A Walk Through the Planned CS Building

Delnaz Khorramabadi

Master's Project Report
Under the Direction of
Prof. Carlo H. Séquin

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

ABSTRACT

Using the architectural plan views of our future CS building as test objects, we have completed the first stage of a Building walkthrough system. The inputs to our system are AutoCAD files. An AutoCAD converter translates the geometrical information in these files into a format suitable for 3D rendering. Major model errors, such as incorrect polygon intersections and random face orientations, are detected and fixed automatically. Interactive viewing and editing tools are provided to view the results, to modify and clean the model and to change surface attributes.

Our display system provides a simple-to-use user interface for interactive exploration of buildings. Using only the mouse buttons, the user can move inside and outside the building and change floors. Several viewing and rendering options are provided, such as restricting the viewing frustum, avoiding wall collisions, and selecting different rendering algorithms. A plan view of the current floor, with the position of the eye point and viewing direction on it, is displayed at all times. The scene illumination can be manipulated, by interactively controlling intensity values for 5 light sources.

We have improved the rendering speed of our system by subdividing the model into cells and reducing the fraction of the building that is sent to the rendering pipeline at display time. Each floor is divided into approximately room-sized cells, each with a precomputed cell-to-cell visibility list. During display, the cell containing the viewpoint is identified, and the contents of the potentially visible cells are retrieved from storage. This set is further reduced by culling it against the observers view frustum. The contents of the remaining cells are then sent to the graphics pipeline.

September 17, 1991

Table of Contents

A Walk Through the Planned CS Building

Delnaz Khorramabadi

Master's Project Report
Under the Direction of
Prof. Carlo H. Séquin

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

1. Introduction

Computers and CAD tools have been incorporated in most professions, including architecture. Many architects are now using software such as AutoCAD and CADDS, for efficient generation of high quality architectural floor plans, in the drafting stage of the architectural process. However, using computers merely for drafting purposes is only a glimmer of the true potential that they can offer to architects. Today, graphics work-stations are capable of rendering scenes with tens of thousands of polygons in a single frame-time of 1/30 second. This offers a great potential for real-time simulation of movement through complex environments, particularly huge buildings. A computer-based interactive building walkthrough system, which simulates the human visual experience of buildings without physically constructing them, expedites the design, planning, evaluation, and presentation stages of the architectural process.

Traditionally, two-dimensional floor plans of buildings and elevations or perspective projections have been the basic communication media between architects and their clients. Particularly with respect to the interior of buildings, architects rely on the client's imagination to visualize a proposed building from its architectural plan views, assuming that clients are familiar with architectural symbols and rules, and have the training and experience to construct three-dimensional images from two-dimensional plan views. These methods might be suitable for small projects, but are clearly inappropriate for large projects that require consensus among several parties and a large number of design-evaluation-feedback cycles.

Experience with the planned CS building has made it clear that traditional architectural methods, particularly in the evaluation phase where designs are presented to clients, should and can be enhanced with interactive computer-based methods. It is time that two dimensional architectural plan views be replaced by advanced graphics work-stations, on which architects and their clients can explore a proposed building, furnish its rooms, change its surface characteristics and even try potential structural changes.

Using the design of Soda Hall, the planned new CS building, as a test object, we have finished the basic components of such a walkthrough system. In the first stage of the project, our research efforts have been focused on four issues:

1- Conversion of architectural plans into a consistent 3D model (Berkeley UNIGRAFIX).

2- Automatic error-detection and correction in the architectural description.

3- A display system with a friendly interface for novice users to "walk" through buildings.

4- Real-time rendering speed for interactive exploration.

The first concern of a walkthrough exploration of a planned building is to obtain a correct model. The modeling stage of buildings consumes vast amounts of human resources. A walkthrough system should ultimately include a modeling tool, with a library of building components that have been designed with anticipation of the needs of the display system. However, we prefer to focus our efforts for now on other aspects of the system and obtain our models from architects. Computer generated architectural databases are now available which contain geometrical descriptions of buildings. However, they typically are not suitable for direct use by computer-based rendering programs. Depending on the software used to prepare them, these databases have various formats and use a different collection of primitives from those needed for efficient rendering. For our purposes, some of these primitives need to be processed to obtain a suitable polyhedral representation of the underlying objects. We have implemented a converter that extracts the geometrical and surface attribute information embedded in AutoCAD DXF files [Aut 90] [Aut 91] and translates them into the Berkeley UNIGRAFIX format [Seq 83] [Seq 91], suitable for 3D object modeling and rendering.

Because architectural models are created with concern for architectural needs rather than those of a three-dimensional display system, they may contain geometrical and topological inconsistences. These inconsistencies include: building elements that are not modeled as closed surfaces; faces of random orientations; arbitrary polygon intersections; non-planar polygons; or several polygons that lie in the same plane and overlap. Advanced shading techniques require flat faces with consistent orientations where inside and outside are clearly distinguished. Our system detects and fixes many of these problems by further processing of the converted database.

Once a suitable building model is created, the user must be provided with a simple-to-use interface for interactive exploration. To make the simulation close to the actual human experience of walking through a building, we have considered issues such as the following: How much freedom of motion should be allowed inside and outside buildings? How can we help the user to keep global orientation of where he/she is in the building? Should users be allowed to pass through walls, ceilings and floors or should they be constrained to use only portals, such as doorways and stairways? Which rendering features should the user be allowed to control interactively? We started with the user-interface designed in the walkthrough project at the University of North Carolina [Air 90a][Air 90b], and with these questions in mind, complemented it with several modifications and enhancements.

We have also investigated preprocessing methods to enhance rendering speed with the goal of obtaining real-time update rates of 15-20 frames per second. Our models are huge, consisting of millions of polygons when fully furnished. To achieve reasonable interactivity, the fraction of the building model that must be processed at display time by the hidden surface removal algorithm or hardware must be reduced significantly. In other words, a major part of the visibility problem must be solved before display time. The visibility problem, in short, states that: given a set of opaque polygons, compute the portion of the scene visible to any view point. Exact solutions to this problem are expensive in time and space. Methods that compute correct superset solutions and remain efficient are under investigation [Tel 91]. The results of these efforts have been integrated with the system.

In this paper, my research efforts in the above areas will be discussed in detail. My work has focused on: writing a converter for AutoCAD, identifying database problems and designing

algorithms and interactive tools to detect and fix them, and on implementing a display system for our walkthrough system. I have also integrated the results of the work done on the visibility problem [Tel 91] with the rest of the system. There are several other issues that our walkthrough system has to deal with. Other members of our team are investigating issues such as improving realism of lighting simulations by using radiosity algorithms [Smi 91], storage handling techniques using high-bandwidth networks and massive disk-based storage systems [Fun 91], and generating models from existing buildings [Sre 91].

The paper is organized as follows. The remainder of this Section provides an overview of the system and a brief description of previous work done in this area. Section 2 introduces AutoCAD's DXF file format. An overview of the conversion of AutoCAD primitives into UNIGRAFIX format is presented in Section 3 and the details are discussed in Appendix A. Section 4 clarifies major database problems and discusses various solutions to them. In Section 5, a brief introduction to our visibility precomputation is provided. Section 6 describes our display system and Section 7 is a tutorial on the processing pipeline.

1.1. An Overview of the System

Figure 1.1 shows the basic structure and components of our walkthrough system. Our system is divided into three phases : a conversion phase, a preprocessing phase, and a walkthrough phase. In the first phase, AutoCAD DXF files are read and translated to UNIGRAFIX format. The result is then processed for error detection and correction. Most problems are fixed automatically and some are corrected manually using interactive editing tools. The conversion process ultimately results in a collection of colored planar polygons with topologically correct shared edges and vertices and consistent face orientations.

In the second phase, the building description along with a description of the interior lights are passed to the radiosity module. The radiosity program simulates effects such as shadows and color bleeding that are caused by multiple bounces of emitted light. Polygons are finely subdivided and a new color value for each vertex is computed. This new description contains strictly view-independent information which will be used at display time [1].

Another preprocessing step is the visibility precomputation, where *major* structural elements of the building are identified and used for the subdivision of the model into cells. For each cell, a list of cells that are visible to an observer moving freely in the cell is computed and associated with the cell. The building polygons (eventually radiosity-shaded) are then attached to the cells that they occupy. All this information will be used in the display phase.

In the walkthrough phase, the display program loads the precomputed visibility information along with model polygons, and moves the viewer through the building. Device inputs are interpreted to obtain viewer position and orientation as well as various display parameters.

---

[1] Radiosity color values are not yet incorporated in the system.

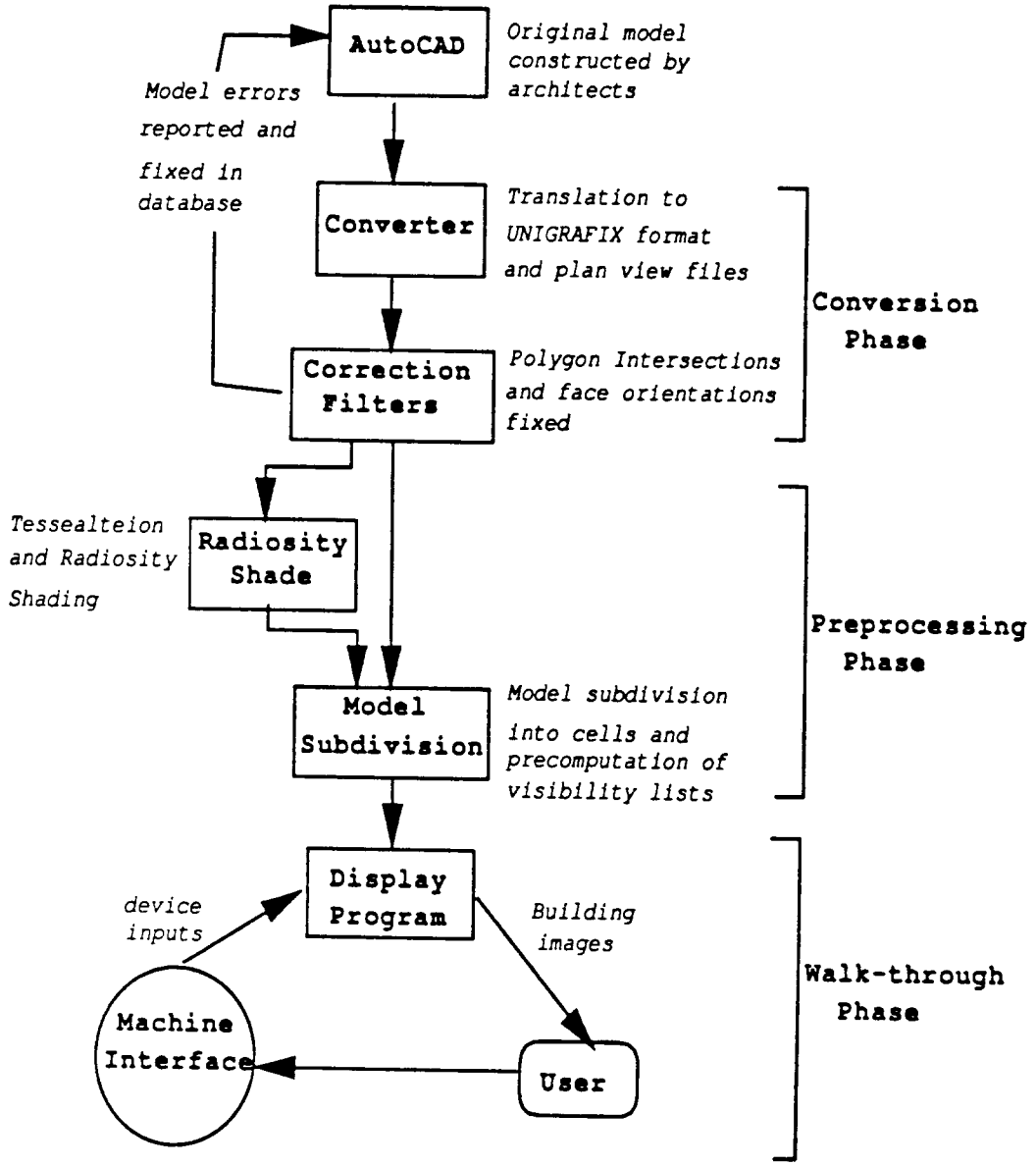


Figure 1.1-Basic components of our walkthrough system

1.2. Previous Work

The UNC walkthrough project, developed by John M. Airey and other graduate students at the University of North Carolina, is the most relevant previous work to us [Air 90a][Air 90b]. The UNC walkthrough system concentrates on techniques to enhance visual simulation. Two strategies are used to combine interactivity with high image quality; precomputation before display and adaptive refinement. Precomputation is used for partially solving the hidden surface

removal problem and for shading the model. Adaptive refinement is used to improve image quality dynamically when the user is stationary.

In this system, the model is automatically subdivided into sets of viewpoints called cells. For each cell an approximation of the potentially visible subset (PVS) of the model is calculated. This preprocessing stage substantially cuts down on the number of polygons to be rendered from a view point. With this scheme, the system has been able to achieve a speedup of 3.25 in the worst case and 30 in the average case for a model with 7125 polygons, divided into 269 cells with a largest PVS of 2196 polygons and an average PVS of 230 polygons.

Recursive binary partitioning planes are used to divide the model into cells. The orientation of these planes are restricted to be normal to one of the coordinate axis and limited to planes that contain model polygons. A heuristic function evaluates each axial plane containing a model polygon for its suitability as a splitting plane. Criteria considered for this selection are: (1) how well the plane separates the model; (2) how well it hides two sides of the subdivided cell from each other; (3) how little it splits individual polygons of the model. This subdivision process results in a search tree with interior nodes representing binary separating planes and leaf nodes representing cell volumes.

