USACERL Technical Manuscript P-90/34
September 1990

2

US Army Corps
of Engineers
Construction Engineering
Research Laboratory

AD-A227 158

An Architectural Modeler and Its Incorporation Into the Design 4D Program

by
Laura S. Bond-Harris

A common user complaint about computer-assisted design (CAD) programs is that the software is marketed as a design tool but does not help significantly in the building design process, which is a decisionmaking activity. Design 4D is an architectural modeler that uses 3D geometry and a seamless database manager to address the needs of Schematic Design. The user interface of Design 4D provides the architect with several new capabilities: to sketch in perspective, to alternate between two- and three-dimensional views, and to use a varied set of layout tools that work equally well when orientated to any axis.

This manuscript gives an overview of the three-dimensional interface, the display manipulation within Design 4D, and the modeling implementation developed for this study. Also examined are the geometric database and additional modeling data structures used by Design 4D.

90 10 01 040

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1990	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE An Architectural Modeler and Its Incorporation Into the Design 4D Program	5. FUNDING NUMBERS IDIR
6. AUTHOR(S) Laura S. Bond-Harris	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Construction Engineering Research Laboratory (USACERL) PO Box 4005 Champaign, IL 61824-4005	8. PERFORMING ORGANIZATION REPORT NUMBER USACERL TM P-90/34
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Copies are available from the National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A common user complaint about computer-assisted design (CAD) programs is that the software is marketed as a design tool but does not help significantly in the building design process, which is a decisionmaking activity. Design 4D is an architectural modeler that uses 3D geometry and a seamless database manager to address the needs of Schematic Design. The user interface of Design 4D provides the architect with several new capabilities: to sketch in perspective, to alternate between two- and three-dimensional views, and to use a varied set of layout tools that work equally well when orientated to any axis.

This manuscript gives an overview of the three-dimensional interface, the display manipulation within Design 4D, and the modeling implementation developed for this study. Also examined are the geometric database and additional modeling data structures used by Design 4D.

14. SUBJECT TERMS Design 4D computer-aided design architecture			15. NUMBER OF PAGES 40 16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

FOREWORD

The work discussed in this manuscript was performed as partial fulfillment of the requirements for a Masters of Computer Science at the University of Illinois at Urbana-Champaign. Funding for this investigation were provided under the In-Division Independent Research (IDIR) program.

This research was made possible by the development of the Design 4D program by the U.S. Army Construction Engineering Research Laboratory (USACERL) Facility Systems Division (FS). The Principal Investigator was Laura S. Bond-Harris. USACERL personnel involved in the project included: Kenneth H. Crawford, Beth A. Symonds, Jeffrey S. Heckel, Eric D. Griffith, and Darlene Melton. Special acknowledgement is given to Professor Herbert Edelsbrunner, of the University of Illinois at Urbana. Dr. Michael J. O'Connor is Chief, USACERL-FS. The USACERL technical editor was William J. Wolfe, Information Management Office.

COL Everett R. Thomas is Commander and Director of USACERL, and Dr. L.R. Shaffer is Technical Director.

CONTENTS

1 INTRODUCTION

Historical Background

The U. S. Army Construction Engineering Research Laboratory (USACERL) was founded in 1969 by an act of Congress. Champaign, Illinois was selected as the research laboratory site to foster a cooperative research environment with the University of Illinois at Urbana-Champaign. USACERL develops innovative engineering techniques and state-of-the-art technology to support military construction efforts. This research helps Army engineers manage the construction, operation, and maintenance of Army installations. Part of this overall research effort includes computer-aided systems as they apply to the construction process. This aspect is being investigated by the Architectural Design and Management Team of the Facility Systems Division. The research discussed here was incorporated within Design 4D, a microcomputer based architectural design system developed by the Architectural Design and Management Team at USACERL.

Users of computer-aided drafting (CAD) systems commonly complain that while CAD software is often marketed as a design tool, it does not significantly help in the building design process. Current systems are of greatest use in developing contract drawings. This is a documentation activity of a finished design process. The design process is a decisionmaking activity which has additional needs that are not addressed in CAD systems. To be a decisionmaking tool, a computer-aided architectural design (CAAD) system needs several capabilities that must be successfully married. These capabilities include easy input techniques to replace the designer's sketch paper and pencil, and the ability to visually model objects in three-point perspective to replace the designer's cardboard model. In addition, an object-oriented database that manipulates textual as well as geometric attributes allows the designer to create instances of geometric objects.

Design 4D was developed at USACERL to address the needs of an architect, or any designer, during the concept design phase. Design 4D is a microcomputer-assisted architectural modeling system that has a unique three-dimensional (3D) user interface and a fourth "dimension," an object database manager. It provides several new capabilities to the architect. First, sketching may be done entirely in perspective if the designer chooses, or by alternating between 2D and 3D sketching as desired. The 3D sketching interface is not limited to extrusions of 2D shapes and line segments. A set of 3D layout tools is provided that works equally well in any drawing plane (xy, yz, xz or any user-defined plane). Second, Design 4D allows designers to easily manipulate the view of the model. By dynamically rotating the perspective view, a designer can see a "bird's eye view" or see things from the vantage point of a person walking by on the street. This not only replaces a typical model, but surpasses the cardboard model by allowing the designer to see a more detailed view. Lastly, Design 4D's object-oriented database manager provides flexible manipulation of textual and geometric information. The integration of the object-oriented database and geometric engine permits the user to sketch a schematic design and then to incorporate its geometric objects into the database.

During fiscal year 1988, the Design 4D system, including online help and user's manual, was distributed to four test sites. These test sites were Corps of Engineers District and Division offices. Two of the test sites had existing turnkey workstation-based CAD systems and two had microcomputer-based drafting systems. Architects at the test sites used the program on projects based on their current workload. The modeler was evaluated on its ease of use, its ability to help the user analyze early design alternatives, and its ability to share data with other tools. The architects felt that Design 4D offered three advantages. First, the majority felt it addressed the conceptual design phase in the architectural process. Second, there was a consensus that Design 4D was helpful in 3D visualization of design alternatives. Third, Design 4D proved to be an easy sketching tool. During fiscal year 1989 Design 4D is being distributed to an additional five test sites.

The 3D interface technology is being transferred to private industry through a Cooperative Research and Development Agreement (CRDA) signed in June 1988 with Isicad, Inc. of Anaheim, CA. They are incorporating a subset of Design 4D's 3D-sketching approach and dynamic interfaces within their commercially available microcomputer based CAD system, "Cadvance."

Approach and Objective

A unique approach was taken to incorporate a modeler into Design 4D. Taking advantage of Design 4D's interface, a user can quickly sketch line segments from which a model is then derived. By performing a depth-first search on the existing line segments, it is possible to determine all the polygons. These are then stored in a boundary representation model of cell complexes. This model is an implementation of the ideas presented by Dobkin and Laszlo in "Primitives for the Manipulation of Three-Dimensional Subdivisions." This model saves the edges that define a polygon (a facet) in an "edge-facet ring." It also links the polygons (facets) that share a common edge in a "facet-edge ring." This information, along with the polygon's color, planar orientation, and additional information, is saved. Using this definition of polygons, it is possible to display hidden line drawings and ambient light renderings. By making these techniques available during the early concept design phase, designers are able to make informed decisions between alternatives.

This paper will give an overview of the three-dimensional interface, the display manipulation within Design 4D, and the modeling implementation developed by the author. The geometric database used will be examined, along with the additional modeling data structures. The object oriented database was not part of this research project and will not be discussed here.

2 DESIGN 4D - A COMPUTER-AIDED ARCHITECTURAL DESIGN SYSTEM

Hardware Platform

The minimum hardware platform required for Design 4D is an AT compatible microcomputer system with an 80286 processor and 640K RAM. A math coprocessor is used to speed up execution time. The Professional Graphics Array Standard, a high-speed graphics controller board, is used to implement the graphic calls in on-board memory. A three-button mouse or digitizer tablet is used as an input device. The software was developed in the C language. Its execution speed allowed the development of a three-dimensional perspective sketching system with good response time.

The Three-Dimensional Drawing Interface

To encourage an architect to design in three dimensions, a highly flexible and interactive user interface was developed. The user can switch between a two-dimensional and a three-dimensional drawing mode with a click of a button. To make full use of three-dimensional drawing, a variety of three-dimensional interactive drawing cursors have been developed. These include: *rubber banding*, *rubber rectangles*, cubes, stretched polygons, cylinders and circular arcs. These cursors are also available in the two-dimensional drawing mode.

Several aids help the designer work in perspective without becoming "lost in space," which frequently occurs in other systems. The primary display technique for locating oneself in 3D space is a background perspective grid (Figure 2-1). Design 4D employs a "cross-sectional" approach to visually facilitate 3D drawing. A current drawing plane is displayed within the 3D perspective grid, located along either the x-, y-, z-axes, or a user-defined plane located at any angle in the perspective space. Figure 2-2 shows the zy drawing plane, which is perpendicular to the x-axis, and figure 2-3 shows the background perspective grid. A user defined plane, (Figure 2-4), allows for a sloped drawing plane that simplifies the generation of roofs and other slanted entities. One of several available cursors resides on the current drawing plane giving the user a variety of sketching techniques.

