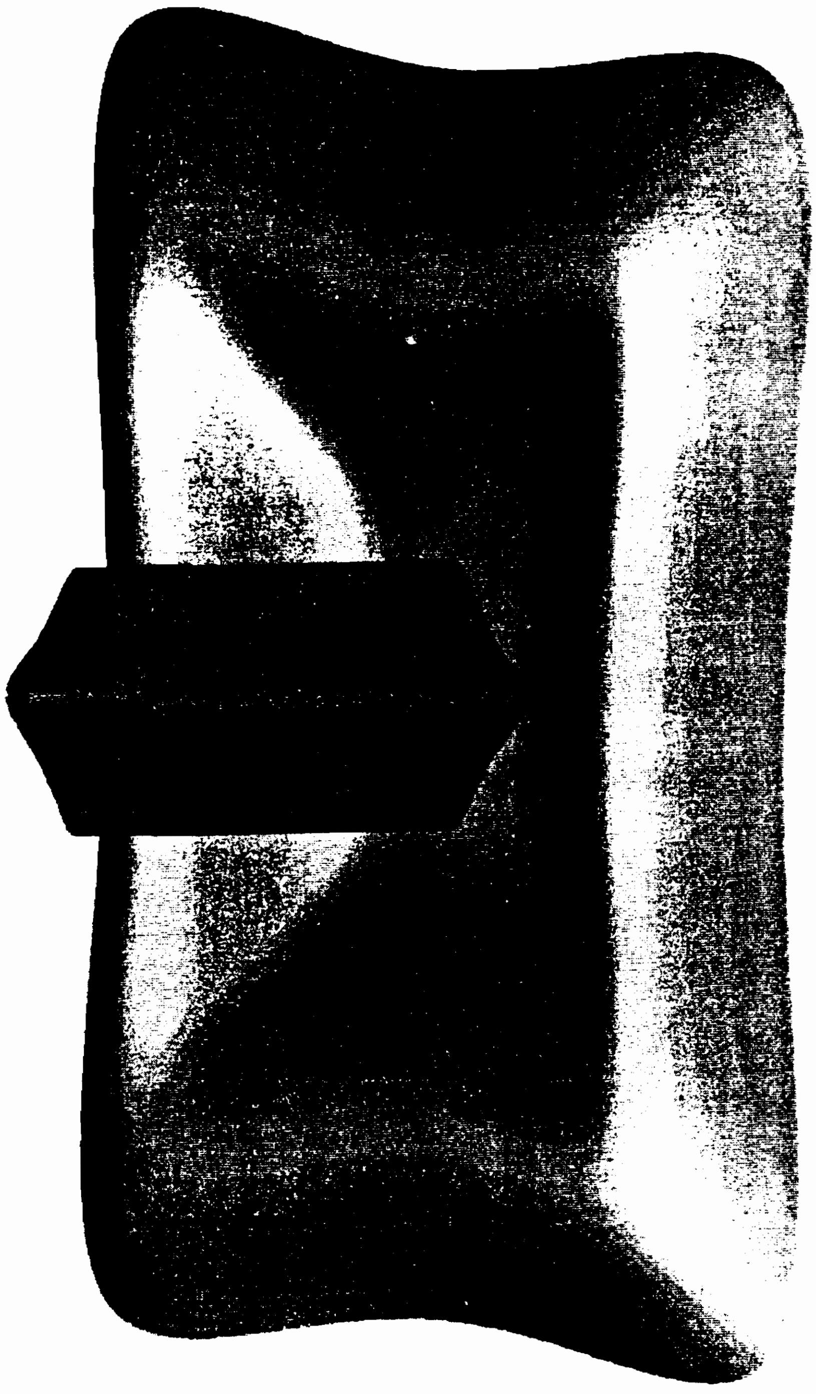


Figure 40. Plate and Tube Before Filleting

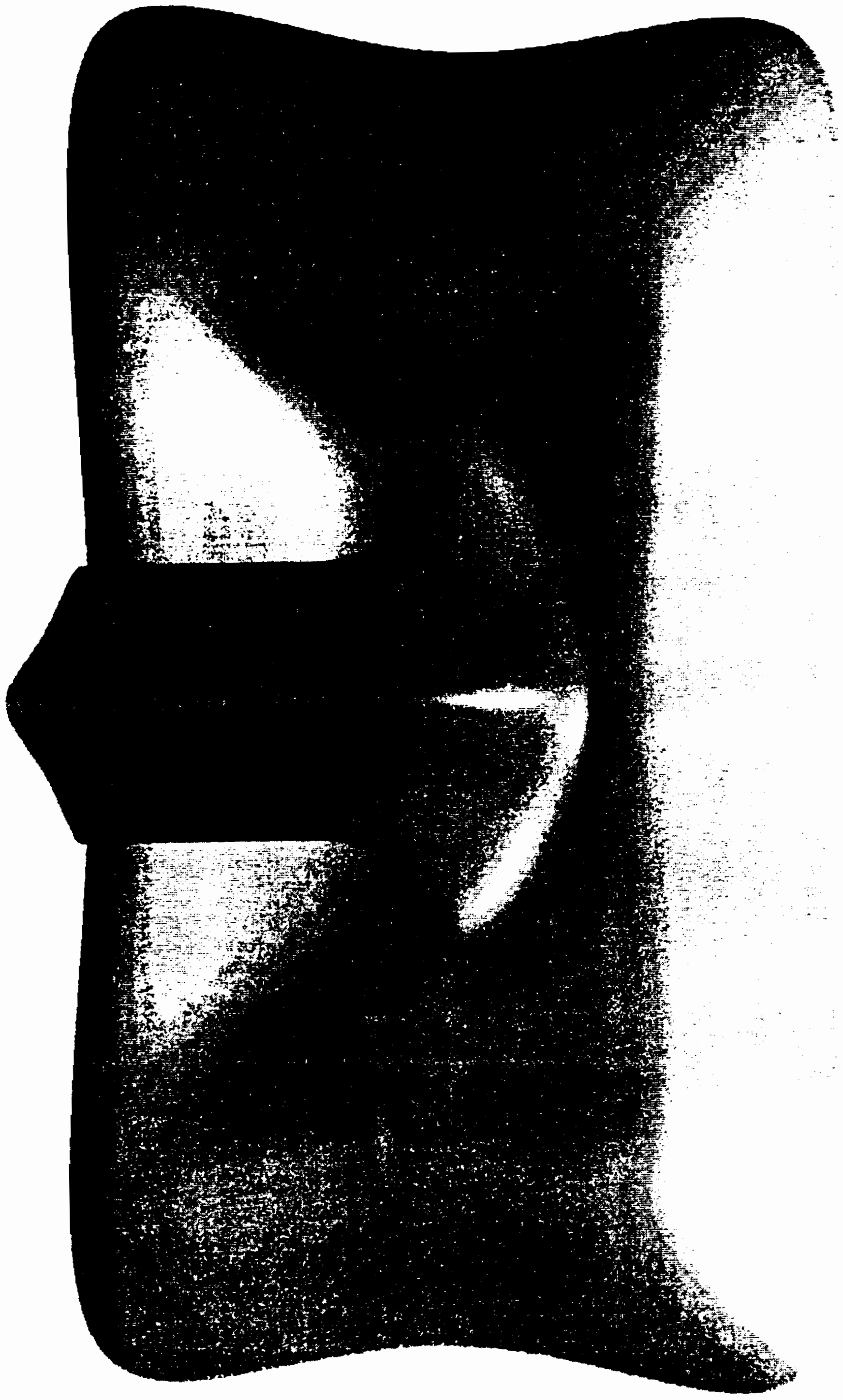


Figure 41. Filleted Plate and Tube

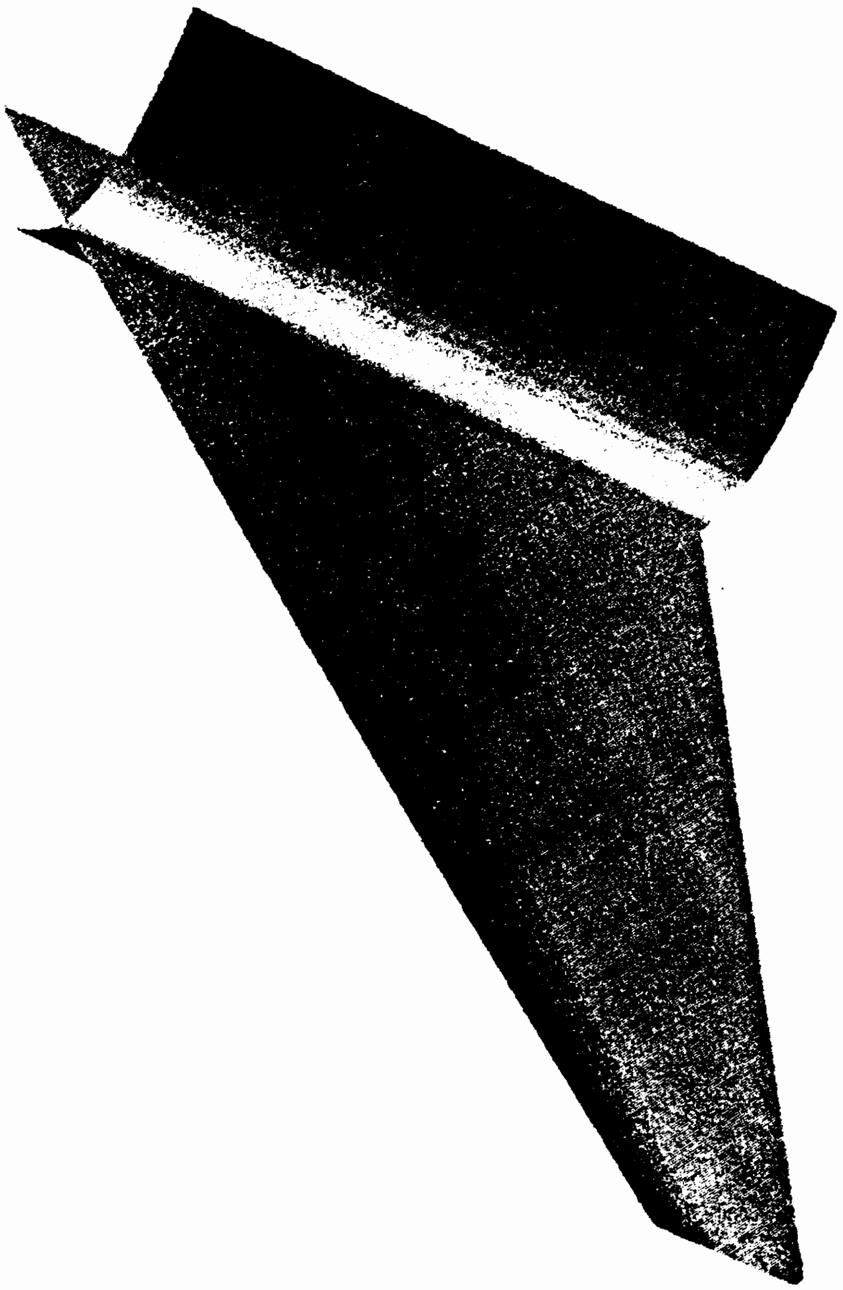


Figure 42. Example Fuselage and Wing Before Filleting

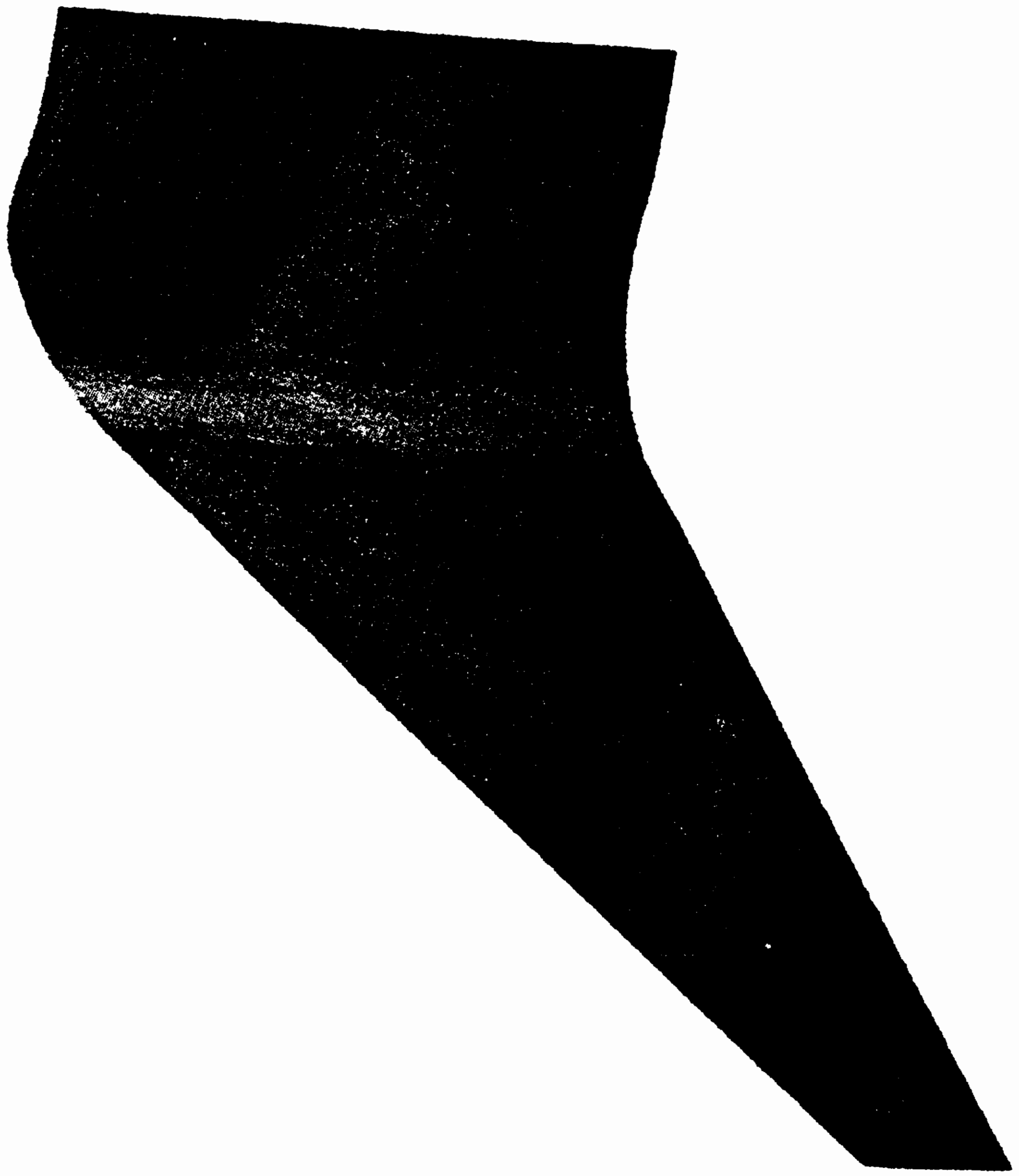


Figure 43. Filleted Fuselage and Wing

10.0 Recommendations

10.1 ACSYNT B-Spline Module

Although the ACSYNT B-Spline Module provides a robust interface for the conversion of ACSYNT data to non-uniform bi-cubic B-Spline surface representations, several topics need further investigation and development. The most important of these is the development of a shading module for B-Spline models. The shading algorithm should be developed to take full advantage of the hardware shading capabilities of most graphics workstations. In most cases, it will be necessary for the shading module to tessellate the B-Spline surfaces and compute polygons and polygon normals for the shading routines. The tessellation routine should allow the user to specify the number of polygons desired on the shaded image.

Research into the development of a method for creating models directly in the B-Spline surface format would enhance the capabilities of the B-Spline Module. The method used for designing components should allow users to build objects based on parametric data similar to the manner in which the ACSYNT main module creates Hermite data. The parametric design should be written as an interface to the B-Spline Module. In this manner parametric interfaces for several different design criteria could be integrated into the B-Spline Module. Cross-section design of components could also be added to the design module. The cross-section design should allow users to move control points on any specified cross section and see the resulting change to the surface interactively. Integration of these tools will allow designers to enter a few widely used parameters to get a basic configuration of their design. The cross section design module would then allow the designer to fine tune the design for future analysis or visualization.

10.2 Intersection of B-Spline Surfaces

The sorting routine used in the intersection method mimics the widely used bubble sort algorithm. This type of sorting is called an n^2 sort where n is the number of elements to be sorted. The utilization of the quad-tree data structure will greatly reduce the number of elements in any one sorting. Therefore, the quad-tree data structure would reduce the sorting time and speed up the overall time required to compute the curve of intersection between two B-Spline surfaces.

Another method for speeding up the intersection time is the development of a hunting algorithm once an intersection point has been found. Most hunting algorithms use subdivision techniques to find the first point of intersection. Once a point of intersection is found, numerical methods such as Newton's method are used to march across the intersection curve. These marching methods rely on good initial guesses for proper convergence.

10.3 Filleting of B-Spline Surfaces

The one-dimensional filleting algorithm described for blending two surfaces along iso-parametric curves must be extended to include a two-dimensional or corner filleting method. An algorithm for parametric corner filleting was designed and tested by Gloudemans [Glou90] at VPI&SU. This algorithm was verified using special user input for the blend points and matching of knot sequences.