After model-space subdivision, the subset of the model potentially visible to an observer inside each cell is computed and stored with the cell. For this, the portals, corresponding to the hallways, doorways, stairwells, etc. in a building, are found in each cell. Explicit polygonal descriptions of portals are obtained by computing the boolean difference of the cell boundary and polygons lying in the cell boundary planes. The union of what is visible through all the portals of a cell and the cell itself yields the solution to the PVS problem.

Two classes of algorithms are presented in Airey's thesis to solve the portal to polygon visibility problem. One class uses point sampling and may underestimate the set of polygons in computing a given cell's potentially visible set. The other one establishes occlusion relationships among polygons and is based on the computation of shadow volumes. This method might overestimate the solution and is computationally expensive. The sampling-based method is implemented.

The radiosity scheme in this system uses a ray casting approach based on a jittered hemispherical distribution. The ray-polygon intersection algorithm is accelerated by taking into account the major characteristics of architectural databases such as the predominancy of axial polygons. The radiosity program computes and records the contribution of 20 sets of lights individually for each patch. A linear combination of these values is used for the final scene. The intensity settings for each of the light sets may be altered by the user during display. This allows the scene illumination to be manipulated interactively.

The radiosity algorithm is also used to fix face orientations by keeping track of the radiosity value for both sides of every polygon. Display polygons are generated for both sides of the polygon if both sides receive a certain amount of energy. Alternatively, the side that receives the most energy is used as the front side of the polygon.

The idea of adaptive refinement is used to deal with image features that cannot be handled by precomputation. The subdivision of polygons into patches by the radiosity program is used to produce finer levels of detail. Each polygon has an associated list of polygons used to refine it. Two adaptive refinement steps are currently being implemented.

All these components are integrated into a user-interface that coordinates input devices with the display system. This interface allows unrestricted motion inside and outside buildings and

provides interactive control on rendering features such as scene illumination and radiosity refinement levels.

The UNC project and our walkthrough system share some components and many goals. We were introduced to this project in the early stages of our work. Instead of starting a whole new system from scratch, we decided to examine this system first, learn from it, discover potential shortcomings and build upon it. We adopted some components of the UNC walkthrough system and concentrated on enhancing them. Particularly, we based our user-interface on the UNC user-interface and focused our efforts on modifications and additions that would facilitate the navigation and exploration of buildings. In other areas, although we share most basic goals (model subdivision, radiosity lighting model, etc.), we discovered the potential for more efficient solutions and implementations.

2. The AutoCAD Building Model

Architectural models are prepared using various drafting tools, such as CADDS, Architrion or AutoCAD. Our model of the CS building was originally created by architects using AutoCAD. Therefore, we concentrate on AutoCAD, noting that many issues and problems encountered in our model also arise in other models and software tools.

AutoCAD is not a true three-dimensional modeling tool. AutoCAD databases consist of various two-dimensional and three-dimensional entities that often require processing to obtain a consistent three-dimensional polyhedral representation. A significant portion of these files deal with AutoCAD's internal variables and other information that are not relevant to us. AutoCAD files are not suitable for direct use in a sophisticated three-dimensional display system with advanced rendering capabilities such as radiosity-based shading.

We have written an AutoCAD converter that translates the geometrical and surface attribute information in AutoCAD files into a simple format, suitable for three-dimensional object manipulation and rendering. We picked the Berkeley UNIGRAFIX format [Seq 91] because of its simplicity and compatibility with other software available in our group. With our basic converter, we are also able to detect model problems and deficiencies. This information was used to devise guidelines for our modelers to produce complete and correct AutoCAD models, that can be converted easily to UNIGRAFIX and used for interactive walkthroughs.

Incompatible features and ambiguous entities in AutoCAD make the conversion of such files to UNIGRAFIX format a tedious task. There are many drawing entities in AutoCAD that have no direct correspondents in UNIGRAFIX. For example, it is not clear whether the closed POLYLINE in AutoCAD should be interpreted as a closed wire frame or an opaque face. Difficulties also arise from the fact that there are many different ways in AutoCAD to draw the same object. A simple face in R3 can be drawn using the closed POLYLINE, the extruded LINE, the 3DFACE or the SOLID primitives. Finally, most documentation on AutoCAD databases do not provide enough information on details and are too general [Aut 90] [Aut 91].

In this Section AutoCADs DXF file format and other relevant features will be introduced, whereas the conversion to UNIGRAFIX is discussed in Section 3 and Appendix A.

2.1. AutoCAD DXF File Formats

There are two file formats defined in AutoCAD to assist in interchanging drawings between AutoCAD and other programs: the "Drawing Interchange" file format (DXF) and the Initial Graphics Exchange Standard (IGES) file format. We chose to work with the DXF format because we found it to be more readable and better documented.

DXF files are standard ASCII text files (.dxf extension), produced by the AutoCAD command "DXFOUT". They are converted back into AutoCADs internal format (.dwg extension) by the "DXFIN" command. The overall organization of DXF files is well described in the AutoCAD Reference Manual and briefly in any other AutoCAD handbook. However, there are many undocumented details and variations that must be discovered through close experience with various databases.

2.1.1. DXF Group Codes

Everything in a DXF file is (wastefully) stated as a group. A group (wastefully) consists of two lines. The first line is the group code and the second line is the group value. Group codes are used both to indicate the type of the value of the group (string, integer, float ...) and the general use of the group. Tables 2.1 and 2.2 (extracted from the AutoCAD Reference Manual) describe the type of the value each range of group codes specify, and the use of each group, respectively. Some of the groups in these tables are not useful for our purposes and some have never been used in our database; however, they are included for completeness.

2.1.2. DXF File Sections

A DXF file is divided into four sections:

a-HEADER Section:

This section contains all AutoCAD drawing variables that were current when the DXF file was created. These variables describe the AutoCAD drawing environment. None of the drawing variables in this section are useful for our purposes.

b-TABLE Section:

Definitions for LINETYPEs, LAYERs, TEXT styles and VIEWs used in the drawing are stored here. The only piece of information relevant to us in this section are the items in the LAYER table. The name of all layers used in the drawing along with some AutoCAD layer attributes are stored in this table.

c-BLOCKS Section:

The definition of all blocks used in the drawing resides here. Each block consists of several drawing entities that appear between BLK and ENDBLK codes. Block definitions are never nested; however, one can find a reference to a previously defined block in another block.

Blocks in AutoCAD correspond to definitions in UNIGRAFIX, and provide for hierarchy in the drawing. Blocks, similar to definitions in UNIGRAFIX, can be scaled, mirrored, translated, rotated, and inserted in a drawing.

d-ENTITIES Section:

This section is the main part of a DXF file. Every DXF file should have an ENTITIES section. It consists of drawing entities (LINEs, 3DFACEs,...) and BLOCK references. The ENTITIES Section calls all entities that are outside BLOCK definitions. The ENTITIES section in a DXF file corresponds to the top level of a UNIGRAFIX file that contains all statements except definitions.

Figure 2.1 shows a concise version of a DXF file and its corresponding UNIGRAFIX description. The DXF file describes a block called Window constructed in layer EXT_WALL which contains a three dimensional triangular face. The Window block is referenced in the ENTITIES section using the INSERT primitive.

Group code range	Group type
0-9	String
10-59	Floating-point
60-79	Integer
210-239	Floating-point
999	Comment(string)

Table 2.1

Group code	Group Value
0	Identifies the start of an entity, table entry, or file separator. The text value that follows indicates which.
1	The primary text value for an entity.
2	A name; Attribute tag, Block name, etc.
3-4	Other textual or name values.
5	Entity handle expressed as a hexadecimal string.
6	Line type name (fixed).
7	Text style name (fixed).
8	Layer name (fixed).
9	Variable name identifiers (used only in HEADER section).
10	Primary X coordinate (start point of a Line or Text entity, center of Circle, etc.).
11-18	Other X coordinates.
20	Primary Y coordinate. 2n values correspond to 1n values and immediately follow them.
21-28	Other Y coordinates.
30	Primary Z coordinate. 3n values correspond to 1n and 2n values and immediately follow them.
31-38	Other Z coordinates.
38	This entities elevation if nonzero (fixed). Output only if system variable FLATLAND is set to 1.
39	This entities thickness if nonzero (fixed).
40-48	Floating point values (Text height, scale factors , etc.).
49	Repeated Value. Appears in tables.
50-58	Angles.
62	Color number (fixed).
66	"Entities follow" flag (fixed).
70-78	Integer values, such as repeat counts, flag bits, or modes.
210,220,230	X,Y,Z components of extrusion direction.
999	Comments

Table 2.2

AutoCAD DXF	Comments	Berkeley UNIGRAFIX
0		
SECTION	*{beginning of header section}*	
2		
HEADER		
0		
...	*{header variable items}*	
0		
ENDSEC	*{end of header section}*	
0		
SECTION	*{beginning of tables section}*	
2		
TABLES		
0		
TABLE	*{beginning of layer table}*	
2		
LAYER		
0		
LAYER	*{new layer}*	c_rgb EXT_WALL 0.4 0.3 1. ;
2		
EXTERNAL-WALL	*{layer name}*	
...	*{description for this layer}*	
0		
...	*{description for other layers}*	
0		
ENDTAB	*{end of layer table}*	
...	*{description for other tables}*	
0		
ENDSEC	*{end of table section}*	
0		
SECTION	*{beginning of BLOCKS section}*	
2		
BLOCKS		
0		
BLOCK	*{new block}*	def Window ;
2		
Window	*{block name}*	
8		
EXT_WALL	*{block layer}*	
0		
3DFACE	*{new drawing entity}*	
8		
EXT_WALL	*{layer name}*	
10		
0.	*{x coordinate of first corner}*	
20		
10.	*{y coordinate of first corner}*	
30		
10.	*{z coordinate of first corner}*	v v1 0. 10. 10. ;
11		
10.	*{x coordinate of second corner}*	
21		
10.	*{y coordinate of second corner}*	
31		
10.	*{z coordinate of second corner}*	v v2 10. 10. 10. ;
12		
0.	*{x coordinate of third corner}*	
22		

```
10.                 {y coordinate of third corner}
 32
10.                 {z coordinate of third corner}          v v3 10. 0. 10. ;
 0                  {end of entity}                         f f1 ( v1 v2 v3 ) EXT_WALL;
...                 {other entities for block}
 0
ENDBLK              {end of block}                          end ;
...                 {other block definitions}
 0
ENDSEC              {end of BLOCKS section}
 0
SECTION             {beginning ENTITIES section}
 2
ENTITIES
 0
INSERT              {block reference}
 2
WINDOW              {name of inserted block}
 8
EXT_WALL            {name of layer}
 38
100.                {elevation of insertion point}
 10
200.                {x coordinate of insertion point}
 20
0.                  {y coordinate of insertion point}
 41
-1.                 {x scale value}
 42
2.                  {y scale value}
 50
90.                 {rotation angle}
 0                  {end of insert}                         i (Window EXT_WALL -tx 200. -tz 100 -mx -sy 2. -rz 90.);
...                 {other drawing entities and block references}
 0
ENDSEC              {end of ENTITIES section}
 0
EOF
```

Figure 2.1- Comparison of a DXF file with corresponding UNIGRAFIX description.

2.1.3. Drawing Entities

Basically, there are two kinds of drawing entities in AutoCAD DXF format: planar and 3D. The 3D entities are 3DLINE, 3DPOINT, 3DFACE, and 3DPOLYLINE. The planar entities are LINE, CIRCLE, ARC, SOLID, INSERT, and POLYLINE †. Every entity in a DXF file starts with a 0 group identifying the entities type and is followed by an 8 group indicating the layer on which the entity resides and several other groups that are specific to each entity. The appearance of a new 0 group terminates the groups for the previous entity.

---

† Other AutoCAD primitives such as POINT, TEXT and TRACE also exist. These primitives are not mentioned here because they are either not useful for our purposes or not used in our model.

Three-dimensional objects in AutoCAD are constructed in two different ways. The direct method is to use one of the 3D primitives such as 3DFACE. Another method is to first draw a planar entity in the *xy* plane and then extrude it in the *z* direction. For example, a face can either be drawn by the 3DFACE command or by drawing a line with the LINE command and then extruding it.

2.2. AutoCAD Coordinate Systems

AutoCAD uses two coordinate systems: the World Coordinate System (WCS) and the Entity Coordinate System (ECS). The WCS is the Cartesian *x, y, z* system with its origin defined in the lower left-hand corner. This is the base coordinate system and everything is ultimately measured in this system.

The ECS, also referred to as the User Coordinate System (UCS) is used to define planes for constructing planar entities. Planar entities in AutoCAD are placed in the *xy* plane and then extruded in the *z* direction. However, one can rotate the coordinate system and place the planar entity anywhere in space. This is done in AutoCAD by using the User Coordinate System.

When group codes 210, 220, and 230 appear in an entity record, it means that the entity has been constructed on an Entity Coordinate plane and all coordinates are relative to this ECS. The coordinate values of groups 210, 220, and 230, define a vector along the ECS *z* axis relative to WCS. Since for a given *z* axis there are an infinite number of coordinate systems, defined by rotating the *xy* plane around the origin or translating the origin, the information contained in these groups are not enough to define the entity. The missing information comes from AutoCAD's internal rule, called the Arbitrary Axis Algorithm.

Given a unit-length vector to be used as the *z* axis of the ECS, the arbitrary axis algorithm, as described in AutoCAD's Reference Manual, generates a corresponding *x* axis for the coordinate system. The *y* axis follows by application of the right hand rule. If the given *z* axis is close to the positive World *z* axis, the *x* axis is found by crossing the World *y* axis with the given *z* axis. Otherwise the World *z* axis is crossed with the given *z* axis. The boundary at which the decision is made is chosen to be 1/64.

The complete steps to compute an ECS is as follows:

1- ECS origin coincides with WCS origin.