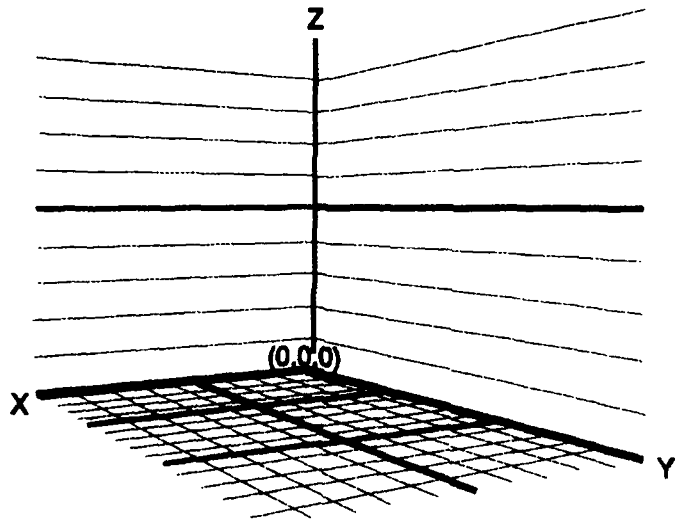

Figure 2-1 Design 4D's background perspective grid.

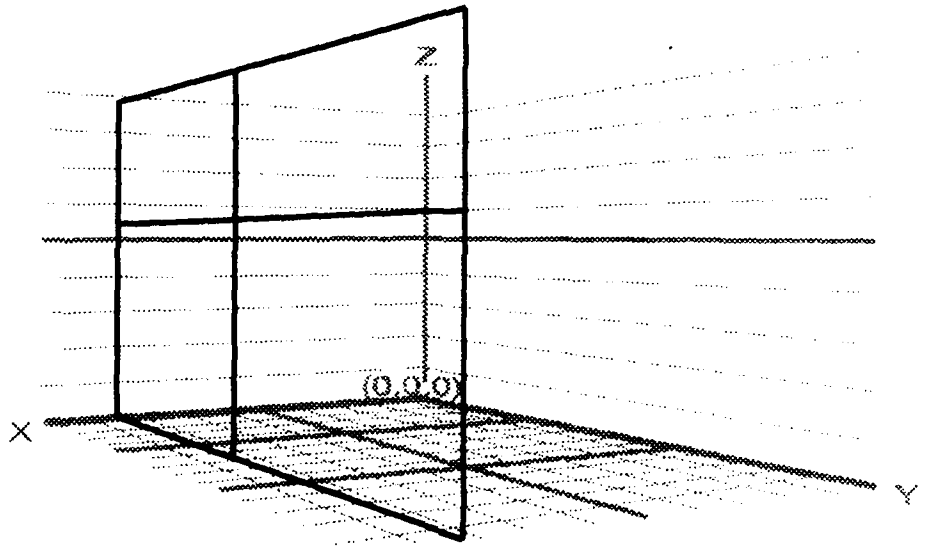

Figure 2-2 Background grid with the zy plane as the current drawing plane with a simple cursor on the plane.

The current drawing plane can be dynamically moved along any principal axis or user-defined plane. Using a graphical "tab," the user can move the drawing plane easily through space, jumping to the next ascending or descending line segment end point. This gives the user easy access to the next end points on either side of the drawing plane. When a point is found, it is highlighted with a tick mark, as are all other points on the new drawing plane (see Figs. 2 - 3 and 2 - 4).

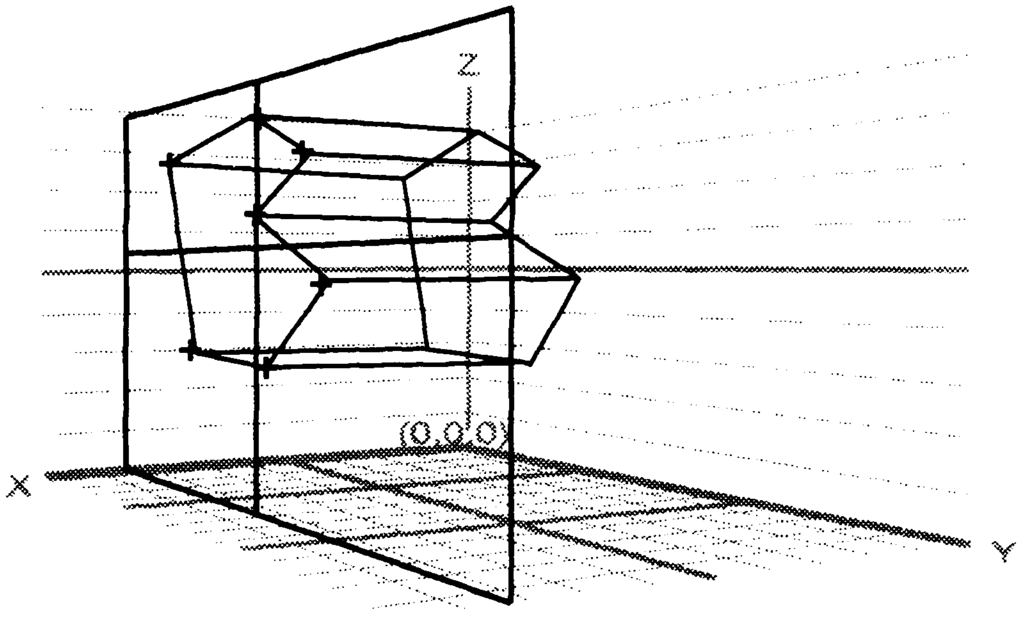

Figure 2-3 Current zy drawing plane incident to the object.

All the object's points on the current drawing plane are highlighted with tick marks.

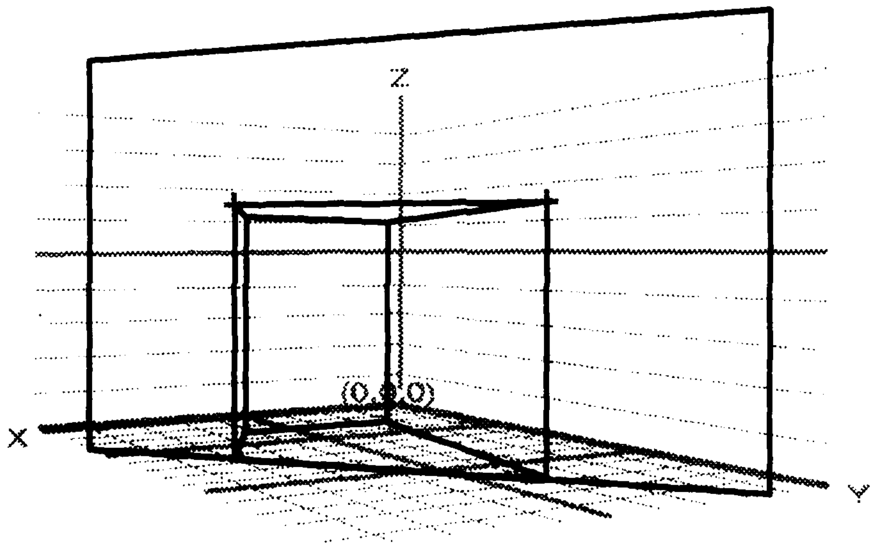

Figure 2-4 User-defined plane as the current drawing plane.

The Display Manipulation Interface

Design 4D starts with a default 3D perspective view. As the design develops, an architect can see the model from any other view. The background perspective grid may be defined by a set of commands including changing the drawing x-, y- and z-extents, and grid spacing. To manipulate the 2D or 3D view, the user can zoom and pan the display. One of the most powerful 3D view features is the ability to dynamically rotate the background perspective grid (see Fig. 2 - 5). As designers evaluate alternatives, they are able to go from a bird's eye view to an eye level view by changing the viewing cone angle and the focus point. In this way, most 2D or 3D views are achievable.

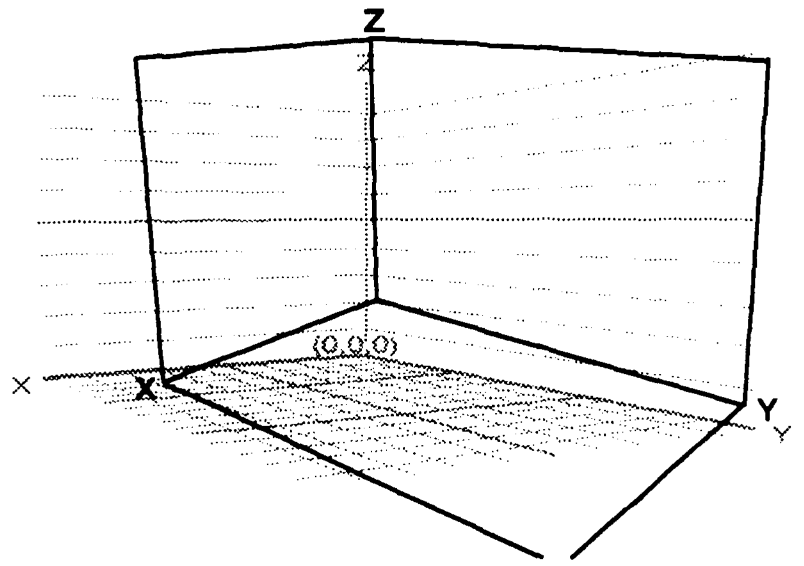

Figure 2 - 5 Possible new grid superimposed on background perspective grid in the dynamic rotation option.

3 EXISTING GEOMETRIC OBJECT DATA STRUCTURES

The Geometric Objects Data Structure

Design 4D's line segment and point data structures were designed to support its easy sketching interface. The geometric data had to be organized to allow quick moving of the current drawing plane from the current point to the next ascending or descending point or line segment end point. In addition, the system intersects each new line segment with existing line segments when it is drawn. This requires that the data structure support a search on the existing line segments to find all the intersecting line segments.

To simplify data manipulation, Design 4D accesses the same geometric data structu s regardless of whether the user drew the line segment in a two- or three-dimensional sketching mode. For each point and line segment end point, an x-, y-, and z-coordinate is stored. When displaying a two-dimensional view, Design 4D converts the three-dimensional geometric objects into the required two-dimensional objects and displays them. In doing this, it simulates the mental process that architects use when developing two-dimensional floor plans or sections. Using this technique, it is not necessary to maintain a separate database for both the two- and three-dimensional objects. Design 4D maintains one set of geometric information that is interpreted for each display.