The algorithm developed by Gloudemans isolated the parametric corner by inserting four knots in both parametric directions. The four knots force a break point at the parametric corner. Additionally, both sides of this break point are completely independent of each other. In this manner, changes to one side of the corner have no effect on the opposite side of the corner.

The following sequence of steps documents Gloudemans' parametric corner filleting algorithm along with additional research topics which must be tackled in order to produce a robust two-dimensional filleting method.

1. Compute the curve of intersection between both surfaces using the subdivision algorithm discussed in this thesis.

2. Write a ray tracing algorithm to determine the filleting direction along both surfaces. The algorithm could consist of defining a vector from the origin to any point along the surface. If the vector intersects the surface an odd number of times, the point is inside the surface and is in the wrong direction for filleting. If the vector intersects the surface an even number of times, the point lies along the proper filleting direction for that surface.

3. Once the filleting directions are determined, compute the blend curve where the fillet surface should blend both surfaces. An offset distance from the intersection curve can be used to compute this curve. The blend curve is shown in Figure 44 on page 116 [Glou90].

4. Define several conics along the intersection curve between the blend curves on both surfaces.

5. Compute the intermediate points along both surfaces up to the blend curve along with points along the conic definitions to provide shape control of the fillet. These points are shown in Figure 45 on page 117 for the example fillet [Glou90].

6. Invert the points in the parametric u direction only.

7. Add three knots at the parametric value of the fillet surface which corresponds to the corner of surface A.

8. Load the surface data of both components involved in the intersection into temporary data structures. This is done in order to manipulate the temporary data without destroying the original surface description.

9. Insert three knots at w = 2.0 into surface A in order to isolate the parametric corner.

10. Insert two uniformly spaced knots on either side of the parametric line u = 2.0 on surface A and store the control vertex values along the u = 2.0 line.

11. Repeat steps 9 and 10 for the parametric w line on surface A.

12. Insert three knots into surface B at u = 5.0 in order to match the number of control vertices at both ends of the fillet.

13. Add two uniformly spaced knots along the break line of surface B.

14. Store the fillet blend control vertices from surface B.

15. Invert the fillet surface along the parametric w direction using fixed control vertex inversion. The final example fillet is shown in Figure 46 on page 118.

The above algorithm assumes that the surfaces have similar knot sequences for blending. An intriguing area of research for producing a two-dimensional fillet is the pre-processing of the surfaces. A knot ratio matching method must be created to take into account the parametric corner on one of the surfaces. For instance, the original knot sequence involved in the preceding example fillet is 4.0, 3.0, 2.0, 3.0, 4.0. In this case, the knot sequence for the corner fillet first goes along the parametric w direction and then turns a corner and goes along the parametric u direction. This knot sequence would first be converted to a nondecreasing knot sequence (e.g. 0.0, 1.0, 2.0, 3.0, 4.0) before any ratio matching is performed.