2- Orientation of *x* and *y* axis are calculated by the following algorithm:

Let × represent the cross product of two vectors,

Let N be the given unit vector along the ECS *z* axis,

Let W_x be a unit vector along the WCS *x* axis (1,0,0),

Let W_y be a unit vector along the WCS *y* axis (0,1,0),

Let W_z be a unit vector along the WCS *z* axis (0,0,1),and

Let A_x and A_y be vectors along the ECS *x* and *y* axes.

if (N_x<1/64) and (N_y<1/64) then

$A_x = W_y \times N$;

else

$A_x = W_x \times N$;

$A_y = N \times A_x$;

$A_x = A_x / norm(A_x)$;

$A_y = A_y / norm(A_y)$;

Coordinates are always represented as decimal (or possibly scientific notation) numbers, and angles are always represented in decimal degrees with zero degrees indicating the positive x axis. Note that the user of AutoCAD might have entered these values in other formats.

2.3. AutoCAD Layers

AutoCAD provides named layers for drawings. Every entity is constructed on a layer and thus has a layer name (a string) associated with it. The default layer name is the string "0". In AutoCAD, entities that are constructed in the same layer have the same color and line type (unless assigned an individual color or line type) and are turned on or off simultaneously.

The description of all layers used in a drawing are stored as table items in the LAYER TABLE section of the DXF file. Figure 2.2 shows the format of a layer item named *EXTERNAL-WALL* which is assigned color number 127 and line-type DOTTED.

```
 0
LAYER
 2
EXTERNAL-WALL     {layer name}
 62
127               {color index}
 6
DOTTED            {line type}
 70
0                 {bit is 1 if the layer is frozen}
```

Figure 2.2- The DXF format of a LAYER item

For our purposes, the only useful information in a LAYER item is its name. These names should be extracted from the layer table in the DXF file and written to a file for later use by the converter.

Layers are useful for grouping related entities or items of similar type in huge buildings. Making good layer assignments for individual polygons and blocks is a tedious and complicated task. It requires a clear and distinct set of objectives that the assignment should satisfy. An assignment with a minimal number of layers should be found, yet one that provides enough flexibility to change the rendering modes of various groups of objects individually by specifying a different layer treatment. This requires a lot of experience with use of AutoCAD models and may demand frequent updates as new features are added to the system. For example, our current layer assignment that is used for determining colors will have to be enhanced when texture mapping is added to our system. Also, as the interior decoraters decide to paint certain interior walls in more than one distinct color, the corresponding wall polygons need to be split and assigned to two

separate layers.

Architects typically use AutoCAD layers to group items of the same kind together for generation of various architectural blueprints and specifications. For example, external walls might be in one layer, internal walls broken into several layers according to their widths, and doors and window elements might again be in separate layers. Such an assignment certainly is useful, however, it might not be completely compatible with the needs of a walkthrough system. A walkthrough system, might use layers for setting surface attributes (e.g., color, material), or for extracting specific portions of a building required by the various preprocessing phases.

In our walkthrough system, the layer feature of AutoCAD is used to satisfy several needs. The primary use of it is for assignment of colors to surfaces. It is indeed a tedious task to assign colors to all polygons in a building on an individual basis. Since only architecturally-related polygons are grouped in the same layer, using layer names directly as color codes has led to satisfactory results. Layer names are also used to distinguish between entities that are solely included for floor plan generation (e.g., the arcs that show the swing of doors) from three-dimensional entities.

Another use for layers in our walkthrough system, is to distinguish between *major* and *minor* structural elements in a building. This is required by the visibility preprocessing algorithm. For this purpose, all walls need to be in distinct layers, so that the skeleton of each floor can be extracted easily by turning on a set of layers. This should produce a complete and continuous skeleton of the model without any holes, cracks or open-ended walls.

In general, good layer assignments are application dependent. The rules we came up with for our database and specific applications at this stage of the project are :

a- Layer names should be consistent across all floors and files.

b- A BLOCK should not share its layer name with its constituents.

c- Only entities that will always have the same color should be in the same layer.

d- Entities used for two-dimensional plan views should reside in distinguishable layers.

e- It should be possible to extract the *major* structural elements of a floor model by simply turning on a select few layers.

3. The DXF Converter

The basic function of the converter is to translate AutoCAD DXF files into the Berkeley UNIGRAFIX format. The information necessary for this conversion is embedded in the BLOCKS and the ENTITIES sections of the DXF file. In the conversion process, there are some steps that apply to all entities and some that are specific to each one. The general processing steps are described here, and details on conversion and format of individual entities are explained in Appendix A.

3.1. Conversion of AutoCAD Drawing Entities into Berkeley UNIGRAFIX Format

The converter reads each entity in the DXF file and, depending on the type of the entity (LINE, FACE, INSERT etc.), calls the appropriate procedure to collect all the geometric information scattered in various groups of the entity (for details refer to Appendix A). Non-extruded, open planar entities, such as simple lines, are ignored, and a polyhedral representation is computed for the rest. Since the AutoCAD database is basically a record of user actions rather than valid polyhedral objects, the generated polyhedron might not define a true three-dimensional object. For example an AutoCAD 3DFACE might actually consist of four co-linear vertices and have no area; or a closed POLYLINE might have only two distinct vertices. Therefore, the polyhedral representation is first checked for typical drawing and geometrical errors, such as having fewer than three distinct vertices, having colinear or duplicate vertices, and non-planarity. The validated polygons are then stored in UNIGRAFIX face structures.

The converter takes two optional files. One is a user-defined layer file which provides for selective processing of layers and color assignment. In this file, all layers must be stated along with a color index and with a tag to indicate whether the layer is wanted or not. If this file is provided, the converter checks the layer of each entity and processes it only if it resides on a layer that is wanted. In the absence of a layer file, all layers are assumed wanted. The second file is a color file which defines color values for 256 different colors that can be assigned to layers in the layer file. In the absence of a color file, random values are used.

When the DXF file is completely processed, the converter has generated a UNIGRAFIX "definition" data structure for each DXF BLOCK. A UNIGRAFIX definition contains a list of vertices, faces and block references. Faces and referenced blocks are assigned colors depending on the layer that they reside on. These colors are in RGB format and will be defined in the beginning of the UNIGRAFIX file. The ENTITIES section of the DXF file corresponds to the top level of the hierarchy in the UNIGRAFIX file.

3.2. Generating Floor Plan Views

The converter can also be used to create a plan view file for a floor. This plan view is the contents of the frame buffer, after an orthogonal top view of the floor has been rendered. The pixel values are then read into an array and written in binary format to a file which is used at display time. A corresponding UNIGRAFIX file is also generated that can be edited and used to regenerate plan views.

The conversion process required for generating plan views differs from the conversion for the three-dimensional UNIGRAFIX files in one respect. Since the layers used for plan views might contain two dimensional entities that are useful for the user in the walkthrough, the plan view converter, in contrast with the UNIGRAFIX converter, does not ignore non-extruded open planar entities and converts them to UNIGRAFIX wire statements. For example, the two-dimensional arcs that are used in architectural plan views to show door swings are converted in our plan views. This will aid the user to locate the doors that are installed in unexpectable locations, such as emergency doors, and notice the direction that the doors open.

4. Fixing Database Problems

The architectural databases that we had to work from are not true three-dimensional models. They are basically created for the generation of architectural blueprints such as plan views and crude three dimensional views of buildings. Drafting packages used to prepare them, including AutoCAD, were not written with anticipation that their products might be used in interactive walkthrough systems. Consequently, the first draft of the model that we received from the architects had many problems. Stairs and many other objects were missing; building components were not modeled as closed objects; windows were only line drawings on the walls; polygons were drawn with no consistent orientation; many co-planar polygons coincided, and many polygons intersected without sharing edges. Therefore, an element-by-element conversion of the architectural database to the UNIGRAFIX format would not have produced a viable model.

A walkthrough system requires a consistent 3D description of the building which is suited for radiosity analysis and for rendering with standard Z-buffer graphics hardware. It requires a description with the following features:

a- **All polygons are planar**: Nonplanar polygons cannot be rendered accurately by standard hardware. These errors are easily detected and reported during conversion and to a large extent, fixed automatically.

b- **Coplanar polygons do not coincide**: Coincident coplanar polygons of different colors cause problems. During display, these surfaces bleed through one another in Z-buffer algorithms and create noisy patterns. They have to be detected and their display priorities resolved.

c- **Polygon intersections are correct**: Improper edge adjacencies cause cracks in curved surfaces and shading discontinuities in planar surfaces. Polygons that arbitrarily intersect (i.e. do not explicitly share an edge on the intersection line) flicker through each other in the display. They might also cause incorrect shading results. For example, a ceiling polygon that is not cut up along its intersection with a partitioning wall might span two rooms. If it is illuminated in one room, but not in the other, linear interpolation of vertex colors allows light to *leak* through the wall. Edges that are incident on an odd number of polygons imply the existence of redundant polygons that should be discarded for efficiency. Correct edge adjacencies also aid in the determination of face orientations.

d- **Face orientations are correct and consistent**: Most shading programs, including the radiosity model, expect polygons to be consistently oriented and have distinguishable front and back sides. In UNIGRAFIX format, a polygon's front side is defined implicitly by stating its vertices in counter-clockwise order. Oriented faces also reduce display time by allowing back faces to be culled away.

e- **The model is architecturally complete**: Missing objects, for example missing stairs, result in incomplete and confusing images. There are no algorithms to detect entirely missing objects. It is basically the responsibility of architects to ensure model completeness.

Most of these problems can be detected and corrected automatically, while a few require direct inspection and correction in the modeling stage. In this section, these problems are described in more detail, and our algorithms for handling them are explained.

4.1. Fixing Nonplanar Polygons

Nonplanar polygons occur either because of attempts to approximate a curved surface with a number of quadrilateral polygons or as a result of errors made when entering coordinates. The first case can be fixed by tesselating the polygon. The second case, however, is ambiguous and has no automatic solution, however, a few useful heuristics can be applied.

To detect nonplanar polygons, plane equations are computed for all polygons using an algorithm by Newman and Sproull [New 79]. In this method, the plane coefficients A, B, and C (components of plane's normal vector) are computed such that the plane comes close to all vertices. A polygon is reported nonplanar if the distance between one of its vertices and the approximated plane is larger than a threshold ($1e-100$). In this case the polygon coordinates and the DXF line number on which the polygon was specified are reported. The polygon is either flattened or tesselated.

A nonplanar polygon is flattened if it is clear that its nonplanarity is the result of an error. To make this guess, the property of many architectural models, that the majority of their polygons are axial, i.e. parallel to two of the coordinate axes if a suitable coordinate system is picked, is used. Therefore, if by removing the offending vertex, the other vertices form a proper axial polygon, then it can be assumed that the polygon was intented to lie on this axial plane. The polygon is fixed by projecting the vertex into this plane.

If the polygon cannot be flattened into an axial polygon, it is tessellated into triangles. The polygon is projected on to a plane that comes closest to all vertices, i.e. the plane with coefficients computed by the Newman and Sproull algorithm. The tesselation is carried out on this plane by adding edges of shortest length to the polygon such that each edge lies between vertices of the polygon and does not intersect any existing edge.

4.2. Resolving Coincident Coplanar Polygons

Overlapping coplanar polygons occur either because multiple polygons used to cover a surface overlap or because two distinct surfaces coincide. In the first case, the overlapping polygons have the same material, while in the second case they differ. For example, in the CS building model, the arches are constructed by overlaying many polygons on each other to approximate the arch curves. Here the overlapping polygons are all in the same layer and thus have the same color. An example of the second case is seen when the outer shell of the building, which is constructed in a distinct layer and is supposed to cover the whole building, is added to the seven floor models, resulting in many overlapping polygons of different colors.

The first step to solve this problem is to identify all overlapping coplanar polygons and find their intersections. Program *ugcopl* [Smi 91] can be used for this purpose. *Ugcopl* cuts up overlapping polygons and merges them if they have the same color. For example, given a desk with a piece of paper on top, *ugcopl* cuts out a polygon that matches the paper out of the desk. If the paper and the desk were made out of the same material (i.e. they were constructed in the same layer), *ugcopl* will discard one of the coinciding polygons. Otherwise, two polygons with the same size and different colors will remain. The next step is to determine which one of the overlapping pieces to discard.

Ugclean, a preprocessor for the radiosity program [Smi 91], is available to clean models with overlapping polygons of different colors. This program uses the heuristic that the piece coming from the larger polygon should be discarded. This heuristic works for cases where small objects are placed on larger ones, however, it does not apply to many cases in our building model. For instance, in the above example many of the shell polygons have the same size or are larger than the building polygons with which they overlap.

To overcome this problem, *ugclean* was modified to consult a user-defined *priority* file to decide which polygons to discard. The *priority* file contains an ordered list of layers such that layer *i* overrides layer *i+1*. Whenever two overlapping polygons are detected, *ugclean* looks up their layers in the *priority* list and discards the one that has the lower priority. For example, when the outer shell of the building was added to the model, the layers pertaining to it were added to the beginning of the priority file before the model was passed through *ugcopl* and *ugclean*.

4.3. Improper Polygon Intersections

Polygon intersections are correct if three criteria are met:

a- Polygons that intersect share edges.

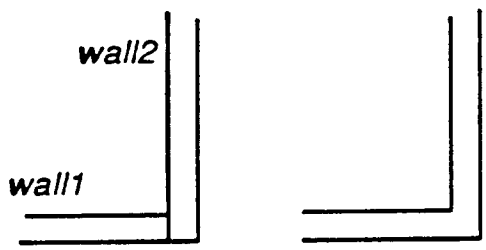
b- Edges that intersect share vertices, i.e., no T-junctions allowed.

c- Every edge is incident on two polygons except when two or more objects join.