Because of the memory limitations of the hardware platform, all geometric data is kept in secondary storage to minimize main memory usage requirements. DOS permits a maximum 640 Kbytes core memory in which to execute an application. This core memory must also accommodate the DOS operating system and any device drivers, effecti ely limiting the available executable RAM to 512 Kbytes. Design 4D stores the geometric data on secondary storage, and accesses it through low-level input-output methods. This assures that Design 4D has the main memory available that is required to run complex graphic routines. Because access to secondary storage is slower than main memory, this approach implies slower runtime response, which is a disadvantage to this approach. The decision to maintain Design 4D's 3D geometric data on secondary storage impacted the geometric object data structure that is discussed in this section. These decisions also affected the model representation that is a subject of this paper and is discussed in the following section.

Design 4D's existing data structure is distributed among five direct access files. As the user starts a new project, these files are created with the appropriate starting values. The files include the points file (called xyzcoords.dat), the line segments file (called lines.dat), and one "slice" file for each dimension (xslice.dat, yslice.dat, and zslice.dat). To implement the modeler, two more files are needed. These are the polygon headers file (called polygon.dat), and the edge facet pairs file (called ef.dat). Any locations or pointers discussed in this text refer to file offsets that can be accessed with low level input and output commands such as the C command "lseek"

The Slice Data Structure

Design 4D conceptually breaks up space into "slices" of a set width, for example 5 ft, that run perpendicular to the primary axes. All the geometric objects within a slice are "chained" together in a double-linked list. Each object keeps pointers to the spatially previous and next objects. The previous object has the largest x, y, or z coordinate, depending on the linked list involved, that is still smaller than the current object's coordinate. Also, the next object in the list is the one with the next largest x, y, or z coordinate. With slices running perpendicular to the three primary axes, it is possible to store each object in three linked lists. In actual implementation, some of the linked lists usually turn out to be redundant and are eliminated to reduce the time required to maintain the lists during insertion and deletion procedures. While the geometric objects are linked together in spatial order using pointers, they are physically stored in chronological order. Therefore, they cannot be accessed using a binary search algorithm. To reduce the search time required, indexes that point to first, middle, and last objects in the linked lists in each slice are maintained. By first using the index for a slice, and then following the linked list of objects, all the objects in a particular slice are accessible.

While storage of objects in secondary memory allows for a more complex CAAD system, it makes it important to reduce the number of disk accesses when searching for a geometric object. Each noncontiguous disk access requires seek and read time. As will be seen below, the slice index records contain additional information to reduce search time of the line segment and point files. The slice indexes for each principal axis are stored contiguously on secondary storage. Therefore, it is more efficient to search the block files than the line segment or point files. This affects the data structures that will be discussed below.

Each slice is given a slice number and is set to the width of the variable "*mesh_fineness.*" An x-slice with slice number i includes points whose x coordinates lie between (i - 1) * mesh_fineness and i * mesh_fineness. Line segments whose end points lie in a slice i are also included in the slice.

The y- and z-slices also follow this rule. For example, if the mesh_fineness is set to 5.0 ft, the first x-slice contains points and line segments with x coordinates between 0.0 ft and 5.0 ft. The indexes of the slices perpendicular to the x axis are stored is the xslice.dat file. The first y-slice references the points and line segments whose y coordinates range from 0.0 ft to 5.0 ft. Additionally, the first z-slice references the points and line segments whose z coordinates range from 0.0 ft to 5.0 ft. To eliminate unnecessary accessing of the slices' indexes, the minimum and maximum x, y, and z extents of all existing line segments are kept in main memory.

The information stored in the slice records allow Design 4D to find line segments and points starting along any axis. The number of slices allowed is set at 800 to limit the amount of information stored. The mesh_fineness is a global variable that can be changed to allow for a larger world coordinate space. Presently, mesh_fineness is set to 5.0 ft allowing objects whose coordinates lie between 0.0 ft and 4000.0 ft. Mesh_fineness can be changed, but projects developed with different mesh_fineness values have incompatible geometric data structures.

For each slice certain information is maintained. This information provides for fast access into the geometric object data. It does this by providing an index into the geometric objects linked list if it is not empty. The specifics of the slice data records will be discussed below regarding the point and line segment data structures as is appropriate.

The Point Data Structure

The point file is used to store points that the user has created while developing a project. The point file is searched to determine which points are on the current plane, and to tab to the next smallest or largest point. These functions determine the information saved for each point.

When users draw two- or three-dimensionally in Design 4D, they select a current drawing plane. The available planes are the planes parallel to the x- and y-axis, x- and z-axis, or z- and y-axis, and the user-defined planes, planes that can be located at any angle to the principal planes. Users can "tab" this current drawing plane perpendicularly through space to the next point in other directions. The point data is organized to assure quick access to these points from the current drawing plane location. This is one of Design 4D's unique capabilities that lets the user quickly orient him/herself in 3D space.

The slice index records contain the following fields to assure timely access of the next points in either direction of the current drawing plane. These fields are also used to speed up the adding and deleting of the points.

number_points	-	the number of points in the x/y/z slice.
loc_first_point	-	location of point with smallest x/y/z coordinate in the x/y/z slice.
loc_mid_pt	-	location of point with median x/y/z coordinate in the x/y/z slice.
val_mid_pt	-	x/y/z value of the point with the median x/y/z coordinate in the x/y/z slice.
loc_last_point	-	location of point with largest x/y/z coordinate in the x/y/z slice.

Figure 3-1 Part of the slice data representation that refers to points.

When tabbing the current drawing plane, or adding or deleting a point, Design 4D must find the next point. Searching for the next point can be reduced by inspecting the number_points in a slice. If the number of points in a slice is 0, this slice does not need to be searched, and the next slice can be evaluated. When adding objects, Design 4D saves all geometric object coordinates in double floating-point precision. This was a result of the graphics hardware platform used and sped up execution time. Storing the coodinates in this format created some additional searching problems. Points on a slice's boundary could lie within either slice, depending on their floating point representation. For example, a coordinate of 4.9999999 would be included in slice 1, while a coordinate of 5.0 would be included in slice 2. When line segments are being connected to form polygons, line segment endpoints with both of these coordinate values would need to be evaluated if the last point's coordinate was 5.0. The extra searching this creates can be reduced by looking at a smaller box, represented by a x-slice intersecting a y-slice intersecting a z-slice, and seeing if it contains any points. This is easily accomplished by testing if the x-slice number_points = 0 and the y-slice number_points = 0 and the z-slice number_points = 0. When this is true, a cube of mesh_fineness * mesh_fineness * mesh_fineness dimensions is eliminated from consideration reducing the worst case of unsuccessful search. (This technique is particularly useful when adding and deleting points.)

The remaining point information in the slice index files is used to find the most efficient starting access place in the point data files. By comparing a point with the value of the slice's median point, it is easy to determine which half of a slice the point is located in. By assuming that the points are uniformly distributed, it is possible to estimate which quarter of the slice the point may lie in. This further reduces the necessary search. By starting at either the first, middle, or last point, and searching the linked list in the selected direction, it is possible to reduce search time for a point from n, the number of points in the slice, to n/4.

The point object information is kept in the "xyzcoord.dat" file. Each point record contains the three-dimensional coordinates, as well as links to the next ascending or descending point. The following information is part of what is included.

x, y, z	-	coordinates of the point.
x(/y/z)_slice_chain	-	location of the next ascending x/y/z point in this x,y and z slice. (Each point belongs to 3 of these chains.)
x(/y/z)b_slice_chain	-	location of the next descending x/y/z point in this x/y/z slice. (Each point belongs to 3 of these chains.)
nı .nber_of_uses	-	number of line segments that share this point as an endpoint.
purge	-	TRUE if this is a purged point. FALSE if this is an active point.

Figure 3-2 Part of the Point Data Representation.

The Line Segment Data Structure

As users draw two- or three-dimensionally in Design 4D, they create line segments located in three-dimensional space. As each line segment is added, the existing intersecting line segments are found and segmented as required. Users can delete or drag line segments, among other operations. The line segment data structure must accommodate these functions.

Line segment searching in Design 4D is a more complex problem than point searching. All new line segments are intersected with existing line segments which requires a search of the line segments for intersections. It is not sufficient to evaluate only the line segments within the search box that is defined by the new line segment to find all of the intersecting line segments. The search box must be expanded to include line segments that intersect the new line segment, but whose end points do not lie within the search box. The slice index records include information that allows the selection of a search box with a small number of total disk accesses, on the average. After the existing line segments are intersected, the problem is reduced to finding the appropriate place in the linked list to add or delete the line segments, similar to point searching.

As with point searching, each slice header includes information to allow elimination of the whole slice of space. The following elements help determine if there are any objects in the slice:

number_lines	-	the number of line segments whose minimum x/y/z endpoint is in the x/y/z slice.
number_linese	-	the number of line segments whose maximum x/y/z endpoint is in the x/y/z slice.
min_cross_chain	-	lowest slice (along the same x/y/z axis) that contains a line segment with one end point in that slice and the other endpoint in this current slice.
max_cross_chain	-	highest slice (along the same x/y/z axis) that contains a line segment with one end point in that slice and the other endpoint in this current slice.
Linked List Starting Addresses		
loc_first_linex	-	location of line segment with smallest x coordinate in this z slice. This point is at the beginning of the linked list of line segment endpoints with minimum x's in the z slice.
loc_first_linez	-	location of line segment with smallest z coordinate in this x slice. This point is at the beginning of the linked list of line segment endpoints with minimum z's in the x slice
loce_first_linex	-	location of line segment with largest x coordinate in this z slice. This point is at the beginning of the linked list of line segment endpoints with maximum x's in the z slice.
loce_first_linez	-	location of line segment with largest z coordinate in this x slice. This point is at the beginning of the linked list of line segment endpoints with maximum z's in the x slice.
loc_mid_linex(/z)	-	location of line segment with median x or z end point coordinate in the linked list of the minimum x or z line segment endpoints in the z or x slice.
loce_mid_linex(/z)	-	location of line segment with median x or z end point coordinate in the linked list of the maximum x or z line segment endpoints in the z or x slice.