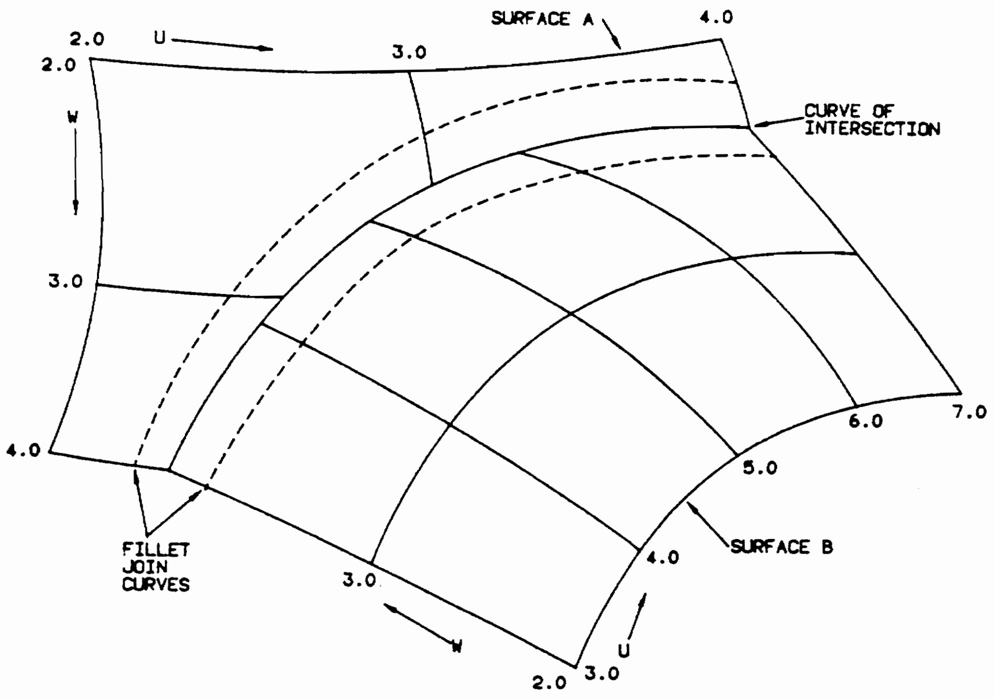


Figure 44. Parametric Corner Filleting - Step 1

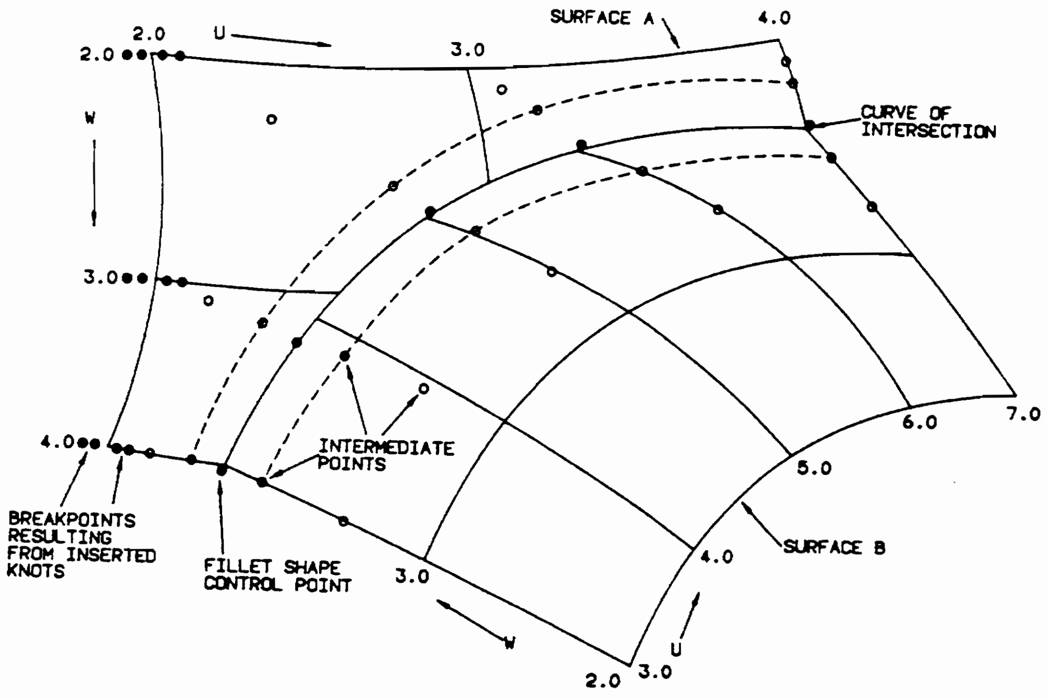


Figure 45. Parametric Corner Filleting - Step 2

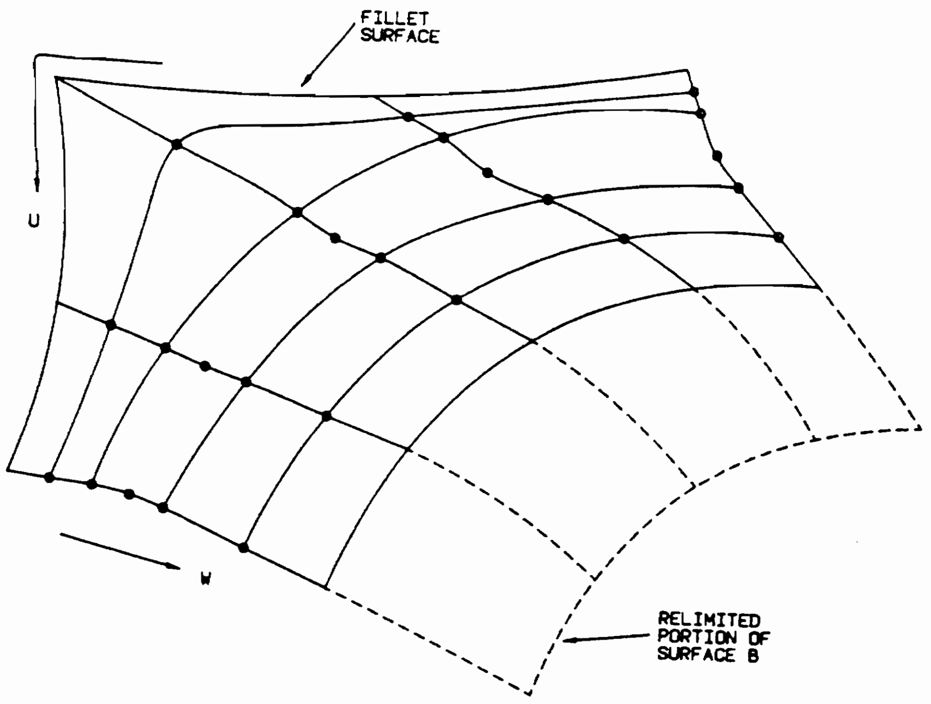


Figure 46. Completed Parametric Corner Fillet

The above mentioned steps include the research done by Gloudemans and the filleting method designed by Gloudemans and the author of this thesis. A robust method for producing parametric corner fillets is the final step in converting conceptual design geometry to a format that meets the requirements of preliminary design systems for aircraft design.

11.0 References

[Aho82] Aho, A., Hopcroft, J., Ullman, J., "Data Structures and Algorithms", *Addison-Wesley*, 1982.

[Aste88] Asteasu, C., "Intersection of arbitrary surfaces", *Computer Aided Design*, 20, No. 9, 1988, pp.533-538.

[Boeh80] Boehm, W., Hartmut, P., "Inserting new knots into B-Spline Curves", *Computer Aided Design*, 12, No. 4, 1980, pp.199-201.

[Boeh85] Boehm, W., "The Insertion Algorithm", *Computer Aided Design*, 17, No. 2, 1985, pp.58-59.

[Chen86] Chen, John J., Tulga, M., "An Intersection Algorithm For C^2 Parametric Surface", *Knowledge Engineering and Computer Modelling in CAD, Proceedings of CAD86*, 1986, pp.69-77.

[Cohe79] Cohen, E., Lyche, T., Riesenfeld, R., "Discrete B-Spline and Subdivision Techniques in Computer-Aided Geometric Design and Computer Graphics", *Computer Graphics and Image Processing*, 14, 1987, pp.87-111.

[Debo72] De Boor, C., "On Calculating with B-Splines", *Journal of Approximation Theory*, 6, 1972, pp.50-62.

[Dokk85] Dokken, T., "Finding intersections of B-spline represented geometries using recursive subdivision techniques", *Computer Aided Geometric Design*, 2, 1985, pp.189-195.

[Fari87] Farin, G., Rein, G., Sapidis, N., Worsey, A. J., "Fairing cubic B-spline curves", *Computer Aided Geometric Design*, 4, 1987, pp.91-103.

[Glou90] Gloudemans, J. R., "Filleting of Aircraft Components Using Non-Uniform B-Spline Surfaces", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1990.

[Greg73] Gregory, T. J., "Computerized Preliminary Design at the early stages of Vehicle Definition", NASA TM X-62,303, 1973.

[Grie88] Grieshaber, M., "Interactive Calculation of Cross-Sectional Areas for Aircraft Design and Analysis", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1988.

[Grie89] Grieshaber, M., Jayaram, S., Jayaram, U., Myklebust, A., Mahan, J. R., "Interactive Aircraft Section Calculation for Drag", *UPCAEDM '89*, 1989, pp.113-120.

[Hill90] Hill, F. S. Jr., "Computer Graphics", *MacMillan Publishing Company*, 1990.

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

[Koch89] Kochan, S., "Programming in C", *Hayden Books*, 1989.

[Lin91] Lin, W., "An Intelligent, Object-Oriented Software Development Environment for Geometric Modeling in Design", Dissertation - PhD. in Mechanical Engineering, Virginia Polytechnic Institute and State University, to be submitted, 1991.

[Lane80] Lane, J., Riesenfeld, F., "A Theoretical Development for the Computer Generation and Display of Piecewise Polynomial Surfaces", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1, 1980, pp.35-46.

[Lass86] Lasser, D., "Intersection of parametric surfaces in the Bernstein-Bezier representation", *Computer Aided Design*, 18, No. 4, 1986, pp.186-192.

[Mala89] Malan, P., "Inlet Drag Prediction for Aircraft Conceptual Design", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1989.

[Mala90] Malan, P., Brown, E. F., "Prediction of Inlet Drag for Aircraft Conceptual Design", *17th ICAS Congress, Stockholm, Sweden*, 1990, pp.568-584.

[Mort85] Mortenson, M., "Geometric Modeling", *John Wiley and Sons*, 1985.

[Peng84] Peng, Q. S., ″An algorithm for finding the intersection lines between two B-spline surfaces″, *Computer Aided Design*, 16, No. 4, 1984, pp.191-196.

[Pieg89a] Piegl, L., ″Modifying the shape of rational B-Spline. Part 1: curves″, *Computer Aided Design*, 21, No. 8, 1989, pp.509-518.

[Pieg89b] Piegl, L., ″Modifying the shape of rational B-Spline. Part 2: surfaces″, *Computer Aided Design*, 21, No. 9, 1989, pp.538-546.

[Tayl88] Taylor, A., ″Specification of Mission Cycles for Aircraft Conceptual Design Using the PHIGS Standard″, Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1988.

[Wamp88a] Wampler, S., ″Development of a CAD System for Automated Conceptual Design of Supersonic Aircraft″, Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1988.

[Wamp88b] Wampler, S., Myklebust, A., Jayaram, and Gelhausen, P., ″Improving Aircraft Conceptual Design - A PHIGS Interactive Graphics Interface For ACSYNT″, *American Institute of Aeronautics and Astronautics*, AIAA-88-4481, 1988.