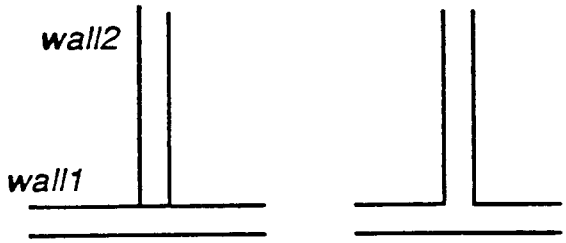
The first two criteria can be met by computing all intersections in a model and cutting polygons into pieces along these intersections. This is a well-defined problem, and programs such as *ugisect* [Seg 88] and *ugcopl* [Smi 91] already exist to do the job for non-coplanar and coplanar intersecting polygons, respectively. The difficult part is to meet the third criteria, i.e., to find and eliminate redundant polygons that are created as a result of the first step or exist because of modeling errors. To gain a better understanding of the problem we will look at some typical cases in our building model.

In the CS building database, improper polygon intersections can be found in four typical situations. First, they occur where wall polygons intersect floor or ceiling polygons without sharing edges. Such a ceiling or floor polygon might span two or more rooms and consequently pick up light intensity from all lights in these rooms, resulting in wrong shading solutions. Another potential problem here is a flickering display of wall edges through ceilings and floors. These intersections are usually not visible to the user. The gap between a ceiling and the floor of the next level prevents users from seeing the abutting walls through floors or ceilings.

The second case occurs in locations where two non-coplanar walls intersect without sharing an edge. Figures 4.1 and 4.2 show the top view of two typical improper wall intersections. During display, the left edge of *wall2* that incorrectly intersects *wall1* in Figure 4.1a, would flicker to a user who is looking at *wall1*. In figure 4.2a, *wall1*, which is not cut up along its intersections with *wall2*, causes light leakage from one side to the other.



(a)Incorrect and (b)correct intersection
Figure 4.1



(a)Incorrect and (b)correct intersection
Figure 4.2

The third case occurs in building components, such as windows and doors. These fixtures are usually constructed by pasting several objects together. For example a window block in our database consists of a frame that will be installed in the window opening of a wall, a narrow glass frame that is attached from the inside to the window frame, and a thin glass plate. These components are pasted together and therefore result in many incorrect intersections. The window frame should be cut up along its intersections with the glass frame; the glass frame should be cut up along its intersections with the glass; and the overlapping pieces should be removed.

The last case occurs when the window or door blocks are inserted into the building walls. Here, the border edges of the block and the edges of the building might overlap without sharing vertices, thus creating T junctions.

The first step to handle all these cases is to run the model through *ugvlmerge* followed by *Ugisect*. These programs are preprocessors used by the radiosity module [Smi 91]. *Ugisect* finds all intersections between polygons and *ugvlmerge* eliminates T vertices. These steps, however, might result in the generation of many redundant or unnecessary polygon pieces adjacent to the intersection points. Although these extra pieces will not have adverse effects on the result of the radiosity computation and can be left in the model, it is preferable to detect and discard them to avoid waste of time and space in the radiosity computation phase and at display time. Furthermore, eliminating these pieces and ensuring that all edges are shared by an even number of polygons aids in the determination of face orientations.

For this purpose, an interactive editor is provided that marks all incorrect edges, i.e. the edges that are incident on an odd number of polygons, and lets the user delete redundant polygons (refer to "ugorient" man pages in Appendix C).

4.4. Fixing Face Orientations

Buildings consist of simple objects, such as walls, doors, windows and stairs. If the modeling of these objects abides by certain rules, the orientation of polygons can be determined quite easily. Our experience has shown that two rules suffice:

a) All entities (walls, doors, etc.) in the building should be modeled as closed solid objects. For example no observer should be able to see the inside of a door slab or a wall.

b) Intersections should be correct, and redundant polygons should be removed. In other words, the edge adjacency problems discussed in the previous section should not exist.

If a database meets these criteria it is not difficult to determine the correct orientation of its polygons. Given an edge list with pointers to polygons and a few seed polygons with correct orientation, the complete solution can be obtained with a traversal of this list. During the traversal, the existing solution for a polygon will be propagated through the common edge to the neighboring polygons. For instance, if edge e is incident on two polygons A and B, and polygon A is correctly oriented then, the normal for B can be found by using the reversed direction of edge e. Another possibility is that the number of polygons incident on edge e is an even number greater than two. This is a rare case and occurs when two objects join each other at the edge. An example is the joint between a door and its frame. In this case, the incident polygons should be sorted by their relative angles first, and the edge direction used to propagate orientation should be reversed on every other polygon.

However, the assumption that a database be absolutely flawless is unrealistic. Architectural databases, particularly for huge buildings such as the CS building, are usually constructed with drafting tools and are rife with errors. Even after running the database through various cleaning filters, it contains many small redundant polygons resulting in edges that are incident on an odd number of polygons. The propagation algorithm applied to such databases can proceed as long as it can find correct edges and seed polygons in their neighborhood. Good results are usually obtained when there are many seed polygons scattered throughout the building.

We have used two methods to initialize the orientations of seed polygons in a building. The first method uses the Z-buffer algorithm and is based on the assumption that building entities are modeled as closed objects. The second method uses building properties to locate two sides of thick objects and determine their orientation. These initialization steps and the propagation algorithm are usually enough to fix orientation of almost all building polygons. However, to ensure correctness in the presence of major modeling errors, the user can run the initialization and propagation routines interactively, view the results, pick additional seed polygons interactively, rerun the routines and flip orientation of individual polygons.

The two initialization methods serve one other purpose. It is indeed a tedious and time consuming task, if not an impossible one, to ensure a model's correctness using drafting tools such as AutoCAD. The users of AutoCAD typically use a wire frame representation for viewing their models † where it is not always obvious which collection of wires belong to the same polygon. The above methods, embedded in an interactive viewer, are also used as error-detectors. They are used to detect open objects, missing polygons, and improper intersections. Errors are displayed using color-coded polygons. Different colors are used to identify polygons that have proved to be erroneous, as well as polygons that the two programs could not make a decision about, and finally polygons that were successfully fixed. The two methods are explained in the following Sections.

---

† AutoCAD has a hidden face removal feature. It is a very time consuming process and not efficient for our purposes.

4.4.1. Method 1

Our first approach to find the proper orientation of faces relies on the assumption that building components are modeled as closed objects. By judiciously picking a set of view points inside and outside the building and determining the set of visible polygons for each one of them, the visible side of these polygons can easily be determined. This method is implemented efficiently using the hardware Z-buffer capability of modern graphics work-stations. The Z-buffer method updates two tables for each pixel, the depth table and the color table. The polygon identification numbers are used as color values. After rendering the scene from a view point, the visible polygons are identified by reading the color values in the frame buffer. Having a view point and its set of visible polygons, the correct orientation of each one can be determined with a single test. A polygon's orientation is correct if its normal (computed by assuming a counter-clockwise vertex order) points toward the view point.

In other words, if A, B and C are coordinates of the face normal, and if x, y and z are a point (vertex) on the polygon, and x', y' and z' are the coordinates of the view point, the test is whether the equation $Ax'+By'+Cz'+D \geq 0$ is satisfied (where $D=-\begin{bmatrix} A,B,C \end{bmatrix} \cdot \begin{bmatrix} x,y,z \end{bmatrix}$ obtained from the polygon plane equation $Ax+By+Cz+D=0$). If this inequality is not satisfied, we know that the polygon normal is pointing in the opposite direction and the order of the vertices should be reversed.

This algorithm requires a flattened model and can not be applied to individual blocks, since many blocks do not represent a complete and closed object. At the same time it is advantageous to retain model hierarchy as long as possible. Therefore, after the analysis is carried out on the flattened instances, the results are combined and recorded in the master block. This requires that during scene flattening, links to the original blocks be established. This is easily implemented using the hardware capability of the SGI machines to carry out transformations. For each instance the transformation matrix is computed and pushed on the transformation stack, and the master block is rendered using the address of each face structure as its color value. With this scheme, the user can check the results only once on the original blocks and, if necessary, apply changes to the original block. The other advantage of this scheme is that in case there are conflicting outcomes for a face in different instances of a block, which might occur because of an incorrect insertion or cracks in the model, the cumulative result of all instances, based on majority vote, determines the final decision.

Another issue is how and where to pick the view points and viewing directions. There are many possibilities. One can use the position of light sources (if available) in the building with a few default directions. Light sources are a good choice since we know that usually at least one will exist in each room. Another possibility would be to pick a view point in the middle of each room. This can be done easily by AutoCAD or our interactive editor.

We started by using a special feature of our database. In our database, all rooms are tagged with their numbers. The tags are usually located at the center of rooms in the floor plan. The position of these tags were used as inside view points. A few outside view points around the building were added to this list. For each view point four directions (east, west, north, south) were picked for rendering.

This method succeeds only if all entities in the building that are supposed to be thick (e.g. walls, doors, etc.) are modeled as closed thick objects. Open objects and missing polygons mar the result of this process. For example, in the early drafts of our database, many walls were not closed at their ends; they were modeled as two parallel polygons. As a result, both the inside and

outside of the walls were visible. The above method was used as a tool to detect such walls.

4.4.2. Method 2

Buildings have properties that can be used to determine the orientation of many of their polygons. One property is that they typically abide by certain construction rules: walls are of a few discrete thicknesses, and rooms have certain minimum sizes. For example, if one takes the skeleton of a building (just walls) and classifies all walls by their thickness, a limited range is discovered. In the CS building, the walls range between 4 and 36 inches, assuming a few discrete values in this range. One can use this property to deduce the orientation of wall polygons. The idea is to identify two sides of a wall and have them point in opposite direction.

To speed up this process, we have exploited a feature of the CS building and many other buildings. The CS building has the property that the majority of its polygons are axial, i.e. parallel to two of the coordinate axes if a suitable coordinate system is picked. In our implementation, polygons are divided into 4 classes: parallel to x and y axes, parallel to y and z axes, parallel to z and x axes, and non-axial. The axial polygons are then sorted into stacks (Figure 4.7). This confines the search for two sides of a wall to a few layers of a class.

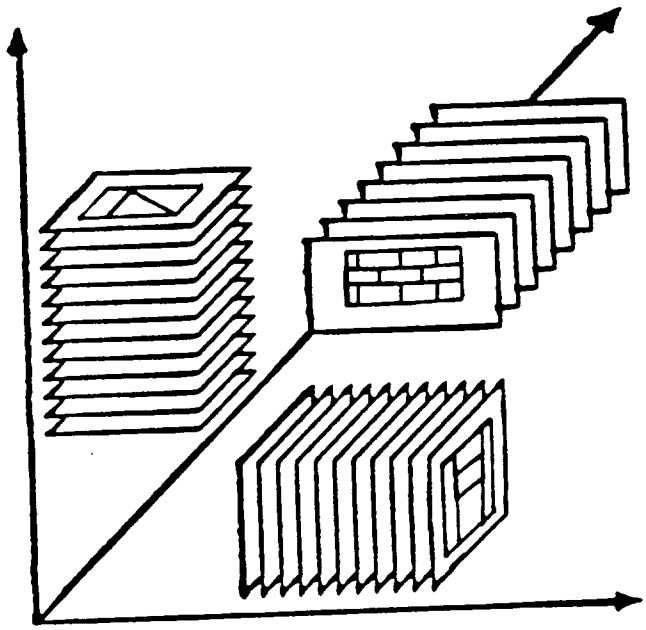


Figure 4.3- Building polygons divided into three sorted lists of parallel planes.

For each class and each polygon in its collection of sorted planes, the neighboring planes are searched for a match, such that the two polygons qualify as two sides of the same wall. If such a match is found, then their orientations should be fixed to point outward and in opposite directions. The necessary conditions for two polygons to be considered as two sides of the same wall are:

a- They overlap completely or partially in their perpendicular projection.

b- Their distance is less than one of the acceptable wall thicknesses.

The above criteria, however, are not sufficient. If two polygons overlap and are apart by a small distance, there is a good chance that they belong to the same wall, however, several contradictory examples exist. In cases where several walls are placed close to each other sometimes their distances lie in the range of acceptable wall thicknesses. In these cases, using only the above conditions to detect two sides of a wall might lead to two polygons from two different walls being matched together. Furthermore, in the presence of modeling errors such as single sided walls, these criteria will fail to detect the missing side and might match the single sided wall with a polygon from another wall if one is sufficiently close.

Although sufficient conditions that are applicable to all buildings and are capable of handling all possible modeling errors could not be stated, several restrictions can be placed to increase the possibility of detecting correct matches. For example, restrictions such as picking two polygons that overlap completely in their perpendicular projection, have the same color/layer and the smallest possible distance leave almost no possibility for an erroneous match. Based on detailed examination of the various wall arrangements in the CS building, several features have been discovered that can be used as restrictions to aid in the correct detection of wall polygons. These features, explained in detail later, are used to complement the above criteria and have proved to successfully detect more that 95% of the polygons in the CS building.

The other problem in the process of fixing wall orientations is error propagation. Once two polygons are picked as the sides of a wall and are fixed appropriately the outcome will affect further decisions. For example, if two polygons from two different walls are erroneously picked as sides of the same wall, then there is a good chance that the other two polygons of these walls will be matched with polygons belonging to other walls. Thus the error may be propagated further through the database.

To reduce the possibility of erroneous pairings and their propagation, the necessary conditions mentioned above are complemented with additional restrictions and are applied iteratively to the classes of sorted polygons. In each iteration, the algorithm searches for candidate pairs, i.e. polygons that satisfy the two necessary conditions, and applies a different combination of restrictions to determine whether a candidate pair qualifies as the sides of the same wall. The restrictions are tight in the beginning and are relaxed after each pass, allowing well-modeled, clear cases to be fixed first. Once a decision is made about a polygon, subsequent passes can make use of it, but cannot modify it.