Figure 3-3 Part of the slice information that refers to line segments.

If the number_lines (or, if searching in a descending direction, number_linese) is 0, this slice is empty, and does not require searching. Also, each slice record maintains two fields, the min and max cross chain, that represents the minimum and maximum parallel slice that any line segment starting in the particular slice extends to. By evaluating this cross chain, a slice can be eliminated from the search for intersecting line segments. In a case where line segments are uniformly distributed, this would not be of great assistance. However, architects' models and drawings usually are concentrated on certain planes perpendicular to one of the principal axes. With this type of distribution, the above information reduces the search for intersecting line segments.

The location information in the slice index files is used to find a good starting place for accessing the line segment data files. To do this, it is first necessary to determine which endpoint belongs in the x, y, and z minimum and maximum endpoint chains. To determine which of the spatial chains an endpoint belongs in, the x, y, and z values are each compared with the values of the other endpoint. Once the correct spatial chains have been determined, the minimum (and maximum) chains can be accessed using the slice record indexes to the first, middle and last objects in each chain.

The line segment object information is kept in the "lines.dat" file. Each line segment record contains the three-dimensional coordinates, as well as links to the next ascending or descending point. The three-dimensional coordinate data is redundant information that is stored to reduce access into the point data files. The following information is part of what is included:

xb, yb, zb	-	x, y, z coordinates of the first end point the user entered when creating the line segment.
xe, ye, ze	-	x, y, z coordinates of the second end point the user entered when creating the line segment.
begin_slicex(/y/z)	-	slice number of the x/y/z coordinate of the first end point entered.
end_slicex(/y/z)	-	slice number of the x/y/z coordinate of the second end point entered.
begin_point	-	location in point file of the first point entered.
end_point	-	location in point file of the second point entered.
xy(/z)_beg_slice_chain	-	location of the line segment with the next biggest y/z coordinate in this x-slice.
xy(/z)_end_slice_chain	-	location of the line segment with the next smallest y/z coordinate in this x-slice.
zx(/y)_beg_slice_chain	-	location of the line segment with the next biggest x/y coordinate in this z-slice.
zx(/y)_end_slice_chain	-	location of the line segment with the next smallest x/y coordinate in this z-slice.
purge	-	TRUE if this is a purged line. FALSE if this is an active line segment.

Figure 3-4 Part of the line segment data representation.

As discussed above, the line segments are linked to allow spatial searching. The minimum endpoint chains that link the endpoints with the minimum x, y, and z points are called the beg_slice chains. The maximum endpoint chains that link the endpoints with the maximum x, y, and z points are called the end_slice chains.

This concludes the brief conceptual overview of the geometric object data structures present in Design 4D when this research began. Some of the specifics were idealized here. In actual implementation, some of the information was eliminated to decrease the time required to add and delete line segments, and points. For further discussion of the three-dimensional linked list created between the point and line segment objects, see "Three-Dimensional Architectural Modeling on a Microprocessor" by Kenneth H. Crawford and Laura S. Bond.

4 SOLID OBJECT DATA STRUCTURES

Immediate Goals of Model Data Structure

One of Design 4D's major achievements has been the development of an easy and consistent sketching interface. This interface reduces the time required to learn the system and encourages the use of Design 4D during the conceptual design phase, as well as during the design documentation phase. While a typical PC-based CAD program requires a user to enter surfaces with a surface drawing interface, Design 4D has no special surface drawing tools. Design 4D allows the user to sketch a line drawing of anything desired, and from that creates a model. Since all line segments are intersected with each other in Design 4D, it is possible to find all the polygons in the sketch and save them as surfaces. This approach naturally leads to a boundary representational (BREP) model instead of a constructive solids geometry model. Design 4D uses the resulting model to create ambient light renderings and hidden line removal drawings for the architect. With a BREP model, it is possible to use these display techniques without overloading the selected hardware platform.

Each surface drawn can have a varying number of edges. This necessitates a data structure that can accommodate both fixed and varied information lengths. This is resolved by storing information of fixed length, the polygon "header" information, in one file ("poly.dat") and the actual edges, which are variable in number, in another file ("ef.dat"). Information that pertains to an individual surface is stored in the polygon header. Information unique to each edge is stored with the individual edge. Duplication of data is kept low to limit any problems with inconsistent updating of information. The polygon header information keeps an index into the edges and the edges maintain an index into the polygon header record. To assist in locating geometric objects and to reduce the number of data file accesses, the x, y, and z extents of each surface are saved in the surface header files.

The line segment and point data structures discussed in Section 3 were part of Design 4D before incorporation of the solid-object BREP model. While the point and line segment object update times were at an allowable level, they were close to the maximum time acceptable to the user. Any extra overhead necessary to create a solids model while sketching would not be tolerated by the users. Therefore the solids model building function was offered as a separate menu option selected by the user. To minimize update time during line segment addition and deletion, the solids model representation of edges was kept separate from the line segment representation created while sketching. This does not create any conflicts between inconsistent data because the graphic routines manipulate either surfaces or line segments.

Dobkin-Laszlo Quarter Node Data Structure

In their paper "Primitives for the Manipulation of Three-Dimensional Subdivisions," David P. Dobkin and Michael J. Laszlo present a data structure developed to model three-dimensional cell complexes. This data structure, called a quarter node, is analogous to the two-dimensional quad-edge data structure developed by Guibas and Stolfi that is used to model two-dimensional objects. Dobkin and Laszlo consider a polygon-edge pair as an atom in their proposed data structure. The data structure links the polygon-edge pair atom to two polygon "rings" and two edge "rings".

An edge ring is the sequence of edges that define the boundary of a polygon. This ring can be oriented two ways around a polygon, either clockwise or counterclockwise, as one of the sides of a polygon is viewed. To determine which side of the polygon is being discussed, the following convention is adopted. If the viewer is located at the target of the normal vector that defines the polygon plane, and looks towards the normal vector's source, he/she will be looking at the polygon side with the clockwise edge ring orientation. In this implementation, the clockwise orientation is considered the primary edge ring, and the counterclockwise orientation is referred to as the symmetrical edge ring. These edge rings are also called the edge-rings of the facet (see Fig. 4 - 1).

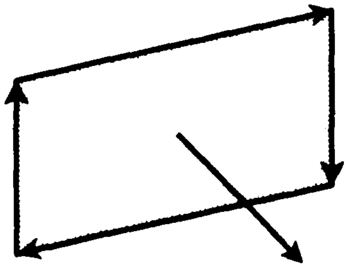
Figure 4-1 Surface's edge ring.

A polygon ring is defined as the cycle of polygons that share an edge. These polygons also form a conceptual ring around the edge and have either a clockwise or counterclockwise orientation. To determine which direction is clockwise, the viewer places him/herself at the target of the directed edge. Similar to the edge-ring convention, when the viewer looks down the directed edge towards the source, the polygons located in a clockwise direction around the directed edge are the primary polygon ring. The polygon ring oriented in the counterclockwise orientation from this view is referred to as the symetric polygon ring. These polygon rings are also called the facet-rings of an edge (see Fig. 4 - 2).

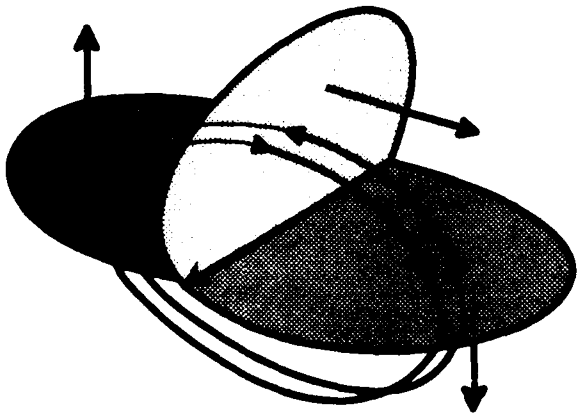
Figure 4-2 Edge's facet ring.

Each polygon-edge pair belongs to a unique combination of edge rings and polygon rings. Dobkin and Laszlo's paper discusses methods to manipulate these polygon-edge pairs including splice, meld, decomposition, and triangulation operations. The conceptual ideas of the polygon-edge pair and its related edge rings and polygon rings were implemented in Design 4D's solids modeler. The following sections discuss how these ideas are specifically implemented. The additional algorithms presented in Dobkin and Laszlo's paper are left for future research projects.

Depth-First Search to Find Valid Polygons

When Design 4D creates a solids model from the user's drawing, it determines all the polygons in the user's drawings. An acceptable polygon for the database does not enclose any other polygons in its plane that share any edges. This assures that any polygon is only represented once in the object database. Any polygon that represents the universe minus a polygon subset is not allowed. These polygons are easily eliminated by discarding any polygons that do not pass the following test:

(number of polygon edges - 2) * pi = sum of the polygon's angles.