[Warr89] Warren, J., ″Blending Algebraic Surfaces″, *ACM Transactions on Graphics*, 8, No. 4, 1989, pp.263-278.

[Wong90] Wong, C., ″Intersection of B-Spline Surfaces By Elimination Method″, Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1990.

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

Appendix A. ACSYNT B-Spline Module Manual

A.1 Introduction to the ACSYNT B-Spline Module

This manual is a technical reference for future research and development in geometric modeling as well as a simple user's guide to orient new users with the ACSYNT B-Spline Module. The first part is a comprehensive tutorial of the B-Spline Module's capabilities. The second section of this manual gives a detailed explanation of the organization of the B-Spline Module along with descriptions of each function available.

Since this code was developed to serve as a platform for future research, some of the algorithms may only work under certain assumed conditions or "special cases". Such algorithms will be preceded with disclaimers to warn the user of any such assumptions.

A.1.1 User Guide

The user guide section of this manual consists of step by step tutorials on the use of the different algorithms available in the B-Spline Module. The tutorials are accompanied by various examples and brief descriptions of the purpose of each algorithm. Please pay special attention to every detail in this section, since some of the examples use code that may not be robust enough to handle user input errors.

A.1.2 Technical Reference

The technical reference section of this manual is divided into the following parts:

- Format
- Input
- File Handling
- Display
- B-Spline Utilities

Each section begins with a brief explanation of its purpose within the B-Spline Module followed by detailed descriptions of each function within that section. The purpose of these detailed descriptions is to make the user/programmer aware of the available utility routines.

A.1.3 Screen Layout

The user interface for the B-Spline Module consists of six different sections:

- Software Title
- Regular Menu Input
- Standard Menu Input
- Message Scroll Area
- String Input Echo Area
- Geometry Display

Figure 47 on page 128 shows the location of each section on the display screen. The regular menu display area allows room for a menu title and a maximum of thirteen menu options. The standard menu area is reserved for those functions which can be accessed at any time during execution of the software. A maximum of five messages can be displayed at a time in the message scroll area to provide proper feedback to the user. Utility routines have been created to aid the programmer in displaying the menus and messages.

The geometry display area is reserved for the four primary three dimensional views of aircraft models. Any of these four views can be displayed alone or all four views may be displayed at once. Control of views to be displayed is handled through a standard menu item. A two-dimensional window can also be displayed in the geometry area. The two-dimensional window gives the user an alternative list of possible options when providing input to the software.

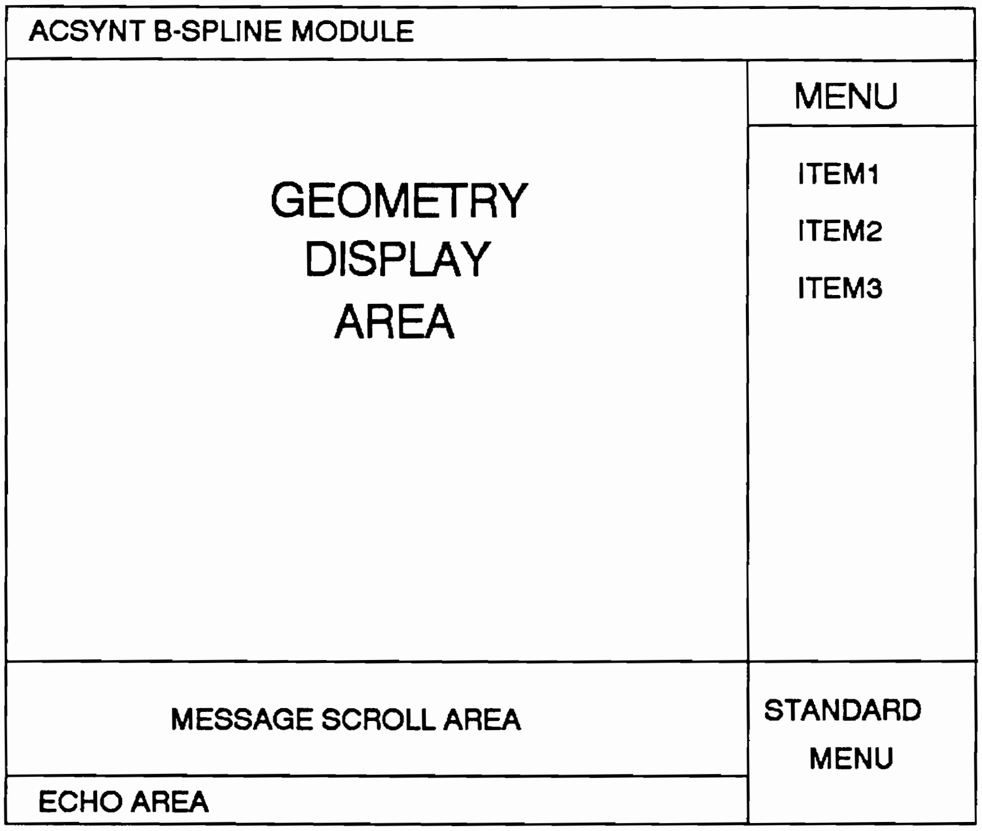


Figure 47. ACSYNT B-Spline Module Screen Layout

A.2 Tutorial - ACSYNT B-Spline Module

This tutorial has been organized to orient the new user with the features of the ACSYNT B-Spline Module. The tutorial gives step by step instructions on every input command necessary to accomplish a given task. Example input files are included, as part of the tutorial, to demonstrate the various capabilities.

The tutorial has been divided into several sections. Each section is designed to cover a new algorithm developed at Virginia Tech. It is not necessary to cover the tutorial in the order given for all sections. However, a sound knowledge of the first section is necessary in order to be successful with any of the later sections.

Each section of the tutorial is described using lessons. The lessons are essentially a list of steps to accomplish a task. The steps will contain the title of the regular menu on the left and the option to select on the right. Whenever necessary, italicized text will be used to give a brief insight into the function being performed. All string entry will be in normal text with the exact string to be entered within quotes.

The section headings are given below:

- Orientation (Basic Operation Features)
- Creating a point data file
- Non-Uniform Cubic B-Spline Inversion
- Intersection of B-Spline Surfaces
- Filleting of two B-Spline Surfaces
- Computation of Mass Properties for a closed B-Spline Surface

A.2.1 Orientation (Basic Operation Features)

This section of the tutorial has been organized to give the user a sound foundation of the overall operation and menu structure of the ACSYNT B-Spline Module. The first step is of course to make sure you have the code properly installed on the workstation and have successfully compiled the source code to build an executable file called 'acsbsm'. If these tasks have been completed type 'acsbsm' from the 'execs' directory. The B-Spline Module display should appear on the graphics display device.

Notice the overall layout of the screen here. The display is divided into six sections. All but one of the sections allow user input or feedback. This section is the title section located across the top of the screen. Menu input has been split into regular menu input and standard menu input. The regular menu is a dynamic menu displayed along the upper right side of the screen. The regular menu has a title defining the module currently active along with the options available within that module. The standard menu is located in the lower right corner of the screen. The standard menu gives a list of six static options. These options can be selected at any time during execution. The standard menu options can also be selected from the choice box or by typing in the name of the command from the keyboard.

The lower area of the screen has been reserved for user feedback. The small black strip at the bottom is the string echo area. Any input from the keyboard will be seen in this box until the enter key has been pushed. The larger box above the echo area is reserved for the display of messages to prompt the user for input or give feedback on the success of any operation. Please pay special attention to these messages as they are intended to guide the user throughout the execution of any module. Finally, in the center of the screen is the geometry display area. The four primary geometry views of any object can be seen in this area.

The orientation section of the tutorial has been divided into two lessons. The first lesson is intended to guide the novice user through the regular menu structure. By doing so, the user will learn the functions of the main, file I/O, and geometry menu modules. The second lesson will demonstrate the function of the standard menu options. A sound knowledge of these two lessons is essential before the user can use the other tutorials.

Lesson 1: Manipulating the Regular Menu Structure

Sequence of Commands:

Menu Title	*Select/Enter*
MAIN MENU	FILE
FILE I/O	READ HERMITE

- Type in "f16.hermite"

An f16 point data file will be read in and the f16 geometry will be displayed in the Hermite surface format. Pay attention to the messages giving feedback about the success or failure of the request. If the f16 is not read in successfully... try again and check the spelling.

FILE I/O	RETURN
MAIN MENU	GEOMETRY
GEOMETRY	COLOR
CHG COLOR	LIST

A list of each component will be displayed in the upper left corner of the screen. Components can be selected by name (from this list) or from any of the geometry views.

CHG COLOR	ONE COMPONENT
COLORS	LIGHT BLUE

- Select the nose of the f16

CHG COLOR	ONE COMPONENT
COLORS	GREEN

- Select the component name "WING" from the Component list.

CHG COLOR	RETURN
GEOMETRY	RE-TILE
RE-TILE	ALL COMPONENTS

- Press enter to accept default of 2 by 2 rendering

• RE-TILE	ONE COMPONENT

- Type in "10 10". Notice, no commas between numbers.
- Select any component on the aircraft

• RE-TILE	RETURN
• GEOMETRY	COMPONENT
• COMPONENT	LIST
• COMPONENT	DELETE

- Select a component from either the geometry or the component list window

• COMPONENT	RETURN

This concludes lesson 1. At this point you should feel comfortable with the functions under the main, file i/o, and geometry menu modules. Try reading in other Hermite input files from the execs directory and manipulating their display.

The next lesson illustrates the standard menu features. This lesson can be initiated from any menu module. The only assumption is that there is geometry displayed on the screen.

Lesson 2: The Standard Menu Items

• *Menu Title*	*Select/Enter*
• (ANY MENU MODULE)	WINDOW (STANDARD MENU)
• WINDOW	MULTI-VIEW
• WINDOW	SINGLE VIEW
• SINGLE VIEW	ISOMETRIC VIEW
• SINGLE VIEW	SOFT VAL (STANDARD MENU)

A software valuator box will be displayed in the message scroll area. Each slider represents a valuator. Simply make a pick inside any of the boxes and watch the resulting change in the geometry display. The geometry can also be manipulated using the hardware valuators if available.