This approach not only decreases the possibility of error propagation but also aids in the determination of orientation of ambiguous cases, since the number of undecided polygons are reduced at each step, leaving less candidates to be matched with ambiguous polygons.

The restrictions used in our algorithm are based on:

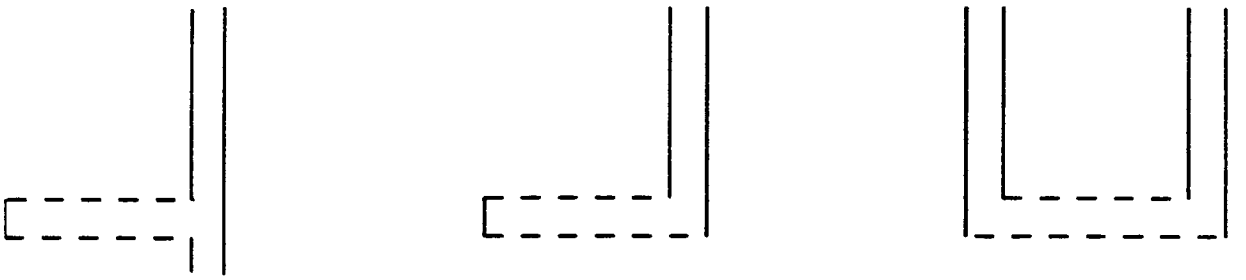
a- **How well a polygon and its match overlap:** The best case of course occurs when a polygon and its match completely cover each other in their perpendicular projection. We call such a match a match of "type one". As shown in Figure 4.4(b,c), there are two other

cases that also fit under this category. The first occurs when a wall intersects two other walls with the same thickness at each end (Figure 4.4b). In this case the ends of the two walls differ by one wall thickness (their distance). Figure 4.4c shows the third case where a wall intersects another one only at one end. In this case the two polygons have the same value at one end and differ by one wall thickness at the other end. The second category, called "type two", occurs when a wall does not intersect any other wall at one end and intersects another one with a different thickness at the other end. In this case its two polygons have the same value at one end (Figure 4.5a). The third category includes polygon pairs where one covers the other completely and the fourth category contains the rest of the overlapping polygons. Figures 4.5b and 4.5c give examples of the third and fourth category, called matches of "type three" and "type four", respectively. Matches of type one are the best qualifiers for wall polygons and matches of type four carry a high risk of being faulty.

b- **Distance of polygons:** In buildings, walls are usually placed apart by a distance, say at least 2 feet, that would allow a person to pass through easily. If two polygons are placed closer than this minimum, it could be assumed that they belong to the same wall. However, there are exceptions like columns and window seals where the distance between two walls lie in the range of acceptable wall thicknesses. These cases are prone to faulty matches and thus are considered last to decrease the risk of error propagation. In general, to reduce the risk of erroneous matches in such cases it is best to identify thinner walls first. Figure 4.5 shows an example of a hollow column modeled with thick walls. In this case the distance between polygon *a* and *b* and between *a* and *d* are both in the range of wall thicknesses and both cases are type one matches. Here, it is essential that polyon *a* and *b* be matched first, rather than polygon *a* and *d*.

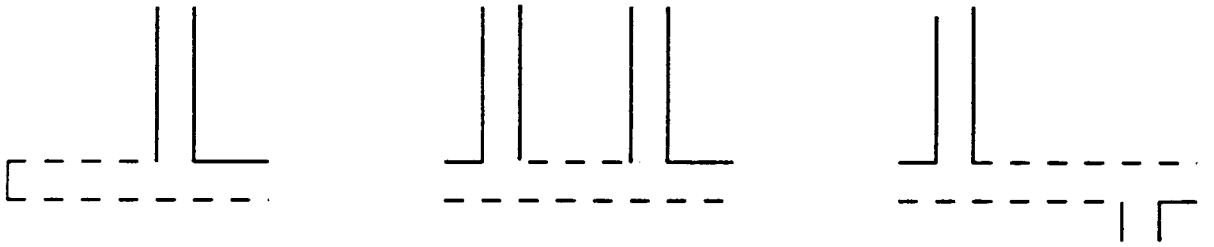
c- **Size of polygons:** Large polygons typically belong to walls that run along the sides of rooms. These walls are seldom placed close to each other and are the least prone to modeling errors. On the other hand, small polygons which are basically used to model columns and close walls at their ends can be found close to each other and constitute the most modeling errors. Therefore matches with large polygons are better qualifiers for wall polygons than small polygons and should be considered first.

d- **Polyon colors/layers:** Polygons in the same layer have a better chance of belonging to the same wall. However the reverse is not true, since two sides of the same wall might belong to different rooms and be in different layers because of having different surface properties.



(a) (b) (c)

Figure 4.4- The three cases of a type one match (dashed lines).



(a)Type two (b)Type three (c)Type four

Figure 4.5- Examples of other match types (dashed lines).

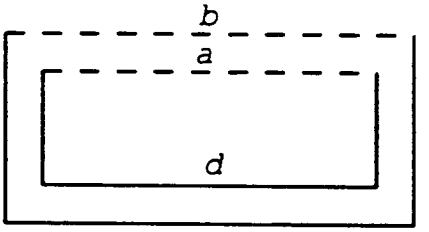


Figure 4.6- Finding matches in a hollow column.

These criteria are based on characteristics of wall polygons and might not be adequate for detecting ceiling and floor polygons. Ceiling and floor polygons are often modeled without thickness as single polygons. One solution is to use a specific layer name to identify them and assign an orientation (up or down) to all polygons in that layer. Such an approach requires usage of specific layer names in the model. Another approach is to search for ceiling and floor pairs in the same room rather than two sides of a floor or a ceiling. In our algorithm, when processing the polygons parallel to x and y axes, in addition to the above rules, two polygons are classified as a ceiling and a floor pair if they overlap and their distance is at least as large as a typical room's height. In this case the orientation of the two polygons are fixed so that they point toward each other.

Here is a more detailed description of the steps :

1- Divide all polygons into 4 categories:

a) Parallel to the x and y axes (called XYplane).

b) Parallel to the y and z axes (called YZplane).

c) Parallel to the z and x axes (called ZXplane).

d) None of the above (called Skews).

2- Sort each of the first three categories:

a) Sort XYplanes according to their z value (XYplane[i-1].z < XYplane[i].z).

b) Sort YZplanes according to their x value (YZplane[i-1].x < YZplane[i].x).

c) Sort ZXplanes according to their y value (ZXplane[i-1].y < ZXplane[i].y).

3- Fix orientations:

For Pass = 1 to 14

For each category c ($c \in$ {XYplane, YZplane, ZXplane})

For each plane l ($1 \le l \le n$) in the category c

For each polygon p in plane l

Ripple through planes $l-1$ to 1 and stop as soon as a match is found for p that satisfies the necessary conditions; look for other matches in this plane and if several are found pick the one with the best match type. Do the same using planes $l+1$ through n. Between the two matches, one on each side, pick the one that is closest to p.

In each *Pass*, fix orientations of p and its match and mark them done if they satisfy the restrictions of the current *Pass* as stated below:

Pass 1- If the two polygons belong to layers that imply their orientation (identified by detection of predefined strings in their layer names) fix them.

In the following passes t assumes values in the range of wall thicknesses, starting with the minimum thickness and ending with the maximum thickness.

Pass 2- thickness t, type one match, same layer, large.

Pass 3- thickness t, type one match, large.

Pass 4- thickness t, type two match, same layer, large.

Pass 5- thickness t, type two match, large.

Pass 6- thickness t, type three match, same layer, large.

Pass 7- thickness t, type three match, large.

Pass 8- thickness t, type one match, same layer.

Pass 9- thickness t, type one match.

Pass 10-thickness t, type two match, same layer.

Pass 11-thickness t, type two match.

The next two passes are only for the XY category (ceiling, floor, stair polygons); *tp* stands for the match type and assumes one of the four match types starting with type one and ending with type four.

Pass 12-thickness greater than minimum room height, type *tp*, large.

Pass 13-thickness greater than minimum room height, type *tp*.

Pass 14-For each remaining undecided polygon, find two matches, one on each side. If both matches have the same orientation, then assign the opposite to this polygon.

4- Propagate orientation of fixed polygons through shared edges.

5. Enhancing Rendering Speed

To make our walkthrough truly interactive, the visibility problem has to be solved in an efficient manner, particularly for large buildings, such as the CS building, with millions of polygons. In our walkthrough system, visibility information is precomputed [Tel 91] and stored for use in the interactive phase. The visibility computation is divided into three phases: the subdivision phase, the cell-to-cell visibility computation phase, and the query phase. The first two phases are precomputed before display time, and the last occurs during the walkthrough.

In this Section a brief description of the current state of each phase is provided. Detailed descriptions can be found in the mentioned references.

5.1. Spatial Subdivision

The first phase of the visibility computation is a spatial subdivision of the building model into cells, from which, typically, only a small coherent fraction of the model is visible.

The subdivision scheme requires identification of the building's *major* and *minor* structural elements [Tel 91]. The subdivision proceeds only with respect to the major elements to produce cells. Minor structural elements, such as furniture, door slabs or stairs, do not participate in the subdivision. However they will get assigned to their containing cell. An interactive tool is provided to identify and extract the major elements of the building (refer to "ugedit" man pages in Appendix C).

Here it is crucial that the major elements of the building be clearly distinguishable and constitute a closed, complete and continuous skeletal representation of the building. Basing the subdivision and the visibility computation on incomplete models that contains many holes and cracks might lead to unnecessary computation. For example erroneous holes and cracks in walls introduce cells that are visible to many other cells.

5.2. Cell-to-Cell Visibility Computation

Once the spatial subdivision has been constructed, for each cell, all other cells that might be visible to an observer moving freely inside the given cell are determined. This cross-listing of cells, or "cell-to-cell visibility" list, contains a superset of the truly visible polygons and is stored for use during the interactive phase of the walkthrough. This involves identification of *portals* (i.e., non-opaque portions of shared boundaries) on cell boundaries. Portals are used to form an *adjacency graph*, in which two cells are connected if and only if there is a portal connecting them. Cell-to-cell visibility information is computed for cells between which an unobstructed sight-line exists [Tel 91].

5.3. Visibility Query

Finally, in the walkthrough phase, an observer's eye is tracked through the cellular structure, and the cell currently occupied is found. The geometry of the cell and its portals along with

its precomputed cell-to-cell visibility list are returned in a query. The cells in this list and their contents are the only data that needs to be processed in the current frame. If the building model resides on disk, then the visible cells are brought into main memory.

Using the exact known position, viewing direction, and field of view of the observer, the cell-to-cell visibility list is further pruned to an eye-to-cell visibility list which contains all cells that are partially or completely visible to the observer [Tel 91]. The polygons from this reduced list are then sent to the rendering pipeline.

6. Display System

Once a suitable building model has been created, it must be made easy for the user to explore it. We have designed a display program with a simple but effective user interface for moving through the building and looking around. Figure 6.1 shows the basic components of our display program.

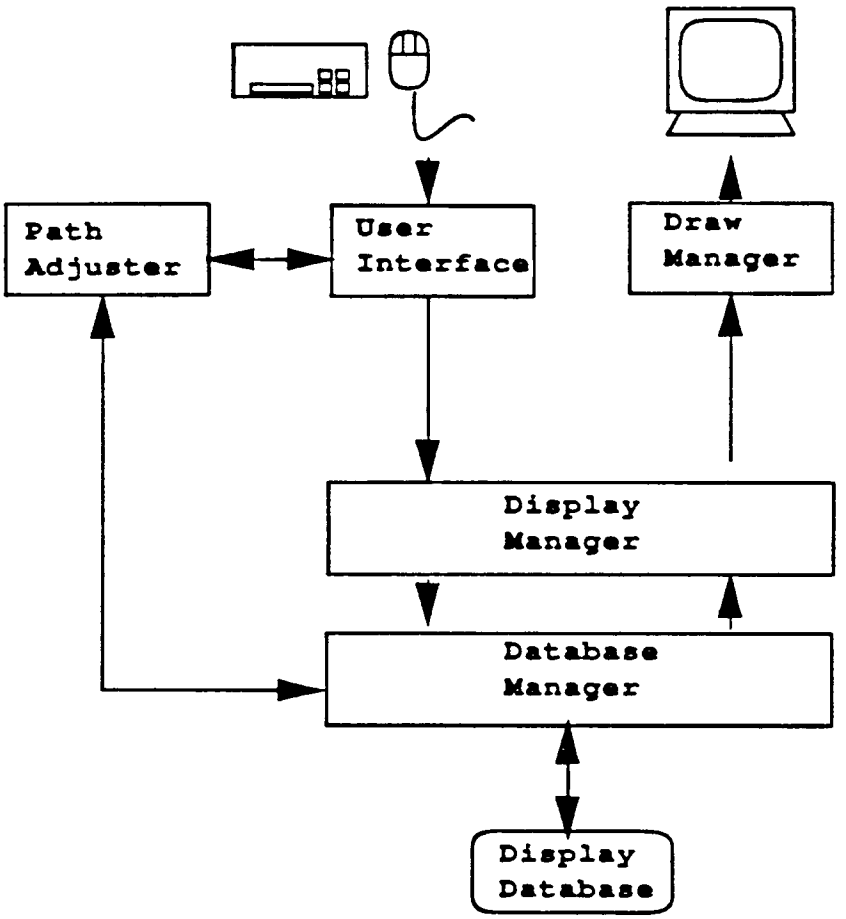


Figure 6.1- Components of our display program

In this section the five components of our display program, i.e. display manager, user-interface, path adjuster, data base manager, and draw manager will be explained.