Restricting the acceptable surfaces simplifies the procedures that identify duplicate polygons. Each polygon can be uniquely identified by the unit normal vector that defines the plane it resides on, and one directed edge from the polygon's boundary. This can be shown by considering the combinations of the normal and the edge. Usually, if the normal vectors are not the same, the two polygons do not reside on the same plane, and cannot be the duplicates. When the polygon normals are opposite of each other, the two polygons are on the same plane, but viewed from opposite sides of the plane. In this case a closer examination of the polygon edges is needed. By looking at any shared undirected edge, it can be determined if these are two unique polygons, or opposite sides of the same polygon. Because each polygon's primary edge ring is oriented clockwise, when the shared edge and normals are oriented in opposite directions, these two polygons are the same polygon. Obviously when the plane normals are the same, the two polygons do reside on the same plane. If the two polygons do not share any common edges, they are not duplicate polygons. Sometimes the two polygons do share an undirected edge. Since an acceptable polygon cannot enclose any other polygon, and the edges must be oriented clockwise, if the shared directed edge is oriented in the same direction, the two polygons are duplicates. In Figure 4 - 3 two polygons are shown that have the same surface normal and share an edge. Since the shared edge is oriented in opposite directions within the polygon's edge ring, the normals and edges define two unique polygons. This technique is used below to prevent entry of duplicate polygons into the object database.

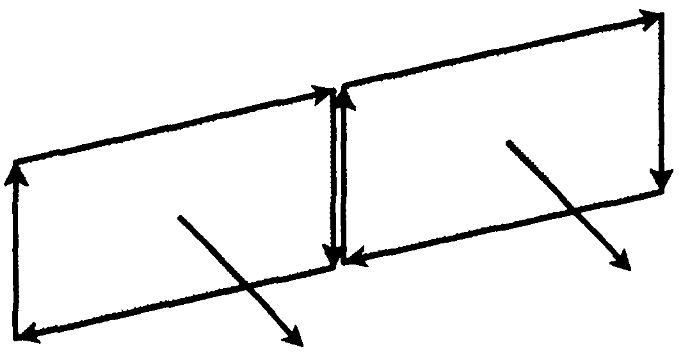

Figure 4-3 Two polygons with the same plane normal that share an edge.

The challenge of finding all acceptable polygons in the user's drawings was met by developing a depth-first search tree. The algorithm that creates the search tree uses techniques discussed below to prune redundant paths. The algorithm considers each undirected edge in the object database to assure that all possible polygons are found. First, the edge is oriented in the direction it was originally drawn in. This directed edge is used as a starting point of a depth-first search of the line segments. The first step in the search is to find a line segment with an endpoint that matches the first line segment's end point and defines a plane. After this is accomplished, the search continues to find successive connected line segments that lie on this plane. Each successive connecting line segment is placed on an *edge stack,* a LIFO queue of the polygon's edges. The process is repeated until either a polygon is found or there are no more matching line segments. If a facet is found, and it is not a duplicate polygon, it is added to the object database. This procedure and the data structure used are discussed in the next section. If this is a duplicate facet, the simple facet with directed edge has already been found. Because of the definition of an acceptable polygon, once a polygon is discovered for a given plane definition and given directed edge, the search is complete for this set of conditions. Typical of a depth-first search, if a dead end is found, edges are popped off the edge stack and the search is continued. When this process is complete, it is necessary to reverse the line segment orientation and perform this process again.

The first step of the depth-first search algorithm is to find two line segments that define a plane. In two-dimensional space, each edge can only be part of two facets. In three-dimensional space, an edge and a normal to the polygon define a unique plane. Since each edge can be paired with arbitrarily many normals, each edge can also be a member of arbitrarily many facets. To find all the possible plane definitions, it is necessary to examine each line segment that is connected to the target endpoint of the directed edge. By looking at three points from these two line segments, a plane definition can be derived using the equation:

$$Ax + By + Cz + D = 0$$

where :

$$-A = y1(z2 - z3) + y2(z3 - z1) + y3(z1 - z2)$$
$$B = z1(x2 - x3) + z2(x3 - x1) + z3(x1 - x2)$$

$$-C = x1(y2 - y3) + x2(y3 - y1) + x3(y1 - y2)$$
$$D = x1(y2*z3 - y3*z2) + x2(y3*z1 - y1*z3) + x3(y1*z2 - y2*z1)$$

Figure 4-4 Equations for a normal defining a plane.

When the two line segments are parallel, this part of the searching is discontinued. This is possible because the line segment searching will eventually find any polygons the two parallel line segments may be in. This process may seem to require an extensive search to find all facets. Fortunately, a typical architect creates masses that are mostly oriented perpendicular to the three principal axes. While the depth first search algorithm utilized must consider any planar orientation, the task is accomplishable through the use of a *normal stack*, a LIFO queue of plane normals. When each plane definition is investigated, the plane normal is placed on the normal stack. This provides a history of possible polygons that have been found that start with this line segment. Since only one facet should be found per plane, each time a new plane definition is determined, and it is also found on the polygon normal stack, the search of this plane definition can be stopped. Also, when the directed edges' orientation is reversed, the normals on the normal stack should be reversed. This eliminates search efforts that would result in discovering the flip side of the polygons found in the search of the previous orientation.

Once the first two line segments are found and the plane normal defined, it is necessary to find the successive facet edges. These edges form the polygon's edge ring. A search is performed on the line segment object database to locate all connecting line segments that reside on the possible polygon's plane. For each line segment found, the counterclockwise angle from the current edge to the found edge end is determined. The line segment with the smallest angle is selected from all these edges as the successor edge. This assures that the polygon found will not enclose any other polygon. The minimum angle, along with the edge information, is saved on an edge stack. If the depth-first search needs to backtrack because of an unsuccessful search path, the search algorithm will look for the next successive edge with the least angle that is greater than this minimum angle. Once the next line segment is found, it is placed on the edge stack. It becomes the current line segment and is used to continue the search.

As the search for connecting line segments continues, there will be some dead ends that require backtracking. A dead end is reached when there are no line segments that connect to the current line segment's end point. It is necessary to pop the top edge off of the stack and discard it. The search can continue, using the edge that is now on top of the stack. When an edge is discarded, it is necessary to change the angle constraints of the search for the next line segment. Each time a search starts, the connecting line segment selected is that with the minimum counterclockwise angle with the current line segment. Using the old criteria will rediscover the edge just discarded. By resetting the angle constraints to the minimum angle greater than discarded edge angle, the best edge is found. As dead ends are found, the edges are popped off the stack and evaluated, until, in the unsuccessful case, no edges are left. Now, the original edge's orientation is reversed, and the process is repeated. If both directions have already been investigated, the next starting line segment is considered.

There is a subtle dead end that can be reached in the search process. When an edge loops back to an earlier edge in the facet, aside from the original edge, the polygon encloses another. This unacceptable facet is checked for during the push operation onto the edge stack. The enclosed polygon has either already been found, or will eventually be found by the depth-first search. Since it will eventually be discovered, it is discarded now. Each of the edges that is part of the enclosed polygon is popped off the edge stack. As with the other dead end condition, the next line segment searched for has angle constraint of a minimum angle greater than discarded edge angle.

Each time a connecting line segment on the polygon plane is found, it is first evaluated to see if it closes an interior polygon. If it does, a polygon has been found, and the search for this edge plane pair is done. If this edge is not a duplicate, it is added to the database. This procedure is discussed later. Every line segment, except the first line segment, is popped off the edge stack. When all possible planes that include this starting directed edge and a connecting edge have been investigated, the search of the line segment with this orientation is done. If only one direction of the current line segment has been searched, the line segment and the polygon normals on the normal stack are reversed, and the search repeated. If both directions have been investigated, the last line segment on the edge stack is removed, and the normal stack is emptied. The stacks are now ready for the next undirected edge search. Figure 4 - 5 outlines the depth search process just discussed.

```
/***********************************************************************
procedure          : BOOLEAN find_polygons_with_line(long int)
purpose            : Given a starting line segment, perform a depth first search to find all polygons
                     that contain  this line segment.  These polygons are added to the polygon
                     database.
 variables passed  : starting_line_loc     pointer to the starting line segment.
 value returned    : TRUE if polygons were found.
                     FALSE if no polygons found or user stopped process.
***********************************************************************/

procedure find_polygons_with_line (starting_line_loc)

long int starting_line_loc;

beginprocedure

                          /* Set up stacks - LIFO queues.*/

    allocate edge_stack;
    allocate normal_stack;

    get line segment information (starting_line_loc, &starting_line);

    for both orientations of the starting_line segment
        beginfor

                          /* If anything is on the edge stack
                          from the last search, remove it. */
```

Figure 4-5 Algorithm that performs depth-first search to locate surface.

```
while NOT EMPTYSTACK edge_stack
    pop edge_stack;
endwhile.

            /* If line segment is oriented from end point to beginning point,
            reverse surface normal definitions found on search of
            line segment oriented from beginning point to end point. */

for each member of the surface_normal_stack
    beginfor
        surface_normal_elt.a_norm = 0.0 - surface_normal_elt.a_norm;
        surface_normal_elt.b_norm = 0.0 - surface_normal_elt.b_norm;
        surface_normal_elt.c_norm = 0.0 - surface_normal_elt.c_norm;
    endfor

            /* Set values for line segment search according to
            current starting_line orientation. */

if (searching from beginning pt to end point) then
    origpt_x = starting_line.x_beg;
    origpt_y = starting_line.y_beg;
    origpt_z = starting_line.z_beg;

    nextpt_x = starting_line.x_end;
    nextpt_y = starting_line.y_end;
    nextpt_z = starting_line.z_end;

endthen
else
    origpt_x = starting_line.x_end;
    origpt_y = starting_line.y_end;
    origpt_z = starting_line.z_end;

    nextpt_x = starting_line.x_beg;
    nextpt_y = starting_line.y_beg;
    nextpt_z = starting_line.z_beg;
endif.

            /* Begin edge stack for this polygon.*/

push starting_line onto edge stack;

            /* Look at each line segment that connects to the starting line segment.
            Determine the plane normal going from origpt to
            nextpt to the connecting line segment's other end point. */
```