Val#	***Action***
1	*Rotation about X axis*
2	*Rotation about Y axis*
3	*Rotation about Z axis*
4	*Translation in X*
5	*Translation in Y*
6	*Translation in Z*

7	*Scale*
8	*Sensitivity (Software Valuators only)*

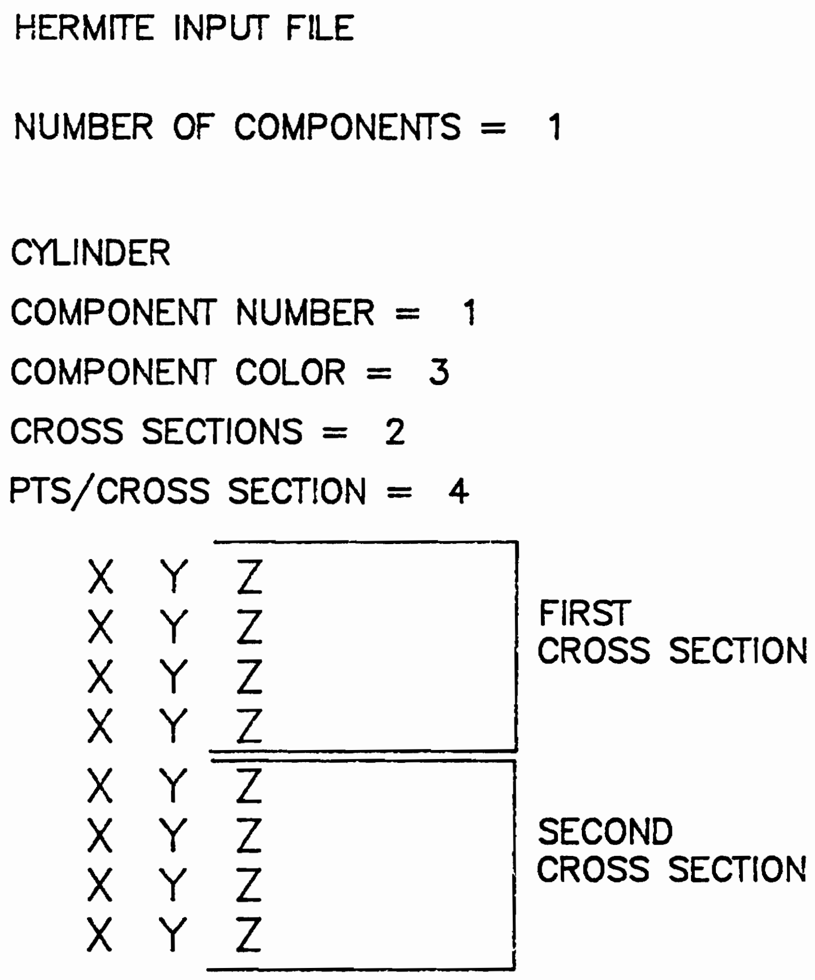
Repeat the above exercise using the standard menu options, only this time select the options from the either the choice box or by typing in the name of the command.

This completes the Orientation section of the tutorial. Make sure these features are well known before attempting any of the following sections. There are several input files included in the execs directory. Load these files and manipulate their display.

A.2.2 Creating a Point Data File

The example data files read in during the first two lessons of the tutorial where labeled "Hermite Input Files". These files contain the basic data necessary to display any geometric object. An example of the format for a Hermite data file is shown in Figure 48 on page 134 The file consists of two parts. The first part is the heading in which the type of file is given (in this case "HERMITE INPUT FILE") along with the number of separate components in the file.

The second part of the input file is the component data section. The component data section is a list of the various components in the model. Each component within the list is described by the following four attributes: name, number, color and point data. The component name can be any descriptive string of 20 characters or less. The component number is any arbitrary number that is unique in the data file. The component color is an index number between 0 and 12 (0 is black).



```
HERMITE INPUT FILE

NUMBER OF COMPONENTS =  1

CYLINDER
COMPONENT NUMBER =  1
COMPONENT COLOR =  3
CROSS SECTIONS =  2
PTS/CROSS SECTION =  4
      X  Y  Z
      X  Y  Z        FIRST
      X  Y  Z        CROSS SECTION
      X  Y  Z
      X  Y  Z
      X  Y  Z        SECOND
      X  Y  Z        CROSS SECTION
      X  Y  Z
```

Figure 48. Hermite Input File Format

After the color index, the number of cross sections and points per cross section are defined. Finally, the actual points describing the component are given. The points are organized per cross section. For example a component with 4 cross sections and 8 points per cross section would have the 8 points on the first cross section followed by the 8 points on the second cross section, etc. resulting in a total of 32 data points.

Within the execs directory there is file labeled "tutor.hermite". Take a look at this data file. The component described is a cylinder with a diameter of 3 and height of 13. Notice, that each cross section consists of 9 points describing a circle in the XY plane. Each new cross section has a different Z value but similar X and Y values. Now try reading the "tutor.hermite" file into the B-Spline Module. The screen should display a cylinder. Refer to lesson 1 in the orientation section for help on reading in the "tutor.hermite" input file.

A.2.3 Non-Uniform Cubic B-Spline Inversion

Non-Uniform Cubic B-Spline Inversion is the process of converting the point data read in from the Hermite input file to a B-Spline surface representation. From a user standpoint this is a simple process. The next lesson will demonstrate the inversion process and allow the user to compare the Hermite and B-Spline representations with the display option of the geometry menu module.

Lesson 3: The B-Spline Inversion Process

Sequence of Commands:

• *Menu Title*	*Select/Enter*
• MAIN MENU	FILE
• FILE I/O	READ HERMITE

- Type in "f16.hermite"

• FILE I/O	RETURN
• MAIN MENU	GEOMETRY
• GEOMETRY	B-SPLINE

The Hermite geometry will now be inverted to the Non-Uniform Cubic B-Spline representation.

• B-SPLINE	RETURN

That was the entire inversion process from the users point of view. Whenever the B-Spline option is selected, the B-Spline representation is automatically displayed. If no B-Spline geometry exists, the inversion process is invoked.

• GEOMETRY	DISPLAY

All four primary views will be displayed

• DISPLAY	HERMITE

- Select any of the views to display the Hermite geometry in.

The display option can be very advantageous when comparing different representations. Try to determine any differences between the Hermite and B-Spline representations.

• DISPLAY	RETURN
• GEOMETRY	RETURN

The goal of the cubic B-Spline inversion was to develop a curvature continuous surface out of cross section data. However, many configurations require curvature continuity at one cross section and only point continuity (C^0) at others. This discontinuity can be accomplished by placing two cross sections at the same location.

In order to demonstrate this procedure, edit the "tutor.hermite" input file created in the above tutorial. Copy the first and last cross sections over so that there are a total of 6 cross sections in all, but only four separate sets of points. Now add starting and ending cross sections that consist of all zeros in the X and Y locations. Finally, increase the number of cross sections to 8. Now try to read this file into the B-Spline Module and invert it to the B-Spline representation. Do not be alarmed by the differences between the Hermite and B-Spline representations. The B-Spline representation should look like a closed cylinder. The C^2 discontinuity has been forced at both ends of the cylinder. If the display does not look correct, there is an example file called "cylinder.hermite" in the execs directory. Compare the "tutor.hermite" file with this file and correct any mistakes.

A.2.4 Intersection of B-Spline Surfaces

Lesson 4 will describe the options available under the "Intersect" menu. The intersect menu module will compute the intersection between any two components and load the data into a data structure for later use in other modules. The intersection data is necessary for filleting (blending) of two components and/or the calculation of mass properties.

Lesson 4: Intersection of two B-Spline surfaces

Sequence of Commands:

• *Menu Title*	*Select/Enter*
• MAIN MENU	FILE
• FILE I/O	READ B-SPLINE
• FILE FORMAT	DAP DEMO

- Type in "f16cap.nurbs2"

  This is an f16 data file with a C^2 discontinuity forced at the wing ends resulting in end caps without distorting the shape of the air foil.

• FILE I/O	RETURN
• MAIN MENU	GEOMETRY
• GEOMETRY	B-SPLINE
• B-SPLINE	INTERSECT
• INTERSECT	LIST
• INTERSECT	SELECT COMPS

- Select any two intersecting components from the geometry or the component list window.

  Messages will give feedback as to the status of the intersection computations.

  Upon completion a final message will display the maximum distance between points on both surfaces along the intersection curve. The intersection curve will also be *displayed with a white line. If more than one curve of intersection is found the different lines will be shown in different colors.*

INTERSECT	RETURN
B-SPLINE	RETURN
GEOMETRY	RETURN
MAIN MENU	EXIT
-EXIT-	YES

A.2.5 Filleting of Two B-Spline Surfaces

Filleting of two B-Spline surfaces was developed to blend any two intersecting surfaces to form one C^2 continuous surface. The filleting algorithm described below is valid for filleting along iso-parametric lines. The first step in filleting two intersecting surfaces is to calculate the intersection curve. For this reason, Lesson 5 will demonstrate the obtaining of intersection data and fillet between two surfaces.

Lesson 5: Filleting of two B-Spline surfaces

Sequence of Commands:

Menu Title	*Select/Enter*
MAIN MENU	FILE
FILE I/O	READ B-SPLINE
FILE FORMAT	SHOWTIME

- type in "wallfloor.showtime".

This file contains two simple components that appear to be a wall intersecting a floor.

FILE I/O	RETURN
MAIN MENU	GEOMETRY
GEOMETRY	B-SPLINE
B-SPLINE	INTERSECT
INTERSECT	ALL COMPS
INTERSECT	RETURN
B-SPLINE	FILLET
FILLET	SELECT COMPS

- Select the wall first and then the floor.

Interactive messages will walk you through the filleting procedures.

The parametric direction refers to which direction the fillet should blend both surfaces (positive or negative). The '1' or '-1' refers to which way the surface was originally put together. For example, the fillet needs to walk up the wall, but the wall was developed from the top down. Therefore, the fillet needs to blend in the negative direction on the wall (-1).

- Accept the default values of -1 1 for the parametric directions

The offsets for both surfaces refers to the distance from the intersection line for the blend to begin.

- Enter 1.50 for the first surface and 2.20 for the second

The 'rho' refers to the rate of change of curvature for the blend. A 'rho' of 0.0 results in a straight line between the offset points while a 'rho' of 1.0 gives the sharpest possible corner between the surfaces.

- Enter a rho value of 0.5

The new fillet surface will be seen in yellow

• FILLET	RETURN
• B-SPLINE	RETURN
• GEOMETRY	COMPONENT
• COMPONENT	LIST

The component list now includes the new fillet surface. This surface can be manipulated just as the other surfaces were in Lessons 1 and 2.

• COMPONENT	RETURN
• GEOMETRY	RETURN
• MAIN MENU	EXIT
• -EXIT-	YES

That concludes all the necessary steps for filleting two surfaces along iso-parametric lines. Try the filleting procedure on the following data files. Notice, a ".hermite" extension refers to a Hermite data file while a ".showtime" extension refers to a B-Spline data file in the showtime format.