6.1. Display Manager

The central component of the display system is the display manager. The display manager receives input from the user-interface, which consists of a viewing frustum and various display

parameters. The viewing frustum is then passed in a query to the data base manager to locate the cell containing the view point and retrieve its visibility list. Finally the list of polygons in the visible cells along with rendering parameters are given to the draw manager and the scene is rendered.

6.2. User-Interface

The user interface manager handles all interactions with the user. Its role is to provide friendly means of data entry through menus, dialog boxes, keyboard and mouse control, and to gather and interpret input data from the user for the walkthrough system. We started with the user interface used in he UNC Walkthrough system [Air 90a] [Air 90b] and added several features to it.

Our user interface, like the UNC system uses the mouse as the basic input device. Three windows are provided for the walkthrough: the scene window, an option window, and a plan view window. The user interface detects mouse input in the three windows and interprets them appropriately. If a change in the view frustum or any of the viewing parameters is detected, the user-interface will consult the path adjuster to apply the user-selected constraints to the viewing frustum. The adjusted viewing frustum and the rendering parameters are returned to the display manager.

6.2.1. Scene Window

The actual walkthrough is displayed in the scene window, where the user can navigate through the building using the three mouse buttons, and four arrow keys. The mouse buttons provide smooth continuous motion forwards and backwards and let the user change viewing directions. The up and down arrow keys provide elevator style motion and the left and right arrow keys lets the user step side-ways. Motion is oriented towards the cursor and the user simply positions the cursor where he/she wants to go/look and presses a mouse button.

A change in the state of these input devices results in a series of viewing transformations. When the user presses a mouse button, one of the arrow keys, or a combination of them, the distance between the current position of the mouse and the center of scene window along with the button combination are interpreted as a series of rotations and translations which are used to compute a new viewing frustum.

The default mode for motion is unrestricted, allowing the user to fly through the building in all directions, and to pass through walls. This unrestricted motion, however, is not always desirable, especially for novice users. As explained in Section 6.2.3, the user is provided with options to restrict the height of the eye point, maintain a viewing direction parallel to the current floor level, and make walls impenetrable.

6.2.2. Plan View Window

Based on experience with the UNC interface, we found the addition of a plan view window with the trace of the user position and viewing direction on it, to be extremely useful in preventing user disorientation and familiarizing first-time users with a building. This window also offers another convenient view selection tool for the user to quickly change views and examine various sections of a floor without having to travel explicitly to the desired location. The user picks a

point on the plan view with the left mouse button to specify an eye-point, then drags the mouse and releases the button to determine the viewing direction.

The plan view of each floor is loaded from a pixel file that is generated in advance by the AutCAD converter.

6.2.3. Option Window

The option window consists of several buttons and sliders that can be used to change rendering and viewing features. The following options are provided:

a- **Restrict view:** Based on some initial experience, it seemed advantageous to provide an option that restricts the user's freedom of motion inside buildings. With this option, the observer's eye position is constrained to six feet above the local floor and the viewing direction is kept parallel to the floor. This restriction is particularly helpful for novice users (maybe the architect's clients) or for control of fast motion inside buildings.

The path adjuster, explained in Section 6.2.4, is responsible for applying this restriction.

b- **Impenetrable walls:** It seemed to be desirable to prevent the user from accidentally moving through a wall into an adjacent room when exploring inside of buildings. Passing through walls, floors, or ceilings can lead to severe disorientation and produce confusing and meaningless scenes when a user is stuck inside wall interstices. The user has the option to make the walls impenetrable. With this option, the user is stopped when collision is detected, and a warning is issued in the plan view window (red flashing dot). In the future, we plan to maintain motion in case of collision. In this scheme, depending on the angle between the wall and the colliding view frustum, the direction of motion is adjusted to avoid the collision and maintain a continuous motion.

Collision detection is also carried out by the path adjuster.

c- **Changing floors:** This option lets the user change between floors quickly by picking at two buttons to go up or down. The name of the current floor is displayed next to the buttons. This option requires a user-defined file that contains the number of floors in the building, their names and elevations. This information is used to detect when the user moves from one floor to another and to update the current floor.

d- **Visibility Computation:** Several buttons are provided to view the effects of the cell subdivision and visibility precomputation and queries. There is the option to turn the use of visibility information on or off. If the display manager is using the results of visibility precomputation, the visibility query can be made either for cell-to-cell or point-to-cell visibility lists. In both cases, the user has the option to view the results on the plan window. With this option, the visible cells and the cell containing the current view point are marked on the plan window. This option is provided to show explicitly the results of the visibility precomputation during the walkthrough.

d- **Rendering quality:** The user can select between two rendering algorithms: wire frame and flat shading. In the future this set may be enhanced with Gouraud shading, radiosity, ray-tracing and texture mapping. The chosen algorithm has significant impact on both the rate and quality of frames in the walkthrough.

e- **Light intensities:** At this stage of our system, four directional (east, west, north, south) light sources and one ambient source are provided for scene illumination and shading. Each light source has a variable intensity. Interactive modification of the scene illumination is provided

via 5 slider bars. The user can change the intensity of any one of the five light sources interactively by sliding the mouse on the bars. This will result in the recalculation of color values by the draw manager for the next frame.

f- **Miscellaneous:** Buttons are provided to turn on back face culling and display help information on how to use the system. Information such as the current frame rate and current floor are also displayed in this window.

6.3. Path Adjuster

The path adjuster is called by the user-interface each time a change in the viewing frustum or viewing parameters is detected. The path adjuster applies the restrictions defined by the viewing parameters to the viewing frustum. The following restrictions are possible: constraining the viewing position to "eye level", keeping the viewing direction parallel to floor, and avoiding wall collisions. The path adjuster will set the height of the eye position to 6 feet above the current floor if the user has requested this restriction. The current floor level is obtained from a user-defined file. This file specifies the number of floors in the building and the zero level for each floor. The restriction on the viewing direction is applied by substituting the viewing direction vector with its projection on the *xy* plane. Here it is assumed that the floors are level and always parallel to the *xy* plane and the level change on stairs are ignored.

The path adjuster also checks the new view frustum for collisions with walls or other major structural elements of the building. Our current implementation uses the plan view to detect collision and halts the user before the actual collision.

The plan view is stored as a set of pixel colors corresponding to position of walls, doors, and windows. Since we only want to disallow the user from passing through walls, all we need to do is to somehow distinguish between wall pixels and the rest. During display, the view point is mapped to a pixel on the plan view, the color value at the pixel is looked up in the pixel array. If this color denotes a wall, the view point is declared to be illegal.

For this purpose, when a plan view is generated, a distinguishable color is used for walls or other major structural elements of the building. These elements are drawn using a color called the FORBIDDEN color. Since it is preferable to stop the user before the actual collision, walls are drawn using thicker lines. However, this will result in crowded plan views. To prevent this, two FORBIDDEN colors are used. One, called FORBIDDEN1 is the wall color and the other, called FORBIDDEN2, differs slightly with the back ground color. Walls are drawn twice. The first time they are drawn with a line width of three pixels using color FORBIDDEN2. The second time they are drawn with a line width of one with FORBIDDEN1 color. Both colors denote a collision.

This scheme is fast and simple, but suffers from a number of problems. First of all, the plan view description of a floor is not an accurate description of the floor. Furthermore, the mapping of view points to pixels introduces additional inaccuracies in the computation, particularly when the size of the plan window is small. The second problem is that this scheme cannot handle collision with ceilings or floor polygons.

A better implementation of collision detection relies on the cell structure and the information stored in the precomputed cells. In this scheme, the path adjuster queries the data base manager to locate the cell containing the new view point. If the cell containing the new view point is the same as the cell containing the previous view point, the new view point is declared

legal, since no cell boundary, in other words no *major* structural element, has been crossed. Otherwise, the geometry of the boundary between the two cells is used to check for collision.

Given two cells, one containing the previous view point and the other the new one, wall collision is detected by finding the intersection of a line joining the two view points and the boundary of the two cells. If the intersection point lies on a portal, the transition between the cells is accepted, otherwise a collision is reported.

In case of a collision, the new view frustum should be adjusted to disallow cell to cell crossings. The view point trajectory is clipped at a small distance from the wall and the viewing direction vector is rotated gradually until it coincides with its projection on the boundary of the two cells. This corresponds to a viewer in motion that collides with a wall, gradually changes direction and maintains motion by sliding along the wall. This feature and collision detection based on precomputed cells will replace the pixel-based collision detection in the future.

6.4. Data Base Manager

The data base manager is responsible for the storage, management, and retrieval of display data. It starts by loading the precomputed subdivision tree and the cell-to-cell visibility lists [Tel 91]. Model polygons are then associated with each cell. The data base manager is now ready to answer visibility queries and retrieve display data. Queries are made for three purposes:

a- **Point-location:** Given a point, the visibility manager might be asked to locate it in the subdivision tree and return the cell containing it. The information stored in each cell contains the geometry of its boundaries and portals (if any) which is used by the path adjuster to detect collisions with the walls.

b- **Cell-to-cell visibility:** Given a cell, a query might be made for its cell-to-cell visibility list. This list is precomputed and includes all the cells that are visible to any observer moving freely inside the cell. This query is done by the display manager if the user has selected the option to use the cell-to-cell visibility lists. The polygons associated with the cells in this list are given to the draw manager for rendering.

c- **Point-to-cell visibility:** Given a cell and an observer's viewing frustum in the cell, a query might be made for the cells visible to this observer. This list is a subset of the cell-to-cell visibility list of the cell containing the observer. This query requires some computation. This query is the default query type and is used by the display manager to retrieve display data for the draw manager.

6.5. Draw Manager

The draw manager is the rendering engine of the display program. It is called by the display manager to draw objects in all windows. It supports various routines to draw the building polygons in the scene window, trace the position and viewing direction of the user in the plan window, and show the results of the visibility queries on it. For now, the drawing algorithms used to draw the objects in the scene window are wire frame and flat-shaded. This list is going to be extended to include Gouraud shading, radiosity, ray-tracing, and texture mapping.

The draw manager is also responsible for adjusting the illumination of the scene to reflect changes in the intensity of the five light sources. During the walkthrough, any change in the intensity of the five light sources is reported to the draw manager. In the current implementation, where only flat-shaded polygons are rendered, this adjustment translates to recomputing color values for all polygons. Each polygon has two color values, one containing the original color and the other the adjusted display color. The display colors are computed for each polygon in the building, using the original color of the polygon, its angular relationship with the four direction light sources, and the intensities of the four directional light sources and of the ambient light.

7. Tutorial

This section describes the sequence of steps that have to be taken to prepare a DXF file, obtained from architects, for the walkthrough display. The exact sequence is of course dependent on the condition of the file. A file that contains a well-modeled building with correct face orientations require less processing than one with no notion of face orientations and many geometrical errors (which is the typical case). Here, we take one of the floor files of the CS building (*csb6.dxf*) as our test case. This file is about 170 KB and contains a description of the sixth floor with random face orientations, relatively few incorrect intersections and coplanar overlapping polygons.

To get a rudimentary idea of what the file contains, it can be simply converted without error checking to UNIGRAFIX format and displayed and examined with **ugitools** (an interactive viewing/editing program). The commands to do this are:

acad2ug *csb6*
ugitools < *csb6.ug*

This simple conversion takes about 17.5 seconds, and the generated file can be used as a first draft for the walkthrough. However, because of randomly oriented polygons in the model back face culling should be turned off, and rendering should be limited to flat-shaded polygons during the walkthrough. In addition, the resulting UNIGRAFIX file that is directly converted from the DXF file, might contain unnecessary layers, incorrect intersections and coplanar overlapping polygons. In general, correcting these problems is an iterative process. The walkthrough program and the editing tools should be used several times to detect and correct as many errors as possible.

The first step to clean the data base is to identify unnecessary layers and turn them off. This is done by creating a layer file (as explained below) and running the converter with the -l option. The next step is correcting face orientations. Two algorithms are provided to automatically fix the orientation of polygons. Experience has shown that using the algorithm described in Section 4.4.2 right after conversion (-f2 or -f3 options of **acad2ug**), followed by approximately thirty minutes checking and further correcting with **ugorient** suffice. For the sixth floor model, this algorithm extends the conversion process only by two more minutes and has proved to fix 97% of the polygons. The other option is using the algorithm described in Section 4.4.1. This algorithm will take much longer since it has to render the scene from different view points (approximately 4 seconds for each view point) and requires a view point file. It is better to create a view file and use this option only if the results of the other algorithm is not satisfactory. In this case, a few view points should be picked only in the neighborhood of polygons that are not fixed.

The final cleaning step is fixing modeling errors. To a large extent, incorrect intersections and coplanar overlapping polygons are fixed automatically. However, the many polygon pieces generated by this process should be checked visually and deleted if necessary. This is the most time consuming stage and experience has shown that it takes about two to three hours for each DXF floor file. In addition, there are other tasks that take time such as creating several files required by the various stages of the walkthrough process, editing colors, placing furniture and light definitions, and assembling the building. On the whole, the process of creating a complete and clean data base of the CS building from AutoCAD DXF files, will take about two weeks.

The detailed steps to process a file are:

1-Make a layer file:

acadlayers < *csb6.dxf* > *csb6.layer*

This will produce a layer file containing all layers used in the DXF file. As explained in Section 2.3, every entity in the DXF file is associated with a layer. The generated layer file will have these layers readily available in an easily accessible form. This file is used to select a subset of the layers for conversion as required by the specific application. This is done by assigning a tag (Y or N) to each layer. In addition, the layer file is used to assign color values to the layers. Originally, layers are assigned arbitrary color indices and are tagged with Y (layer turned on). The tags should be edited to turn unnecessary layers off. The colors can be edited to reflect true colors, however, this step can be done later in the UNIGRAFIX file interactively with **ugcolor**. If layer names are unfamiliar, the file can be converted as explained above, and **ugitools** can be used to interactively familiarize oneself with the different layers by turning them on and off.