Figure 4-5 (cont'd.).

```
while (get_connecting_line(nextpt_x, nextpt_y, nextpt_z,
    &connecting_line, &new_pt_x,&new_pt_y,&new_pt_z))

    beginwhile
    if (parallel_lines(origpt_x, origpt_y, origpt_z, nextpt_x, nextpt_y, nextpt_z,
        new_pt_x, new_pt_y, new_pt_z))
        continuewhile;
    endif.
    get_plane_definition (origpt_x, origpt_y, origpt_z, nextpt_x, nextpt_y, nextpt_z,
         new_pt_x, new_pt_y,new_pt_z, &a_normal, &b_normal, &c_normal, &d_normal);

            /* Check and see if this plane definition is already on the
            surface_normal_stack. If so, the search of this plane definition can be stopped.
            Else, search for any surface on this plane. */


    for each member of the surface_normal_stack
        if (points_are_equal(a_normal, b_normal, c_normal, d_normal,
            surface_normal_elt.a_norm, surface_normal_elt.b_norm,
            surface_normal_elt.c_norm, surface_normal_elt.d_norm))
            continuewhile;

            /* Now we have a plane definition that needs to be searched. We want to
            to find the successive connecting line segments that have the smallest clockwise angle
            from the current_line segment to the next line segment. */

            /* Initialize values. */

    angle_btwn_lines = start_angle = 0.0;
    sum_polygons_angles = 0.0;
    current_line = starting_line;
    pt1_x = origpt_x;
    pt1_y = origpt_y;
    pt1_z = origpt_z;

    pt2_x = nextpt_x;
    pt2_y = nextpt_y;
    pt2_z = nextpt_z;

        /* Push plane definition onto surface_normal_stack. */

    surface_normal_elt.a_norm = a_normal;
    surface_normal_elt.b_norm = b_normal;
    surface_normal_elt.c_norm = c_normal;
    surface_normal_elt.d_norm = d_normal;
    push surface_normal_elt  onto surface_normal_stack;
```

Figure 4-5 (Cont'd.).

```
        /* Search for a connecting_line segment on defined plane with the minimal clockwise
        angle from the current_line segment. Continue searching until polygon is completed
        or all the possible paths have been searched and all the connecting edges have
        have been popped off the edge_stack.*/

while MORE THAN 1 member on the edge_stack
    beginwhile
    get_connecting_line_on_plane(pt1_x, pt1_y, pt1_z, pt2_x, pt2_y, pt2_z,
        a_normal, b_normal, c_normal, d_normal,
        &connecting_line, &new_pt_x, &new_pt_y, &new_pt_z,
        &angle_btwn_lines, start_angle);

        /* If the connecting line segment completes the polygon, add it to surfaces database. */

    if (found connecting line) then
        if (points_are_equal (origpt_x, origpt_y, origpt_z,
            pt2_x, pt2_y, pt2_z)) then

            find_angle_btwn_lines(pt2_x, pt2_y, pt2_z, origpt_x, origpt_y, origpt_z,
            nextpt_x, nextpt_y, nextpt_z, a_normal, b_normal, c_normal, &orig_angle);
            sum_polygons_angles = sum_polygons_angles + orig_angle;

        /* If this is an "interior" polygon, add it to the geometric database */

            if ((number of edges - 2) * pi == sum_polygons_angle) then
                push edge onto edge_stack;
                add_polygon(edge_stack, a_normal,b_normal, c_normal, d_normal);
            endthen
        /* If this is an "exterior" polygon, take this plane definition off the
         surface_normal_stack, so that the "interior" polygon can be found when the
         line segment's orientation is flipped. */

            else
                pop surface_normal_elt off surface_normal_stack;
            endif.


        /* Remove all the connecting line segments from the edgestack
         so the searching can start again */

            while MORE THAN 1 member on the edge_stack
                pop edge off edge_stack;
            endwhile;
        endthen;            /* Closed polygon. */

        /* Found new connecting line segment of polygon. */

        else
```

Figure 4-5 (Cont'd.).

```
            /* If the connecting_line segment's next point loops back onto a previous
            end point in the polygon,  we want to back track in our search. */

                for each edge in the edge_stack
                    if (points_are_equal (pt2_x, pt2_y, pt2_z,
                        edge.next_x, edge.next_y, edge.next_z)) then
                          start_angle = angle_btwn_lines;
                        continuewhile;
                   endif.

            /* Found a valid connecting line segment.  Add it to the edge_stack, and reset values
             for search next connecting line segment.*/

                push connecting_line onto edge_stack;
                pt1_x = pt2_x;
                pt1_y = pt2_y;
                pt1_z = pt2_z;
                pt2_x = next_x;
                pt2_y = next_y;
                pt2_z = next_z;
                sum_polygon_angles = sum_polygon_angles + angle_btwn_lines;
                start_angle = 0.0;
            endthen         /* Found connecting line segment */

            /* Didn't find a  connecting line segment, so reset values so we can backtrack in the
            search path. */

            else
                edge = peep on top of edge_stack;
                start_angle = edge.min_angle;
                if (NOT EMPTYSTACK edge_stack) then
                    pt2_x = edge.nextpt_x;
                    pt2_y = edge.nextpt_y;
                    pt2_z = edge.nextpt_z;
                endif.
                if (no_edges NOT EQUAL 1) then
                    edge = peep at second member from the top of  the edge_stack.
                   pt1_x = edge.nextpt_x;
                   pt1_y = edge.nextpt_y;
                   pt1_z = edge.nextpt_z;
                endthen
                else
                   pt1_x = origpt_x;
                   pt1_y = origpt_y;
                   pt1_z = origpt_z;
                endif.
            endif.     /* did not find a connecting line segment. */
        endwhile.     /*  while still looking  for connecting line segments */
    endwhile.           /*  while searching line segments on different plane definitions */
endfor.                 /* for both line segment orientations */
```

Figure 4-5 (Cont'd.).

```
                    /* Empty stacks when routine is done. */

    while NOT EMPTYSTACK surface_normal_stack
        pop surface_normal off surface_normal_stack;
    endwhile.

    while NOT EMPTYSTACK edgestack
        pop edge off edge_stack;
    endwhile.
endprocedure.
```

Figure 4 - 5 (cont'd.).

Implemented Solid Objects Data Structure

Once a nonduplicate polygon has been found by the depth search algorithm, it is sent to the add polygon module. Before that procedure is looked at in detail, let's consider the object's data structure. Each surface found in the user's drawings has a set of unique information. This information is maintained in the polygon header record in the "poly.dat" file. It includes search extents, polygon plane normals, display values and housekeeping information. (See Fig. 4 - 6). Additional information is saved to give the user the capability to display light renderings and hidden line drawings.

orig_a_normal	-	polygon plane normals
orig_b_normal		
orig_c_normal		
orig_d	-	plane d value
x_min	-	polygon extents
y_min		
z_min		
x_max		
y_max		
z_max		
nolines	-	number of lines in this polygon
edgefacet_offset	-	pointer to the edge-facet pairs of this polygon in "ef.dat".
solid	-	TRUE if polygon solid FALSE if polygon hollow
purge	-	TRUE if polygon has been deleted FALSE otherwise
lut0_orig_color	-	look up table 0 color for first found side
lut0_orig_shade	-	look up table 0 rendering shade for first found side
lut1_orig_color	-	look up table 1 color for first found side
lut0_opp_color	-	look up table 0 color for other side
lut0_opp_shade	-	look up table 0 rendering shade for other side
lut1_opp_color	-	look up table 1 color for other side

Fig. 4 - 6 Part of Polygon header data representation.

The plane definition normal is determined from the first three unique points of the polygon. The vector is normalized to make it easier to compare with other normals. The equations used to derive these values are listed in the section above. The volume extents are used in search and display modules to discard polygons that are not candidates for the given task.

The various shades and colors are used in display modules discussed below. Values for two different color look up tables are maintained. Look-up table 1 (LUT1) is the principal display setting for Design 4D. It has good resolution and an easily seen cursor, but it is limited to 16 colors. Look-up table 0 (LUT0) is used to display light shading drawings. It has a cursor that is harder to see, and displays a full color range of 256 colors. LUT0 was not the default display color table because the cursor was not clearly visible on all of the possible backgrounds. Each color field called "orig" is the color of the first side of the polygon that was found. In other words, this is the side seen by looking down the normal towards the edge ring. All the "opp" information fields refer to opposite side of the polygon. The "shade" fields are used to save one set of light rendered shadings. This allows the user to save one display to show clients and colleagues.

The polygon header record also contains some housekeeping information. When set to true, the purge flag indicates that a polygon has been deleted. The nolines field records the number of line segments in the polygon, and is used in modules that iterate through the edge ring. The edge pointer is the location, actually offset address, of the first polygon edge in the "ef.dat" file. This is also used when accessing the polygon's edges. When set to true, the solid flag indicates that a polygon is a solid, otherwise it is hollow. Traditionally this is indicated by the orientation of a surfaces' edges. Because of the inefficiency of the multiple disk accesses needed to determine this, a flag was used instead. In addition to polygon header information, each edge and facet ring must be represented. Figure 4 - 7 lists the specific data structure fields used.

origin[2][3]	-	beg and end x,y, and z coordinates of edge
onext[4]	-	[0] address of previous edge in this polygon (i.e., going counter-clockwise direction looking down polygon normal)
		[1] address of next polygon that shares this edge (i.e., clockwise direction around this edge looking down from origin[1] to origin[0])
		[2] address of next edge in this polygon (i.e., going clockwise direction looking down polygon normal)
		[3] address of next polygon that shares this edge (i.e., counter-clockwise direction around this edge looking down from origin[1] to origin[0]) */
index[2]	-	[0] if onext[1] references the next_polygon.onext[1] at the next polygon this will be 1, else references 3 and this will be 3

Figure 4-7 Edge - facet data representation

		[1] if onext[3] references the prev_polygon.onext[3] at the previous polygon this will be 3, else references 1 and this will be 1
edge_poly	-	address of polygon header record in "poly.dat"
prev_angle	-	angle between this edge facet and the previous edge facet

Figure 4 - 7 (Cont'd.).