- ex30.showtime
  - Parametric directions fuselage 1, wing 1
  - Offsets fuselage 1.00, wing 1.50

    - rho 0.35
- con_cyl.showtime
    - Parametric directions cone 1, cylinder 1
    - Offsets cone 3.00, cylinder 2.40
    - rho 0.30
- plnsqr.hermite
    - Parametric directions floor 1, pipe 1
    - Offsets floor 0.8 pipe 1.0
    - rho 0.25

A.2.6 Computation of Mass Properties for a Closed B-Spline Surface

Lesson 6 will describe the automatic computation of mass properties for arbitrary B-Spline surfaces. The algorithm developed assumes a closed surface. Therefore, some error may exist if the surface has holes or is not completely closed. To demonstrate this feature read the "tutor.hermite" input file created earlier. If this file does not exist or is not correct, read the "cylinder.hermite" input file.

Lesson 6: Computation of Mass Properties

Sequence of Commands:

• *Menu Title*	*Select/Enter*
• MAIN MENU	FILE
• FILE I/O	READ HERMITE

- Type in "tutor.hermite" if the file exists. Otherwise, type in "cylinder.hermite".

• FILE I/O	RETURN
• MAIN MENU	GEOMETRY
• GEOMETRY	B-SPLINE
• B-SPLINE	MASS PROP
• MASS PROP	ALL COMPS
• MASS PROP	EXECUTE

The mass properties will be computed for the cylinder and displayed in the message scroll area. The location of the center of gravity is also displayed in the geometry window.

This is an easy case to check; the cylinder has a diameter of 3 and a height of 13. Check the values given against your own calculated values for the cylinder.

In this case there was only one component. If a model has multiple components, the mass properties for the entire model can be calculated or the contribution of any one or group of components can be computed.

• MASS PROP	RETURN
• B-SPLINE	RETURN
• GEOMETRY	RETURN
• MAIN MENU	EXIT
• -EXIT-	YES

Read the other geometry files and compute their mass properties. A simple sanity check is the marker locating the center of gravity for the model. If this does not look correct, calculate the center of gravity and compare the results.

A.3 Software Development Using the B-Spline Module

A.3.1 Development Guidelines

The organization of the B-Spline Module follows explicit rules for the format of data storage, documentation and menu display. First and foremost, header blocks have been designed to precede any file containing source code and any function in that file. These header blocks are located in the 'execs' directory of the source code directory tree and are labeled 'header.file' and 'header.module' respectively. These header blocks can be obtained using the 'recover' command within any UNIX 'vi' editor. Sample header blocks can be seen in Figure 49 on page 142.

The main data structure is contained in the include file 'showtime.h', which is also located in the 'execs' directory. This data structure contains all the data necessary for

"header.file" Header Block

```
/**********************************************************************
*   Name:
*   Author:
*   Date:
*
*   Description:
*
**********************************************************************/

#include <afmnc.h>
#include "../execs/showtime.h"

/*--------------------FUNCTION DECLARATIONS-------------------------*/

/*--------------------END OF FUNCTION DECLARATIONS-------------------*/

/*--------------------CONSTANT DEFINITIONS--------------------------*/

/*--------------------END OF CONSTANT DEFINITIONS--------------------*/
```

"header.module" Header Block

```
/**********************************************************************
*   Module Name:
**********************************************************************
*   Description:
*
*   Input:
*
*   Output:
**********************************************************************/
```

Figure 49. Header Block Formats

the display of geometry using the bi-cubic Hermite and bi-cubic B-Spline representations as well as attributes such as color, name and number of each component. When any file input is performed, this data structure is loaded into a linked list format. A listing of this data structure appears in Figure 50 on page 144.

The main 'model' data structure is used to define the initial entry point of the linked list along with the root structure identification numbers (id's) for the various types of geometry representations. When a new representation is implemented, the main data structure should be updated with a new root structure id assigned to the particular representation. The root structure id is very important, as it is the only method for knowing the data which has been loaded into the data structure. The main goal of the Model data structure is to limit the data passed between modules to that of the pointer to the Model data structure. A listing of the model data structure is shown in Figure 51 on page 145.

Also contained in the 'showtime.h' include file are definition statements for constant variable names used throughout the source code. These definitions can be seen in Figure 52 on page 146. All definitions in the 'showtime.h' include file are in capital letters to designate them as global declarations. Any definition made within a single file is designated with the first letter capitalized while local variables are in all lower case letters.

It is recommended that the designer develop a data structure for the development of new code to store any data that is needed in any new algorithm. This should be done in order to limit the complexity of data passed between individual routines. Additionally, data structures should be dynamically allocated upon entry into the module and later freed when leaving the module. All variable definitions should be accompanied by a brief

Component Data Structure

```
typedef struct compdata_type {
        int comp_number;                    /* component number */
        char comp_name[20];                 /* component name */
        int acs_id;                         /* structure id    */
        int nubs_id;
        int *hull_id;
        int fillet_id;
        int open[2];                        /* open flag 1 closed 0 open */
        int color;                          /* component color */
        int existence;                      /* 1 exists 0 does not exist */
        int nu;                             /* rendering in u */
        int nw;                             /* rendering in w */
        int acs_ncross;                     /* number of cross sections */
        int acs_npts;                       /* number of pts per xsection */
        float ***acs_pts;                   /* pointer to component pts */
        float ***acs_utan;                  /* pointer to tangents in u dir */
        float ***acs_wtan;                  /* pointer to tangents in w dir */
        int nu_knots;                       /* number of u knots */
        int nw_knots;                       /* number of w knots */
        float *u_knot;                      /* u knot array */
        float *w_knot;                      /* w knot array */
        float ***hull;                      /* control hull */
        struct compdata_type *next;         /* pointer to next component */
                                            }comp_data;
```

Figure 50. Main Data Structure

Model Data Structure

```
typedef struct {
        int num_comp;                          /* number of components in model */
        int acs_root;                          /* root structure id */
        int nubs_root;                         /* Non-Uniform B-Spline root id */
        int int_root;                          /* Structure id for intersection data */
        int fillet_root;
        comp_data *comp;                       /* pointer to beginning of linked list */
        struct intersection_type *intlist;     /* list of intersections */
                                               }MODEL;
```

Figure 51. Model Data Structure

```
/* ----------- Declare View Variables --------------------------- */
#define ON 1
#define OFF 0
#define MENU_VIEW 1
#define ECHO_VIEW 2
#define STRING_VIEW 3
#define STDMNU_VIEW 4
#define TITLE_VIEW 5
#define GEOM1_VIEW 6
#define GEOM2_VIEW 7
#define GEOM3_VIEW 8
#define GEOM4_VIEW 9
#define WINDOW_VIEW 10
#define SOFT_VAL_VIEW 11
/* ------------ End View Declarations ---------------------------- */

/* ----------- Define Colors ----------------------------------- */
#define BLACK 0
#define RED 1
#define GREEN 2
#define BLUE 3
#define YELLOW 4
#define PURPLE 5
#define LT_BLUE 6
#define DK_GREY 7
#define WHITE 8
#define DK_BLUE 11
#define LT_GREY 12
#define GREY 13
/* ----------- End Color Definition Block ------------------------ */

/* ----------- Define Input Variables ---------------------------- */
#define NONE 0
#define LOCATOR 1
#define STROKE 2
#define VALUATOR 3
#define CHOICE 4
#define PICK 5
#define STRING 6

#define REG_MENU 1
#define STD_MENU 2
#define GEOMETRY 3
#define WINDOW 4

#define VAL1 1
#define VAL2 2
#define VAL3 3
#define VAL4 4
#define VAL5 5
#define VAL6 6
#define VAL7 7
#define VAL8 8
/* ----------- End Input Declarations ---------------------------- */
```

Figure 52. Global Constant Definitions

description of the purpose of the variable. Also note that variable names are not limited to seven characters and therefore should be as descriptive as possible.

While developing new algorithms it is important that individual routines be written with a general purpose so that future developers can utilize these functions. Additionally, all new algorithms should be completed with full documentation of each individual function.

A.3.2 Building Menu Modules

Any entry into an algorithm is initiated with a new menu module. However, the use of menu modules is not limited to new algorithms. As stated before, only the pointer to the Model structure should be passed between separate modules. Several utility routines have been written to free the developer from the menial tasks of menu display, user feedback and input retrieval. Descriptions of these routines can be found in the Input section of this manual. Figure 53 on page 148 is an example of the format for a new menu module. This format can be retrieved from the 'execs' directory using the vi 'recover' command.

Special attention must be devoted to the format of menu items. The new module name should be given followed by a list of the user options. The menu options are organized with the most used option at the top and the least used option at the bottom. Following this format, the RETURN option is always listed first when creating a new menu module. The title and all menu options should be displayed in capital letters. The same principal is followed when sending messages to the message scroll area.

```
int Return = 0;                                  /* return code */
int no_items = 3;                                /* Menu Parameters */
STATIC CHAR *TITLE = "MENU TITLE";
STATIC CHAR *ITEMS[] = { "RETURN",
                         "item 1",
                         "item 2" };

newmenu(title,no_items,items);    /* display new menu */

while ( Return != 1 )          /* initial loop to start execution */
  {
   Return = proc_input();

   if ( Return > 0 )
     {
      switch (Return)
        {
         case (1):
                 Return = 1;
                 break;

         case (2):
                 Action for item 1
                 break;

         case (3):
                 Action for item 2
                 break;

         default:
                 message("BAD INPUT IN 'title'",1);
                 break;
        }

     }

  }

  oldmenu();                    /* display old menu */
```

Figure 53. Menu Module Format

A.3.3 Summary

To reinforce the above rules for format, the following list of development guidelines has been composed. These rules were defined during the initial design of the B-Spline Module and have proved to be very useful throughout the development.

- Begin all files with the 'header.file' title block.
- Begin all modules with the 'header.module' title block.
- All geometry representations must have a distinct root id in the Model data structure.
- Global definitions must be designated in CAPITAL LETTERS.
- Local definitions must be designated with the first letter Capitalized.
- Local variables must be designated by all lower case letters.
- Separate data structures should be implemented for each new algorithm.
- Only the pointer to the model data structure should be passed between modules.
- Variable names should be as descriptive as possible.
- All menu options begin with the RETURN option.
- All menu options are in CAPITAL letters.

- All messages are in CAPITAL letters.

A.4 Input Functions in the ACSYNT B-Spline Module

The B-Spline Module can accept input from the following five logical input devices: Pick, Locator, Valuator, Choice, and String. The B-Spline Module operates in event mode input except when request or sample mode is invoked for String, Locator, or Pick input. In addition, for those workstations not equipped with hardware valuators, a software valuator algorithm has been designed and developed. The software valuators display sliding bars in the message area of the screen.

All input is processed through one main input routine which determines the type of input and processes the input based on the type. If possible, the input is processed internally (i.e. moving the geometry based on valuator input) and no action is required from a developer's standpoint. Any input that can not be processed internally is returned to the calling routine for processing. Several input utility routines have been written to retrieve pick input from two-dimensional windows or the geometry display area. All input routines are in this section.