2- Convert to UNIGRAFIX:

acad2ug -l *csb6.layer* -f3 *csb6*

This will produce two files, *csb6.ug* and *csb6.errors*. *Csb6.ug* is the converted UNIGRAFIX description and *csb6.errors* contains comments on conversion errors. The option -l with file *csb6.layer* is used to specify the layers that need to be converted and their associated colors. The option -f3 is used to fix the orientation of polygons using the method discussed in Section 4.4.2. With this option all polygons that are parallel to the *xy* plane and reside on layers with string "CL" (ceiling) and "FL" (floor) are fixed to point down and up respectively. Option -f2 should be used instead, if floors and ceilings are modeled with thickness or their layer names do not contain the mentioned strings.

3- Eliminate intersections and T vertices:

ugcleanup -m -i < *csb6.ug* > *csb6clean.ug*.

This will call the programs **ugvlmerge** and **ugisect** to process all faces in the UNIGRAFIX file, while preserving the hierarchy. New vertices are introduced to eliminate T junctions, and intersecting polygons are detected and cut at their intersection lines. As a result many small redundant pieces might be generated at wall intersections which should be located and removed using **ugedit**. This step might be time consuming depending on the condition of the model, and thus should be carried out on final versions of files.

4-Check the file:

ugitools < *csb6clean.ug*

Ugitools provides several interactive tools to check and correct the file:

ugview to examine each definition in the file and identify the layers;

ugcolor to edit colors and assign colors to instances, faces and wires;

ugorient to correct the orientation of faces;

ugedit to edit geometry of definitions;

ugplace to place definitions, such as furniture or lights.

5- Eliminate overlapping coplanar polygons:

ugcleanup -c -p *priorityfile* < *csb6clean.ug* > *csb6cleaner.ug*.

This will call the program **ugcopl** to process each definition and find coplanar overlapping polygons. Coplanar polygons are fixed by cutting up the larger polygon and discarding the

overlapping piece. If the two pieces have the same color one is arbitrarily discarded; otherwise a priority file is consulted and the one with the lower priority is discarded. *Priorityfile* is a file that contains the priority of overlapping layers (Section 4.2). It consists of pairs of layer names where the first one has a higher priority that the second one. It is important that at this point the orientation of polygons be correct so that a double-sided polygon modeled with two polygons with opposite orientations will not be mistaken as overlapping polygons.

6- Make a plan view file:

acad2ug -l *csb6.planvlayer* -p *csb6*

A special version of the layer file should be used for this purpose in which only those layers that should be used for plan view generation are turned on. In addition, all layers that contain walls should have the color index zero. This is used to detect wall collisions and make walls impenetrable during the walkthrough. Two files are generated: *csb6.planv* and *csb6.planv.ug*. The first file is a binary file and contains the plan view image. The second file is the corresponding UNIGRAFIX file that can be edited and used to generate plan view files with **ugview**.

7- Make a macro file:

The cell subdivision and visibility precomputation programs require a file containing only major structural elements for each floor. Use **ugitools** with the converted UNIGRAFIX file to remove the layers containing nonmajor structural elements such as door slabs, stairs, glass frames. This can be done by interactively turning the layers containing these elements off and saving the result in another UNIGRAFIX file. It is important that this description be complete and contains no cracks or holes.

8-Make a floor file:

This file is used by the walkthrough program and contains information about all floors of the building. The first line specifies the number of floors and the initial floor to be displayed. The name, elevation, and corresponding plan view file of each floor should follow in order. The format of the file is:

<number of floors> <initial floor>

<name of floor i> <elevation of floor i> <plan view file for floor i>

9-Compute visibility information and create the database:

Several programs are under development to create an efficient database out of the UNIGRAFIX files (floor files, furniture files, macro files) for the walkthrough program. For an up-to-date usage of these programs refer to **wksplit**, **wkcreate**, and **wkadd** on-line man pages.

10-Walkthrough:

ucbwalk -l *floorfile* -h *helpfile* -a <x y z> -f <x y z> -d *datafile*

The -f and -a arguments specify the initial position and viewing direction of the viewer respectively. The -l option is used to specify a file containing information about each floor in the building. The -h argument specifies a file containing information on how to use the walkthrough which will appear on the walkthrough scene when the "help" button is pushed. The default file is "ucbwalk.help".

The -d option specifies the database containing the description of the building at various levels and visibility information for the walkthrough.

8. Future Work

One of the major remaining tasks of our walkthrough team is to design and implement a modeling tool. Our experience with architectural models and the evolution of the needs of our walkthrough system, has clarified some of the major requirements for such a modeling system.

A building modeling tool should address issues such as ease of construction, model modification, and maintenance. At the same time it should be able to build upon databases generated by architects. Since plan views are the basic representation used by architects, an architectural modeling tool should be able to construct three-dimensional descriptions using their underlying architectural plan views. Computer-generated plan views with high precision and accuracy are now vastly available. These plan views have several features that can facilitate the three-dimensional construction. The most important one is the fact that architecturally-related items in these files are grouped into layers. Many drawing primitives can be applied to layers instead of individual items. For example, if all lines representing external walls are in the same layer, one can construct their corresponding three-dimensional description by extruding the layer EXTERNAL_WALL rather than the individual lines.

An architectural modeling tool should support a library of topologically correct building components and carry out model construction based on correct assembly of these components using boolean set operations. For example, a component can be a wall consisting of six polygons. When the user inserts this wall in between two other walls, the overlapping polygons are removed automatically. One major problem of most modeling tools is that they get harder to use as the model gets larger and more detailed. A representational mode of display would facilitate construction, checking and viewing of huge models. In this mode each component is assigned a two-dimensional symbolic token (e.g. color-coded lines with various widths) which will be used for display. To model a building, the user starts by constructing its various components (e.g. walls, windows, and doors of various dimensions) and assigning symbolic tokens to each one. Tokens are used to assemble the building. At any time, the user can switch to the three-dimensional display mode and view the real building. Another feature that facilitates construction of huge models, is the concept of layers. A building modeling tool should allow objects to be constructed on layers that can be turned on or off as needed.

Enhancements and modifications on other components of our walkthrough has already started. The user interface will be enhanced with several other options such as control of the navigation speed and the level of detail with which the objects in the scene are rendered.

The display manager and data base manager are being upgraded to answer the storage demands of huge models and support better rendering algorithms. Our enhanced display manager will support selection of two display modes: static and dynamic [Fun 91]. The data base manager will support storage of building objects at various levels of detail and provide for fast access and transfer of data through high band-width networks and massive disk-based storage systems [Fun 91].

Once algorithms to handle massive storage and fast transfer of display data are fully integrated in our system, the display of radiosity-colored polygons [Smi 91] will be added to our various rendering options.

Finally, the integration of the visibility precomputation library into our walkthrough system has introduced the potential for more accurate and efficient implementations of the path adjuster (Section 6.3).

9. Conclusions

The focus of this project was to complete major components of a walkthrough system. The test object was a description of the planned CS building generated by architects using AutoCAD. The first task of this project was to convert the architectural model to a format suitable for our walkthrough system (Berkeley UNIGRAFIX format), and to design algorithms to detect and fix major modeling errors, such as random face orientations and incorrect intersections.

One of our goals was to make this conversion automatic. We have used typical architectural properties of buildings and the fast rendering capabilities of SGI graphics work-stations to determine the correct orientation of building polygons. We have used existing UNIGRAFIX tools, such as *ugisect*, *ugvlmerge*, and *ugcopl* to detect and eliminate T junctions, incorrect intersections and coplanar overlapping polygons. In most cases, these algorithms detect and correct the errors automatically, however, in the presence of serious modeling errors, some direct user interaction and several passes of the algorithms might be needed. Particulary, user interaction is required to delete the unneccesary polygon pieces that might be generated as a result of cutting intersecting polygons along their intersection lines.

In general, our experience shows that as long as the original model is the product of drafting packages such as AutoCAD, the conversion process requires some user interaction, in particular in the error correction phase. Drafting packages used by architects are not true three-dimensional modeling tools. These tools are used mostly for generation of architectural blueprints and specifications such as plan views and crude wire-frame perspective views. Models generated with these tools have no topological consistency and are rife with missing polygons, cracks, incorrect intersections and randomly oriented polygons. The many hours that we spent on advising the architects on how to extend their two-dimensional plan views to three-dimensional descriptions, on identifying typical modeling errors, and designing algorithms to detect and correct them, on writing interactive editing tools for error correction, and finally on the detailed checking and correction of our models, made it clear that better modeling tools and techniques are required which are designed specifically to satisfy the needs of walkthrough building models.

The second task of this project was to implement a display program with an easy-to-use user-interface and with support for various rendering algorithms to explore buildings. Through experience, we have identified and implemented various features that facilitate such exploration. Our display program provides for both restricted and unrestricted motion. The former aids novice users and is advantageous when the user is traveling at high speed. We have found the display of a plan view with the current eye-position and viewing direction extremely useful. This plan view not only aids in preventing user disorientation but also serves as a tool to change view points quickly to explore different locations of a building. Another attempt to prevent disorientation and confusion is to disallow passing through walls. The user is provided with the option to make walls impenetrable.

Finally, the rendering speed of our walkthrough system has been improved by integrating the work of Seth Teller on building subdivision and visibility precomputation.

References

[Air 90a] John M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, UNC Chapel Hill, 1990.

[Air 90b] John M. Airey, John H. Rohlf, and Frederick P. Brooks Jr. Towards Image Realism with Interactive Update rates in complex virtual building environments. *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, 1990.

[Aut 90] AutoCAD Reference Manual, Release 10, Autodesk Inc, 1990.

[Aut 91] Frederic H. Jones, Lloyd Martin. The AutoCAD Database Book: Accessing and Managing CAD Drawing Information. Fourth Edition, 1991. Ventana Press. Chapel Hill, North Carolina

[Fun 91] Tom Funkhauser. Work in progress. U.C.Berkeley, 1991.

[New 79] William M. Newman , Robert F. Sproull. *Principles of interactive computer graphics*. McGraw-Hill Computer Science Series, 1979.

[Seq 83] Carlo H. Séquin, Mark Segal, Paul R. Wensley. *UNIGRAFIX 2.0 User Manual and Tutorial*. Technical Report no. UCB/CSD 83/161. U.C.Berkeley, 1983.

[Seq 91] Carlo H. Séquin, Kevin P. Smith. *Introduction to the Berkeley UNIGRAFIX Tools (Version 3.0)*. Technical Report no. UCB/CSD 91/606. U.C.Berkeley, 1991.

[Seg 88] Mark G. Segal, Carlo H. Séquin. Partitioning Polyhedral Objects into Non-Intersecting Parts. *IEEE Computer Graphics and Applications, 8(1):53--67*, January 1988.

[Smi 91] Kevin P. Smith. *Fast and Accurate Radiosity-Based Rendering* . Masters Report, U.C.Berkeley, 1991.

[Sre 91] Ajay Sreekanth. Work in progress. U.C.Berkeley, 1991.

[Tel 91] Seth J. Teller, Carlo H. Séquin. Visibility Preprocessing For Interactive Walkthroughs. *Computer Graphics (SIGGRAPH '91 Conference Proceedings), 25(4):61--69*, July 1991.

Acknowledgements

I wish to express my appreciation to my research advisor Professor Carlo H. Séquin for his guidance and continuous support throughout the course of my graduate studies.

I would also like to thank the UNC team, John Airey in particular, for getting us started with the UNC Walk-through user-interface.

I am indebted to many of the graduate students in the computer graphics group. In particular, the advice and suggestions of Seth Teller, Henry Moreton, Tom Funkhauser, and Kevin Smith are sincerely appreciated.

Thanks to Greg Ward for helping me with AutoCAD and for his generous donation of sophisticated furniture models.

Last but not least, I would like to express my deep gratitude to my husband Mehdi and my daughter Sepideh. Their support, love and patience was of crucial importance for the completion of my work.

Appendix A - Conversion of DXF Entities into UNIGRAFIX Format

In the following sections the specific conversion details for each AutoCAD entity along with the groups that make up the entity are described. As explained in Section 2, each DXF entity is described by a collection of groups. Each group consists of two lines: the first is the group code and the second line specifies the group value. In our explanations, group codes and values appear on the same line, which is not the case in the DXF file; the value of each group is described by a statement inside brackets; groups that are optional, are indicated by "-optional", and comments are stated inside curly brackets. One caution in reading a DXF file, is that one should not assume that the groups describing each entity always appear in the order stated here.

A.1. BLOCK

The DXF groups in a BLOCK are :

8	[layer name]	
2	[name of block]	
10, 20, 30	[x, y and z coordinates of the base point]	
70	[block type (bit coded)]	{*not used*}
0	[entities in block start]	

Each DXF BLOCK record has a name, a layer, a base point and a series of entities. BLOCKs are created in AutoCAD by grouping several entities and picking a point as the base point. AutoCAD then computes coordinates of all the entities in the block relative to this base point and writes the new coordinates to the DXF file. As a result, the base point always coincides with the origin of the block and renders useless in the DXF record.

During conversion, whenever a BLOCK entity is reached in the DXF file, a new structure for a UNIGRAFIX definition is created. The converter then processes all entities until an ENDBLK group is reached. These entities are coverted to a list of vertices, faces, and block references which will make up the new definition.

BLOCKs can also have attribute definitions. These definitions are stated under the ATTDEF group inside a BLOCK. For example a room block can have a definition for its number or area. Whenever a room is inserted, the user is prompted by AutoCAD to enter the room number and area. These attribute values are stored in the INSERT record. Currently attribute definitions are ignored by our converter.