The origin field contains the vertex information for each edge. To traverse an edge in the order of the clockwise edge ring, one goes down the edge from origin[0] to origin[1]. To determine the edge orientation in the counterclockwise edge ring, one simply traverses the edge from the origin[1] to origin[0]. This counterclockwise ring is referred to as the edge.symmetrical ring. Each polygon edge pair contains pointers, here offset addresses, into the previous edge (onext[0]) and the next edge (onext[2]) of the clockwise directed edge ring. Note that the previous edge of the clockwise edge ring is the next edge of the counterclockwise ring. Similarly the next edge of the clockwise edge ring is the previous edge of the counterclockwise ring.

This data structure also represents the facet ring. The pointers to the next polygon (onext[1]) and the previous polygon (onext[3]) allow traversal of the facet ring around a given edge. The next polygon in the ring is the closest polygon in the clockwise direction when looking from the directed edge's target (origin[1]) towards its source (origin[0]). The previous polygon is the closest polygon in the counterclockwise direction from the same viewing point. As with the edge ring, the previous polygon of the clockwise oriented facet ring is the next polygon in the counterclockwise orientation of the same facet ring. Again, the next polygon of the clockwise-oriented facet ring is the previous polygon in the counterclockwise ring. In Dobkin-Laszlo's terminology, the clockwise facet ring is referred to as an edge.rotate ring. The counterclockwise facet ring is referred to as the edge.rotate.symmetrical ring. The edge that is the center of a facet ring may have different directions in different polygons. The index field keeps the facet rings correctly intertwined as will be shown next.

It is difficult to visualize the polygon ring embedded in this data structure. Let's approach it step by step. In the simplest case, there is only one polygon. Here the polygon edges point to the original polygon forming a closed loop. Note the settings of the index fields in Figure 4 - 8.

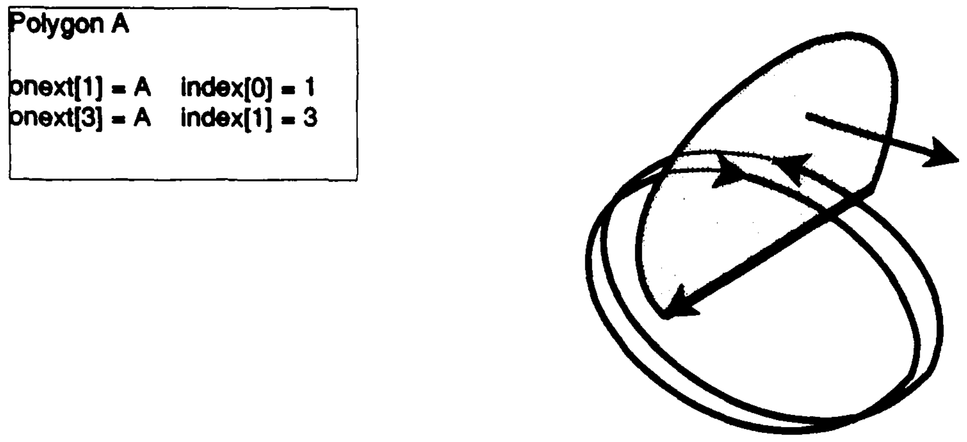

Figure 4-8 Polygon that is the only member in the facet ring.

As discussed above, the polygon-edge pair can be represented by the polygon's normal vector and a directed edge. By evaluating the polygon normal and a directed edge, it is possible to locate the facet ring that the polygon-edge pair belongs in. First an undirected edge that matches the given edge must be found in ef.dat. If no matching edge is found, this atom does not belong in any existing polygon ring. This is equivalent to the polygon shown in Figure 4 - 8. Once an edge is found, the new atom must be linked into the polygon ring. If the two matching edges are oriented in the same direction, then the two polygons are oriented in the same direction around the edge as in Figure 4 - 9. Otherwise the two polygons are oriented as shown in Figure 4 - 10. When linking the edge-facet data structures together, the different orientations determine the index fields.

When the second polygon is added to the facet ring, it must be tied into the loop. If the orientation of the second polygon matches that of the first polygon, the facet ring conceptually forms a pair of circular linked list. When the two are in opposite directions, the location of the next polygon ring and the symmetric ring are the same, but the indexes that indicate the next link to use are different. The index value represents the *onext* field (see Fig. 4 - 7) in the next polygon which, when followed, continues to circle the facet ring in the current orientation.

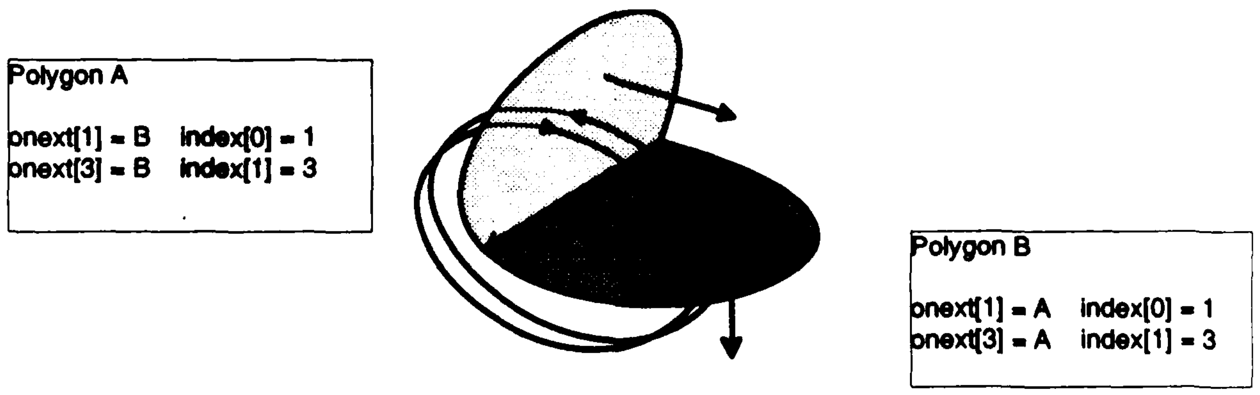
Polygon A
onext[1] = B index[0] = 1
onext[3] = B index[1] = 3
Polygon B
onext[1] = A index[0] = 1
onext[3] = A index[1] = 3

Figure 4-9 Two polygons oriented in the same direction in a facet ring.

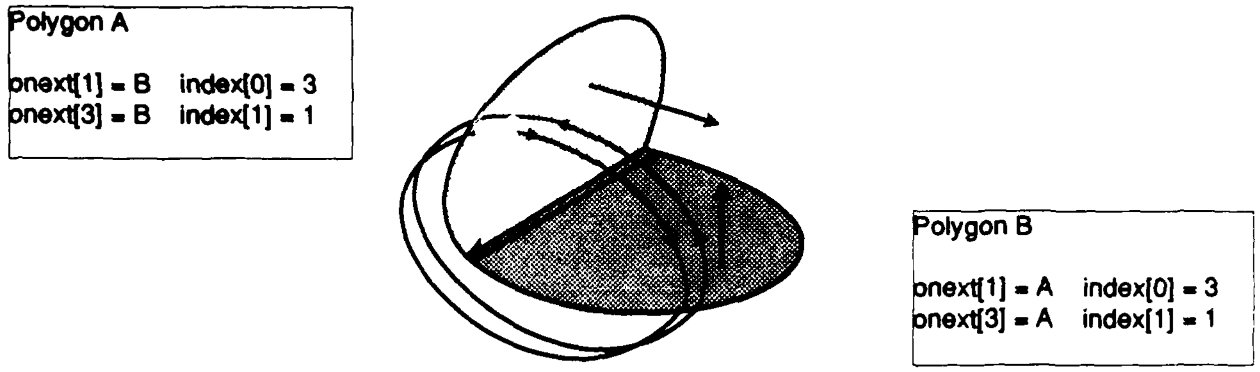
Polygon A
pnext[1] = B index[0] = 3
pnext[3] = B index[1] = 1
Polygon B
pnext[1] = A index[0] = 3
pnext[3] = A index[1] = 1

Figure 4-10 Two polygons oriented in the opposite direction in a facet ring.

As more polygons are added into the polygon ring, the same basic rules are applied. Because Design 4D stores the object database on secondary memory, the second and later polygon insertion must be treated as separate cases. This is to assure that new polygon ring links are not overwritten by multiple low level disk accesses when there are only two polygons in the resulting facet ring. Notice how the indexes change as the facet ring orientation flips between individual polygons. Because different facets incorporate edges in different directions, the index field is needed to thread the rings together.

5 DESIGN DECISION OUTPUT TOOLS DEVELOPED

Depth Sort Algorithm

Once the BREP model is created, as discussed above, Design 4D provides color fill as a real time aid to the designer. This eliminates the need to have a separate painting or rendering program. To facilitate the user's design development process, Design 4D offers surface fills, hidden line removal, and ambient line-rendering capabilities. If the architect wants to save a particularly aesthetic rendering, they can select the "save rendering" option. These tools help designers visualize the model during design development and eliminate the delay normally associated with using a separate rendering program, or building a scale model.