If possible, input from the standard menu, choice box, or the keyboard are processed internally. The standard menu layout is mimicked on the choice box to give the user the option of selecting from the standard menu or the choice box. Finally, any event mode string input is checked against a list of key words to determine if the user is requesting a standard menu option via the keyboard. These three separate devices have been utilized to allow for those workstations that do not have all five logical input devices.

All pickable items are grouped by type. Refer to Figure 52 on page 146 for the various input type definitions. Pick input is processed based on the group pick id. Within each group, every item must contain its own pick id. Essentially, all the pickable items are loaded into a sub-structure which is executed by a root or main structure containing the type id of the group. This corresponds to a pick depth of two with the top level defining the type of pick and the second level defining the item picked within the group.

A.4.1 proc_input - Get and process event mode input

Purpose:

This routine will wait for an input event to enter the input queue. Upon receiving input, the type is determined and the proper input processing routines are activated.

Description:

int proc_input()

Input Arguments:

NONE

Output Arguments:

NONE

Function Output:

Upon receiving and acting on input, this routine will return an integer value corresponding to the resulting action.

Value	Meaning
<0	Item was picked from 2D window
$=0$	Input has been processed internally
>0	Item was picked from regular menu (1 always means return)

A.4.2 get_input - Await event mode input

Purpose:

Make PHIGS calls to await event mode input from the input queue.

Description:

void get_input(class, device)

int	*class
int	*device

Input Arguments:

NONE

Output Arguments:

class	class of logical input received.
device	device number of that class (ie. which valuator)

Function Output:

NONE

A.4.3 get_pick - Get event mode pick input

Purpose:

Gets event mode pick input - Logical Unit Number 5.

Description:

void get_pick(type, item)

int	*type
int	*item

Input Arguments:

NONE

Output Arguments:

type	type of pick made (i.e. regular menu, geometry, etc.)
item	item picked from that type or group (i.e. which menu option)

Function Output:

NONE

A.4.4 get_string - Get string input from console

Purpose:

Gets string input from the keyboard - Logical Unit Number 6.

Description:

void get_string(string)

char	string()

Input Arguments:

string	default values for expected input.

Output Arguments:

string	string returned from keyboard.

Function Output:

NONE

A.4.5 get_valuator - Get valuator input

Purpose:

Gets valuator input from a specific valuator device number - Logical Unit Number 3.

Description:

void get_valuator(value)

float	*value

Input Arguments:

NONE

Output Arguments:

value	value retrieved from valuator (0< =value< = 1)

Function Output:

NONE

A.4.6 get_choice - Get choice or button box input

Purpose:

Gets choice or button box input - Logical Unit Number 4.

Description:

int get_choice()

NONE

Input Arguments:

NONE

Output Arguments:

NONE

Function Output:

Number corresponding to which button was pushed.

A.4.7 proc_std_menu - Process standard menu input

Purpose:

Processes standard menu input option.

Description:

int proc_std_menu(item)

int	item

Input Arguments:

item	Which item was selected from standard menu options
Number	Action
1	SOFTWARE VALUATORS
2	SHADING - (Not Available)
3	COPY - Enlarge view on screen
4	WINDOW - Change view display
5	Empty
6	EXIT - Return out of menu module

Output Arguments:

NONE

Function Output:

Exit flag - 1 means exit was selected from standard menu.

A.4.8 samp_locator - Sample locator device

Purpose:

Samples for which view the locator device is located - Logical Unit Number 1.

Description:

```
int samp_locator(device)

        int             device
```

Input Arguments:

device	device number of locator (1-4).

Output Arguments:

NONE

Function Output:

Number corresponding to which view the locator device is in.

A.4.9 req_locator - Request locator input

Purpose:

Request locator input - Logical Unit Number 1

Description:

```
int req_locator(device)

        int             device
```

Input Arguments:

device	device number of locator (1-4).

Output Arguments:

None

Function Output:

Number corresponding to view locator was in at time of request.

A.4.10 proc_choice - Process choice input

Purpose:

Processes any choice event mode input. Buttons on the choice box can be defined for options accessible at any time during execution. In this manner the user has quick access to functions without leaving the particular menu module he/she is working in.

Description:

int proc_choice()

NONE

Input Arguments:

NONE

Output Arguments:

None

Function Output:

Return flag - 1 means user selected the return option from the choice box.

A.4.11 proc_string - Process string event mode input

Purpose:

Compares event mode string input against several pre-defined commands to determine if the user is requesting a standard menu or button box choice from the keyboard.

Description:

int proc_string()

NONE

Input Arguments:

NONE

Output Arguments:

NONE

Function Output:

Return flag - 1 means user wants to return out of menu module.

A.4.12 get_comp_pick - Get component pick

Purpose:

Gets the component selected from the geometry view area by the user. Will also process any standard input.

Description:

```
comp_data *get_comp_pick(Model)

        MODEL          *Model
```

Input Arguments:

Model	Pointer to the Model data structure.

Output Arguments:

NONE

Function Output:

Pointer to the data structure for a particular component within the main data structure linked list.

A.4.13 newmenu - Draw new menu

Purpose:

Displays a new menu module in the menu area of the screen.

Description:

void newmenu(title, no_items, items)

char	*title
int	no_items
char	*items()

Input Arguments:

title	Title of menu module.
no_items	Number of items in list.
items	List of menu options.

Output Arguments:

NONE

Function Output:

NONE

A.4.14 oldmenu - Draw previous menu

Purpose:

Displays the previous menu module and remove present menu from menu list.

Description:

void oldmenu()

NONE

Input Arguments:

NONE

Output Arguments:

NONE

Function Output:

NONE.

A.5 File Handling in ACSYNT B-Spline Module

File I/O has been divided into two main groups: Hermite I/O and B-Spline I/O. Hermite files follow the format shown in -- Figure id 'hmfil' unknown --. The Hermite data is simply point data. Once the point data has been read the tangents between points and cross sections are calculated and the resulting Hermite surface is displayed.

B-Spline file I/O has the following formats available:

DAP DEMO	Format for input to IBM 6090 distributed process demo.
SHOWTIME	Similar format to that of DAP DEMO but without trimming curves or relimiting data.
IGES	The Initial Graphics Exchange Specification Version 4.0 for Rational B-Spline Surfaces (entity number 128).

Sample formats for these files can be found in the 'execs' directory of the source code.

Whenever a file input operation is requested, the main data structure is initialized and loaded for later display and manipulation. If necessary, the old data allocations are removed before any new data is read.

A.6 Display functions in the ACSYNT B-Spline Module

Several utility routines are available for the display of geometry using the cubic Hermite and the cubic B-Spline representations. These routines are listed on the following pages.

A.6.1 display_geometry - Display specific geometry on screen

Purpose:

Controls what geometry is displayed in the geometry window. More than one type of geometry can be displayed in the same window.

Description:

void display_geometry(no_views, assoc, views, geometry)

int	no_views
int	assoc()
int	views()
int	geometry()

Input Arguments:

no_views	Number of views to display geometry in
assoc	Number of geometries to be displayed in each view
views	array of view indexes for the display of geometry
geometry	array of geometry root id's to display in a view

Output Arguments:

NONE

Function Output:

NONE

A.6.2 display_intsect - Display intersection data

Purpose:

Displays the intersection data of Non-Uniform B-Spline surfaces

Description:

void display_intsect(Model)

MODEL	*Model

Input Arguments:

Model	Pointer to the Model data structure

Output Arguments:

NONE

Function Output:

NONE

A.6.3 display_nubs - Display B-Spline surface

Purpose:

Displays only the Non-Uniform B-Spline surface representation on the screen.

Description:

void display_nubs(Model)

MODEL	*Model

Input Arguments:

Model	Pointer to the model data structure

Output Arguments:

NONE

Function Output:

NONE

A.6.4 display_hull - Draw control hull on screen

Purpose:

Toggles the display of the control hull for a cubic B-Spline surface

Description:

```
void display_hull(Model)

        MODEL           *Model
```

Input Arguments:

Model	Pointer to the model data structure

Output Arguments:

NONE

Function Output:

NONE

A.6.5 draw_hull - Draw components control hull

Purpose:

Loads the point data array for the display of a components B-Spline control hull.

Description:

```
void draw_hull(comp)

        comp_data       *comp
```

Input Arguments:

comp	Pointer to the component data structure

Output Arguments:

NONE

Function Output:

NONE

A.6.6 std_window - Display standard 2D window

Purpose:

Displays a 2D window in upper left corner of geometry window

Description:

void std_window(title, no_items, items)

char	*title
char	*items()
int	no_items

Input Arguments:

title	Title for 2D window.
items	List of strings to be displayed.
no_items	Number of items in list.

Output Arguments:

NONE

Function Output:

NONE

NOTE: The window view id is declared in the showtime.h include file as "WINDOW_VIEW".

A.6.7 open_2D_window - Open a 2D view

Purpose:

Initializes a 2D view for the display of model components

Description:

void open_2D_window(npc_viewpt, view_id)

float	npc_viewpt()
int	view_id

Input Arguments:

npc_viewpt	Array containing NPC coordinates of view.
view_id	ID number for view.

Output Arguments:

NONE

Function Output:

NONE

A.6.8 remove_2D_window - Remove a 2D view

Purpose:

Removes a 2D view from the workstation state list

Description:

```
void remove_2D_window(update, view_id)

        int          update

        int          view_id
```

Input Arguments:

update	Flag; 1 - Update workstation 0 - Do not update.
view_id	View identification number.

Output Arguments:

NONE

Function Output:

NONE

A.6.9 largest_string - Determine largest string in list

Purpose:

Determines the largest number of characters in any one string from a list of character strings.

Description:

int largest_string(no_items, items)

int	no_items
char	*items()

Input Arguments:

no_items	Number of character strings.
items	Array of character strings.

Output Arguments:

NONE

Function Output:

Largest number of characters in any one string.

A.6.10 view_active - Determine if view is active

Purpose:

Determines if a specified view is presently active.

Description:

int view_active(view_id)

int	view_id

Input Arguments:

view_id	View ID number.

Output Arguments:

NONE

Function Output:

Active Flag 1 - active 0 - not active

A.6.11 display_hermite - Display Hermite geometry

Purpose:

Displays only the Hermite geometry in all four geometry views.

Description:

void display_hermite(Model)

MODEL	*Model

Input Arguments:

Model	Pointer to the model data structure.