Each rendering, whether 2D or 3D, is displayed using the depth sort algorithm. This algorithm is also called the painter's algorithm because it builds up the display similar to the way an artist puts oil paint on a canvas. The back surfaces are painted in first, with the forward surfaces being painted in last. When implementing this approach in Design 4D, a few modifications were made to the algorithm. Since Design 4D allows the user to rotate the model to obtain any possible view, the algorithm was modified to transform the objects into the traditional right hand, z coming out of the screen coordinate system. In addition, a transformation matrix was developed that would allow the algorithm to be used for 2D renderings as well as 3D renderings.

The painter's algorithm creates a realistic rendering by creating a display list of surfaces, ranging from those farthest away from the front of the screen to those nearest to the screen. The ordering of this list is important because it reflects the order in which the surfaces will be "painted" onto the screen. In a right handed coordinate system, the computer monitor represents 3D space with the positive z axis coming out of the screen. The x coordinates run horizontally across the screen, and the y coordinates run vertically across the monitor screen. The first step in the painter's algorithm is to create a display list that orders all the surfaces by the smallest z value of each surface. Further examination of the display list is needed to place surfaces in the correct order. Beginning at the start of list with the farthest surface, each surface is compared with any surface that overlaps it in the z direction. The tests listed below are performed to determine if the surfaces should be swapped. As soon as one test proves true, the evaluation is finished and the two surfaces are known to be ordered correctly relative to each other. The tests are, from easiest to hardest:

1. Compare the x and y extents of the two surfaces. If they do not overlap, the polygons' relative order is correct.

2. If the current polygon is completely behind the overlapping polygon in relation to the view plane, the polygons are in the right order.

3. If the current polygon is completely in front of the overlapping polygon in relation to the view plane, the polygons are in the right order. (This test is necessary because it is possible for the current polygon to not be completely behind the overlapping polygon when the overlapping polygon is completely in front of the current polygon.)

4. If the projections of the two polygons onto the view plane don't overlap, the polygons' relative order is okay.

If all these tests fail, the two polygons' order in the display list should be switched. If two polygons fail all of these tests a second time, they may intersect each other, or they may be part of a cycle of surfaces that alternately obscure each other. In both of these cases, the only solution to this problem is to intersect the polygons involved so the final drawing can be rendered correctly. Polygon segmentation is a very computer-intensive process and was not implemented in Design 4D. As each surface is compared with all the other surfaces that overlap it, the surfaces are swapped with each other as need be. When all the surfaces have been evaluated relative to each other, the display list is properly ordered and ready to be rendered.

With Design 4D's flexible display interface, some modifications needed to be made to the painter's algorithm. Both by rotating the 3D view, or looking at a 2D view, it is possible to have a display where the coordinate axis coming out of the screen is not the positive z axis. In these cases, the display list built by the painter's algorithm will not result in the desired rendering. To facilitate the correct display list creation, Design 4D transforms the surfaces using a matrix that changes object coordinates to that of a screen coordinate system with the positive z coordinate coming out of the screen. The 2D views were treated in a similar manner as the 3D views. A matrix was created that transformed the surface coordinates into the appropriate screen coordinate system. The 2D matrices did not apply the perspective transformations that the 3D matrices did. Once the surfaces have been transformed into the appropriate coordinate system, the painter's algorithm described above can be applied.

Types of Renderings Supported

When all the surfaces have been evaluated relative to each other, the list is ready to be rendered. The display list is then emptied with each polygon "painted" as it is removed from the list. The way the surface is painted determines the resulting type of rendering. Design 4D allows the user to render simple surface fill drawings, hidden line drawings, and ambient light drawings. Using the polygon header information, the color of each surface can be determined. When a surface fill rendering is being displayed, each polygon is filled with its color. To produce a hidden line drawing, each polygon is filled with the background color and outlined in its own color.

Design 4D offers a light rendering option to assist in design analysis. In the final display, the model is rendered as if there is a single light source (the sun), combined with ambient light from surrounding surfaces. The user defines the single light source's angle and direction by drawing a line segment. The beginning of the line segment represents the sun and the end of the line segment represents the destination of the sun's rays. This light vector represents the direction and angle of the sun's rays over all of the modeled objects. In addition, the user can set the level of ambient light reflecting on each surface from surrounding surfaces. The ambient light level also lightens or darkens each surface's color. Once the cumulative effect of the light source and the ambient light has been determined, the surface is rendered with a darker or lighter shade of its original color. This produces an effective simulation of a 3D model under user-defined lighting conditions.

Once Design 4D has "built" the BREP solid model, renderings can be created very quickly. The surface data structure maintains all the information the painter's algorithm needs. The various rendering types give the architect many ways to quickly visualize the building model. During system testing, these renderings proved to be some of the users' favorite options in Design 4D. By replacing the architect's traditional cardboard models with Design 4D's rendering capabilities, the designer can consider many more massing configurations.

User Interface Issues

After a model has been developed and rendered, the designer can evaluate building massing and material selection. Different colors can be used to represent different materials. As the design evolves, the architect will change material selections and subsequently need to change surface colors. To allow the user to manipulate an individual polygon, a surface selection technique was developed.

To select an individual surface, the user first selects a line segment in the desired polygon. Once an edge has been selected, each surface this edge is a part of is highlighted using a technique similar to a "needle" diagram (see Winston, p. 340). A needle diagram draws each surface with the plane normal sticking out the center of the surface. Each surface has two sides that the user may want to color. To distinguish the two, the surface normal is drawn from the center of surface extents. The user can determine which side they are selecting by placing themselves at the far end of the normal and looking towards the center of the polygon. Every possible surface is highlighted until the user selects one, or there are none left to choose. With this easy interface, a user can select an individual polygon to manipulate.

6 CONCLUSIONS AND RECOMMENDATIONS

Design 4D's graphic interface is one that the casual user finds easy to learn and use. Realizing that the typical designer would not be using Design 4D continuously (and therefore maintain a sophisticated working knowledge of the system), a transparent interface was a primary concern during the development of Design 4D. By developing a post-sketch modeling system, Design 4D maintains a consistently easy user interface. On this level, the author's implemented modeler meets the users' needs.

Most of the author's research effort went into determining the allowable surfaces and storing them efficiently. Further development of the algorithms that Dobkin and Laszlo delineated to modify facets would add new capabilities to Design 4D's modeling system. Given the data structure that the paper outlines, their algorithms could meld and decompose cell complexes. Also, an algorithm is included to perform Delauney's Triangulation on the cell complexes. Delauney's Triangulation breaks a polygon into the triangles that compose it. This allows for more accurate surface renderings because fewer intersecting polygons remain. If this algorithm had been implemented, Design 4D's surface renderings would have been more accurate. This would have increased required disk storage for the extra surfaces and would have increased the time required to render the surfaces.

Design 4D's use of secondary storage for geometric objects allows a more sophisticated user interface, but slows down execution. Any method that would reduce noncontiguous disk accesses would produce greater efficiency. By implementing a search tree for the polygon headers, the model building process could be sped up. Whatever search method is used, it must substantially reduce the search path to make up for the slow access time of secondary memory.

While Design 4D's modeling capabilities give the designer effective visual analysis tools, improvements could also be made. With the use of hatching and texture mapping, materials could be represented more realistically. The rendering algorithm already used could be modified to map the textures and hatch patterns onto the surfaces. The most difficult aspect of this process would be assuring that the textures were "perspectively" correct and at the appropriate scale. These expanded capabilities would give designers a preview of their work that was not possible before.

Design 4D continues to be developed at USACERL. Currently, architects at nine test sites are using the system during their design process. The models and renderings made possible by this research are among the designers' favorite tools. By allowing architects to better visualize their design alternatives, Design 4D is a system well suited for the conceptual design phase.

REFERENCES

Bond-Harris, Laura S., Beth A. Symonds, and Kenneth H. Crawford, "Desktop Designing—Can You Get There From Here?", *Computer Integrated Construction* (January 1989).

Bond-Harris, Laura S., et al., *A Case Study of a Prototype Computer-Aided Architectural Design System*, Technical Report P-88/17 (U.S. Army Construction Engineering Research Laboratory [USACERL], January 1989).

Crawford, Kenneth H., and Laura S. Bond-Harris, "Three Dimensional Architectural Modeling on a Microprocessor," *CEPA Spring 1987 Conference Proceedings* (June 1987), pp. 7-19.

Crawford, Kenneth H., Beth A. Symonds, and Laura S. Bond-Harris, *Design 4D User's Manual*, Draft ADP Report (USACERL, January 1989).

Dobkin, David P., and Michael J. Lazlo, "Primitives for the Manipulation of Three Dimensional Subdivisions," *ACM Transactions on Computational Geometry*, (Waterloo, Ontario, June 1987), pp. 86-99.

Foley, J.D., and A. Van Dam, *The Fundamentals of Interactive Computer Graphics* (Addison-Wesley, Reading, MA, 1984).

Guibas, Leonidas, and Jorge Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams," *ACM Transactions on Graphics*, vol. 4, No. 2 (April 1985), pp. 75-123.

Hearn, Dwight D., and M. Polly Baker, *Computer Graphics* (Prentice-Hall, Englewood Cliffs, NJ, 1986).

Winston, Patrick H., *Artificial Intelligence*, 2d ed. (Addison-Wesley, Reading, MA, 1984).

DISTRIBUTION

Chief of Engineers
ATTN: CEHEC-IM-LH (2)
ATTN: CEHEC-IM-LP (2)
ATTN: CERD-L

Fort Belvoir, VA 22060
ATTN: CECC-R

Defense Technical Info. Center 22304
ATTN: DTIC-FAB (2)

8
9/90