Output Arguments:

NONE

Function Output:

NONE

A.7 Utility Functions in the ACSYNT B-Spline Module

A number of utility routines are available for the manipulation of non-uniform bi-cubic B-Spline surfaces. These routines range from finding a point on a B-Spline surface to re-sizing the memory allocations for knot insertion. Most routines have been error trapped to prevent a core dump in the event of erroneous input. However, if a core dump should occur check the calling sequence to ensure the proper variables are being passed to the function.

A.7.1 blend - Compute blending functions

Purpose:

Computes the blending functions for a non-uniform B-Spline surface

Description:

void blend(rng, knot, u, N)

int	rng
float	knot()
float	u
float	N(4)

Input Arguments:

rng	Interval in knot sequence for blending functions.
knot	Array of knot values.
u	Knot value to compute blending functions at.

Output Arguments:

N	Blending functions.

Function Output:

NONE

A.7.2 bspline_point - Calculate Point on B-Spline surface

Purpose:

Calculates the point on a non-uniform B-Spline surface for specified u and w parametric values.

Description:

void bspline_point(comp, u_knot, w_knot, point)

comp_data	*comp
float	u_knot
float	w_knot
float	point(3)

Input Arguments:

comp	Pointer to component data structure.
u_knot	Parametric value in u direction.
w_knot	Parametric value in w direction.

Output Arguments:

point	Geometric location of point (x,y,z).

Function Output:

NONE

A.7.3 draw_bspline - Draw B-Spline Model

Purpose:

Draws the entire model in the non-uniform B-Spline format.

Description:

```
void draw_bspline(Model)

          MODEL          *Model
```

Input Arguments:

Model	Pointer to the model data structure.

Output Arguments:

NONE

Function Output:

NONE

A.7.4 draw_nubs - Draw a B-Spline surface

Purpose:

Draws one surface in the non-uniform B-Spline format.

Description:

```
void draw_nubs(comp)

          comp_data          *comp
```

Input Arguments:

comp	Pointer to the component's data structure.

Output Arguments:

NONE

Function Output:

NONE

NOTE: A PHIGS structure must be open before calling this routine

A.7.5 draw_u_nub - Draw along constant u line

Purpose:

Draws a line on the B-Spline surface along a constant parametric u line.

Description:

void draw_u_nub(comp, u_rng, u)

comp_data	*comp
int	u_rng
float	u

Input Arguments:

comp	Pointer to the component's data structure.
u_rng	Interval in the knot sequence.
u	Parametric u value

Output Arguments:

NONE

Function Output:

NONE

NOTE: A PHIGS structure must be open before calling this routine.

A.7.6 draw_w_nub - Draw along constant w line

Purpose:

Draws a line on the B-Spline surface along a constant parametric w line.

Description:

void draw_w_nub(comp, w_rng, w)

comp_data	*comp
int	w_rng
float	w

Input Arguments:

comp	Pointer to the component's data structure.
w_rng	Interval in the knot sequence.
w	Parametric w value

Output Arguments:

NONE

Function Output:

NONE

NOTE: A PHIGS structure must be open before calling this routine.

A.7.7 knot_range - Determine knot range

Purpose:

Determines the interval of a specific value within an array.

Description:

int knot_range(knot, knot_array, num_knots)

float	knot
float	knot_array
int	num_knots

Input Arguments:

knot	Specific knot value of interest.
knot_array	Array of knot values.
num_knots	Number of knot values in array.

Output Arguments:

NONE

Function Output:

Interval in array.

A.7.8 invert_nubs - Invert point data to NUBS format

Purpose:

Iteratively inverts point data to the non-uniform B-Spline format

Description:

void invert_nubs(Model)

MODEL	*Model

Input Arguments:

Model	Pointer to the model data structure.

Output Arguments:

NONE

Function Output:

NONE

NOTE: This routine will also display the model in the NUBS format when the inversion process is completed.

A.7.9 get_nubs_hull - Determine control hull

Purpose:

Iteratively computes the control hull vertices for the conversion of point data to non-uniform B-Spline surfaces.

Description:

void get_nubs_hull(comp, open)

comp_data	*comp
int	open()

Input Arguments:

comp	Pointer to the component's data structure.
open	Array of open/closed flags 1 - closed 0 - open.

Output Arguments:

NONE

Function Output:

NONE

A.7.10 invert_u - Invert in parametric u direction

Purpose:

Inverts point data along the parametric u direction for surface.

Description:

void invert_u(comp, open, hull)

comp_data	*comp
int	open
float	***hull

Input Arguments:

comp	Pointer to the component's data structure.
open	Open/Closed flag 1 - closed 0 - open.

Output Arguments:

hull	Inverted control hull array.

Function Output:

NONE

A.7.11 invert_w - Invert in parametric w direction

Purpose:

Inverts point data along the parametric w direction for surface.

Description:

void invert_w(comp, point, open)

comp_data	*comp
float	***point
int	open

Input Arguments:

comp	Pointer to the component's data structure.
point	Array containing point data
open	Open/Closed flag 1 - closed 0 - open.

Output Arguments:

NONE

Function Output:

NONE

A.7.12 curve_hull - Compute control hull for a curve

Purpose:

Iteratively computes the control hull for a non-uniform B-Spline curve.

Description:

void curve_hull(npoints, point, knot, open, invert, hull)

int	npoints
float	**point
float	*knot
int	open
int	*invert
float	**hull

Input Arguments:

npoints	Number of points in curve array.
point	Array of points to invert.
knot	Knot array
open	Open/Closed flag 1 - closed 0 - open
invert	Invert flag array 0 - invert 1 do no invert

Output Arguments:

hull	Resulting control hull array.

Function Output:

NONE

A.7.13 sur_knots - Calculate knot values

Purpose:

Calculates the knot values for inverting to NUBS format. This routine calculates knot values based on chord length parameterization.

Description:

void sur_knots(comp, open)

comp_data	*comp
int	open()

Input Arguments:

comp	Pointer to the component's data structure.
open	Open/Closed flag 1 - closed 0 - open.

Output Arguments:

NONE

Function Output:

NONE

A.7.14 get_u_knots - Determine knot sequence in u direction

Purpose:

Calculates the u knot vector

Description:

void get_u_knots(comp, open)

comp_data	*comp
int	open

Input Arguments:

comp	Pointer to the component's data structure.
open	Open/Closed flag 1 - closed 0 - open.

Output Arguments:

NONE

Function Output:

NONE.

A.7.15 get_w_knots - Determine knot sequence in w direction

Purpose:

Calculates the w knot vector

Description:

```
void get_w_knots(comp, open)

        comp_data       *comp

        int             open
```

Input Arguments:

comp	Pointer to the component's data structure.
open	Open/Closed flag 1 - closed 0 - open.

Output Arguments:

NONE

Function Output:

NONE.

A.7.16 average_u_dist - Average distance between two u lines

Purpose:

Calculates the average distance between two cross sections

Description:

float average_u_dist(comp, u1, u2)

comp_data	*comp
int	u1
int	u2

Input Arguments:

comp	Pointer to the component's data structure.
u1	Interval of first u line.
u2	Interval of second u line.

Output Arguments:

NONE

Function Output:

Average distance between lines.

A.7.17 average_w_dist - Average distance between two w lines

Purpose:

Calculates the average distance between two constant w lines

Description:

float average_w_dist(comp, w1, w2)

comp_data	*comp
int	w1
int	w2

Input Arguments:

comp	Pointer to the component's data structure.
w1	Interval of first w line.
w2	Interval of second w line.

Output Arguments:

NONE

Function Output:

Average distance between lines.

A.7.18 insert_u_knot - Insert knot into u knot sequence

Purpose:

Inserts a knot into the u knot vector and adjusts control hull vertices accordingly.

Description:

```
void insert_u_knot(comp, knot)

        comp_data        *comp

        float            knot
```

Input Arguments:

comp	Pointer to the component's data structure
knot	Knot value to insert into list.

Output Arguments:

NONE

Function Output:

NONE

NOTE: The memory allocations must be increased using the add_knot_space routine before calling this routine.

A.7.19 insert_w_knot - Insert knot into w knot sequence

Purpose:

Inserts a knot into the w knot vector and adjusts control hull vertices accordingly.

Description:

```
void insert_w_knot(comp, knot)

        comp_data       *comp

        float           knot
```

Input Arguments:

comp	Pointer to the component's data structure
knot	Knot value to insert into list.

Output Arguments:

NONE

Function Output:

NONE

NOTE: The memory allocations must be increased using the add_knot_space routine before calling this routine.

A.7.20 add_knot_space - Reallocate memory

Purpose:

Increases memory allocations to allow for new knots to be inserted into the component data structure.

Description:

void add_knot_space(comp, uw, num_knots)

comp_data	*comp
int	uw
int	num_knots

Input Arguments:

comp	Pointer to the component's data structure.
uw	Parametric direction flag 0 - u 1 - w.
num_knots	Number of knots to be inserted.

Output Arguments:

NONE

Function Output:

NONE

A.7.21 swap_uw - Swap u and w parametric directions

Purpose:

Swaps u and w parametric direction for a non-uniform cubic B-Spline surface.

Description:

void swap_uw(intdata, size, comp)

float	***intdata
int	size
comp_data	comp

Input Arguments:

intdata	Pointer to Array of intersection data in parametric format.
size	Size of intdata array.
comp	Pointer to component's data structure.

Output Arguments:

intdata	New intersection data array.

Function Output:

NONE

NOTE: If no intersection data exists, send in a blank array and a size of zero.

A.7.22 flip_u - Flip parametric u direction

Purpose:

Reverses the order of the components data in the parametric u direction.

Description:

float flip_u(intdata, size, comp)

```
float          ***intdata
int            size
comp_data      *comp
```

Input Arguments:

intdata	Pointer to Array of intersection data in parametric format.
size	Size of intersection data array.
comp	Pointer to the component's data structure.

Output Arguments:

intdata	New intersection data array.

Function Output:

Maximum parametric value before reversing u direction.

NOTE: If no intersection data exists, send in a blank array and a size of zero.

A.7.23 flip_w - Flip parametric w direction

Purpose:

Reverses the order of the components data in the parametric w direction.

Description:

float flip_w(intdata, size, comp)

float	***intdata
int	size
comp_data	*comp

Input Arguments:

intdata	Pointer to Array of intersection data in parametric format.
size	Size of intersection data array.
comp	Pointer to the component's data structure.

Output Arguments:

intdata	New intersection data array.

Function Output:

Maximum parametric value before reversing w direction.

NOTE: If no intersection data exists, send in a blank array and a size of zero.

Vita

The author was born on January 25, 1966 in Saginaw, Michigan. Upon graduating from Western Michigan University in Kalamazoo, the author left his home state of Michigan to pursue a Master's degree at Virginia Tech. Now, once the author graduates, he plans to move back north to Michigan and start a career in Engineering with Johnson Controls Automotive Systems Group.

[illegible]