BLOCK records sometimes include irrelevant groups such as a thickness value. We spent time searching AutoCAD handbooks, trying to figure out the purpose of these groups. We concluded that AutoCAD, regardless of its relevance, records the thickness and elevation variables whenever they are modified. We set one basic rule for processing DXF files: ignore groups that seem meaningless for the entity and process only groups that make sense.

A.2. LINE

The groups in a LINE are :

8	[layer name]
10,20	[x and y coordinates of the start point]
30	[z coordinate of the start point, -optional]

11,21	[*x* and *y* coordinates of the end point]
31	[*z* coordinate of the end point, -optional]
38	[elevation if nonzero, -optional]
39	[extrusion in *z* direction, -optional]
210, 220, 230	[vector along ECS *z* axis, -optional]

LINEs are defined by their two end points. The *z* value of the points can appear under group codes 30 and 31, or under code 38 as the elevation value; they might be missing if they have the value zero †. A LINE is ignored by the converter if it is not extruded. Otherwise, the two other coordinates of the extruded LINE are computed by translating the line endpoints in the *z* direction by the extrusion value (group code 39). If the LINE is constructed on an ECS (group codes 210, 220, 230), a transformation matrix is created (using AutoCAD rules explained in section 2.5). All coordinates are multiplied by this matrix and transformed to world coordinates. These four points define a face and are passed to procedure *MakeFace*.

A.3. ARC

The DXF groups in an ARC are :

8	[layer name]
10,20	[*x* and *y* coordinates of the center]
30	[*z* coordinate of the center, -optional]
38	[elevation , -optional]
39	[extrusion in *z* direction, -optional]
40	[radius]
50	[start angle in degrees]
51	[end angle in degrees]
210, 220, 230	[vector along ECS *z* axis, -optional]

An ARC is described by its center, radius, starting angle, and ending angle. ARCs are drawn counter-clockwise from the starting angle to the ending angle. An ARC is ignored if it is not extruded. Otherwise it is converted to a series of adjacent polygons approximating a curved wall. To do this, the perimeter of the ARC is first approximated. The missing vertices of the extruded ARC are obtained by translating the computed coordinates of the ARC in the *z* direction by the distance specified under group code 39. If necessary, the transformation from ECS to WCS is also carried out, similar to the LINE entity. Vertices are then grouped in quartets and passed to procedure *MakeFace*.

A.4. CIRCLE

The DXF groups in a CIRCLE are :

8	[layer name]
10,20	[*x* and *y* coordinates of the center]
30	[*z* coordinate of the center, -optional]
38	[elevation , -optional]
39	[extrusion in *z* direction, -optional]
40	[radius]

† This is true for all entities. So, to play safe, all coordinates should be initialized to zero before reading a DXF record.

210, 220, 230 [vector along ECS *z* axis, -optional]

The CIRCLE entity has only two defining measurements: its center and radius. CIRCLEs are processed even if they are not extruded, since they define closed areas. The conversion of a CIRCLE is the same as an ARC with starting angle zero degrees and ending angle of 360 degrees. An extruded circle would result in a cylinder with a top and bottom.

A.5. SOLID

The groups in a SOLID are :

8	[layer name]
10,20	[*x* and *y* coordinates of first corner point]
30	[*z* coordinate of first corner point, -optional]
11,21	[*x* and *y* coordinates of second corner point]
31	[*z* coordinate of second corner point, -optional]
12,22	[*x* and *y* coordinates of third corner point]
32	[*z* coordinate of third corner point, -optional]
13,23	[*x* and *y* coordinates of fourth corner point]
33	[*z* coordinate of fourth corner point, -optional]
38	[elevation , -optional]
39	[extrusion in *z* direction, -optional]
210, 220, 230	[direction vector of the ECS *z* axis, -optional]

In AutoCAD a SOLID is a filled quadrilateral or triangle. If a SOLID is triangular the third and fourth corners coincide. If the SOLID is a quadrilateral, it is defined by two opposing pairs of points, rather than by four points around the perimeter, as a normal rectilinear shape would be described. Thus, when reading the corners, the third and fourth points should be exchanged.

The corners of a SOLID are directly used as the coordinates of a face. These corners are extruded if an extrusion value appears in the SOLID's record. An extruded solid has both a top and bottom. The conversion from ECS to WCS is carried out similar to the LINE entity if group codes 210, 220, 230 appear in the SOLIDs record.

A.6. POLYLINE

The groups in a POLYLINE are :

8	[layer name]
40	[default starting width, -optional]
41	[default ending width, -optional]
70	[polyline flags] bit-coded as follows:

1	closed polyline
2	curve-fitted
4	spline-fitted
8	3D polyline in R3
16	3D polyline Mesh in R3
32	closed polyline mesh in the *n* direction

71	[polygon mesh *m* vertex count, -optional]
72	[polygon mesh *n* vertex count, -optional]
73	[polygon mesh smooth-surface *m* density, -optional]
74	[polygon mesh smooth-surface *n* density, -optional]

75 [smooth curve type flag, for 3DPOLYLINEs only] as follows:

0 not curve fitted
5 quadratic B-spline
6 cubic B-spline

0 {*a series of vertices will follow*}

The format of a vertex in the POLYLINE is :

8 [layer name]
10,20 [x and y coordinates of vertex]
30 [z coordinate of vertex, -optional]
38 [elevation, -optional]
40 [starting width, -optional]
41 [ending width, -optional]
42 [bulge factor for a polyarc, -optional]
50 [curve-fitted tangent direction in radians]
70 [polyline flags] bit-coded as follows:

1 extra vertex added by AutoCAD to smooth the curve
2 curve-fitted tangent has been defined(group 50)
8 extra vertex added by Spline fitting
16 spline-frame control point
32 3D polyline vertex
64 3D polyline mesh vertex

0 SEQEND {*indicates the termination of VERTEX records*}

The POLYLINE primitive is the most complicated entity in AutoCAD to convert. It is the one versatile command that is capable of producing almost anything, from simple wires to faces and complex smooth surfaces. A POLYLINE record has a header that contains information about its type (2D, 3D, mesh...). This header is followed by a series of VERTEX records. The 3D and 2D POLYLINE descriptions are both included under the POLYLINE entity. The POLYLINE flags and some group values are used to distinguish between them.

A.6.1. 2D POLYLINES

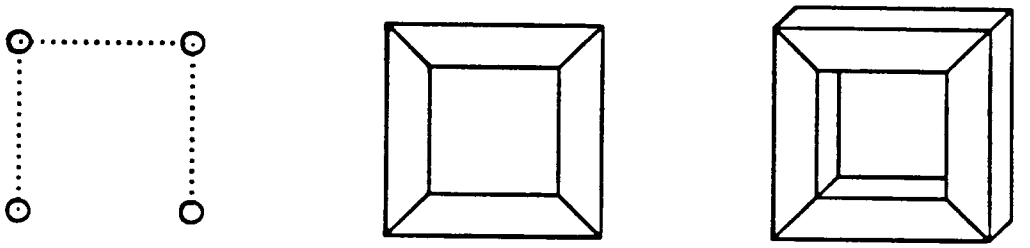
A 2D POLYLINE looks like a series of line and arc segments connected to form a continuous entity. It can have a specified width which can be varied from one end of a segment to the other to form a taper. It can be extruded or closed. Like all other planar entities, 2D POLYLINEs can be constructed in ECS.

A 2D POLYLINE is converted only if it is closed, or extruded, or has non-zero width. If the 2D POLYLINE is closed (but not extruded), it is converted to a single face. However, if a POLYLINE is closed and extruded at the same time, it is converted to several faces approximating a rim surrounding the face without top and bottom faces.

With the POLYLINE command a user can also create arcs. These are called polyarcs in AutoCAD. Polyarcs are not used in our database and not implemented in the converter.

A.6.2. Wide POLYLINE

The wide POLYLINE is the trickiest entity. With the POLYLINE command, one can draw a series of line segments and assign different starting and ending widths to each of them. The result will be a continuous series of filled faces. The wide POLYLINE can also be extruded. Figure A.1 shows a window frame that has been drawn using the wide extruded POLYLINE. Using the POLYLINE command, the coordinates of a rectangle and width values for each line segment are entered by the user (Figure A.1a, A.1b). The wide POLYLINE is closed and extruded (Figure A.1c) to make a three dimensional thick window frame.



(a) Coordinates of the wide POLYLINE as entered by the user

(b) The wide closed POLYLINE

(c) The extruded wide POLYLINE

Figure A.1-A window frame drawn using the wide extruded POLYLINE

The DXF record for a wide POLYLINE consists of a series of VERTEX records, defining coordinates of all line segments and their widths in the order that the user has entered them. The starting and ending widths of the POLYLINE are recorded in the header of the POLYLINE record. However, they might also appear in VERTEX records of the POLYLINE. A VERTEX record contains a starting and ending width value, only if the width has changed at that vertex point.

The converter uses the starting and ending width values to *widen* each line segment. This is done by first computing *side-lines* for each segment. These are lines that run along the perimeter of the final wide POLYLINE. Consider a wide POLYLINE defined by n vertices and $n-1$ segments. Each vertex v_i $(1 \le i \le n)$ has a starting width ws_i and an ending width we_i. For each segment i $(1 \le i \le n-1)$ four points are computed, two on each side, through which the *side-lines* pass. The four points for *side-lines* corresponding to segment i are found by shifting its endpoints, v_i and $v_{(i+1)}$, in the direction perpendicular to segment, by amounts $ws_i/2$ and $we_i/2$, respectively, and passing lines through them. After processing all segments, intersections of the *side-lines* are computed on both sides of the original POLYLINE to find the corner points of the final wide POLYLINE. These corners are now ready to be passed to procedure *MakeFace*. However, converting the wide POLYLINE, in this manner, to a single polygon, often results in an oddly-shaped concave face. To avoid this problem, we chose to decompose the wide POLYLINE into

several four-sided faces.

The example in Figure A.2 shows the entire procedure. In this example, the user has entered four points (hollow circles) with the POLYLINE command, defining three line segments (Figure A.2a). Each vertex i has a starting width of ws_i, and an ending width of we_i. Figure A.2b shows how the converter shifts the four points (filled circles), constructs eight *side-lines* (dashed lines), and computes their intersection (squares in the Figure) to find the final corners of the wide POLYLINE. In this case, the wide POLYLINE is decomposed into three quadrilaterals (Figure A.2c). Notice that representing this wide POLYLINE as one complex polygon would result in a concave face.

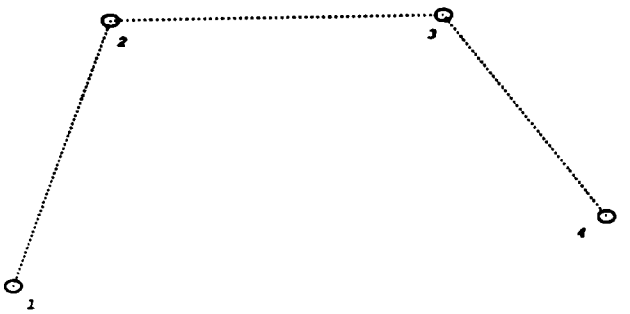
The wide POLYLINE can be closed, extruded, and/or be constructed in ECS. If the wide POLYLINE is closed, an additional line segment connecting the first vertex to the last one is assumed and processed similar to other line segments. If the wide POLYLINE has an extrusion value, all components of the computed wide POLYLINE are extruded. An extruded wide polyline results in an entirely closed object. During extrusion, the creation of hidden polygons inside the object at the boundaries between segments is avoided. If groups 210, 220, and 230 appear in the POLYLINE's record, the conversion from ECS to WCS is carried out similarly to the LINE entity.

A.6.3. 3D POLYLINE

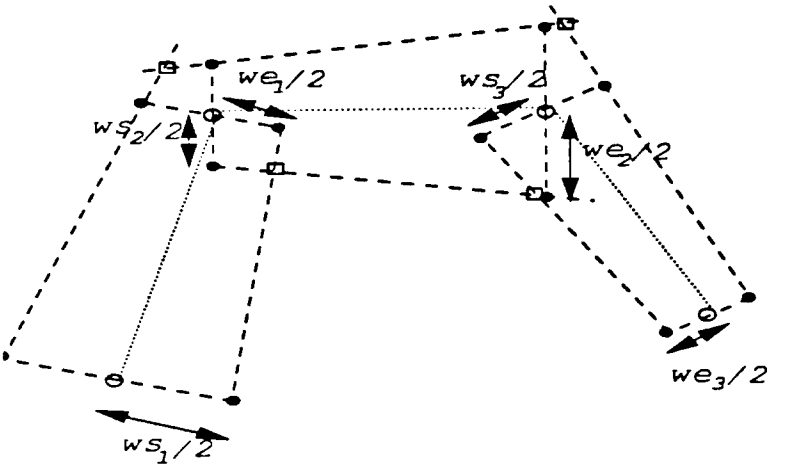
A 3D POLYLINE is distinguished by the value of group code 70. If this value is greater than 7, then that POLYLINE describes a 3D POLYLINE. A 3D POLYLINE is not allowed to have a width or thickness. Its coordinates are always 3D world coordinates. There are two kinds of 3D POLYLINEs. One is generated by the 3DPOLY command and is a zero-width, non-extruded simple polyline in 3D space, which can be smoothed, spline, or curve fitted. This type of POLYLINE is converted to a single face only if it is closed. The corners of the face are obtained from the POLYLINE's VERTEX records.

The other kind is the 3DMESH, which is used to describe complex curved surfaces. It is generated by the 3DMESH, RULESURF, TABSURF, REVSURF and EDGESURF commands in AutoCAD. AutoCAD stores the coordinates of the mesh as an m by n grid that runs across rows (n) and down columns (m). Group codes 71 and 72 specify the values of m and n. To convert a 3DMESH, the VERTEX records are read into an m by n array, and are used to cover the mesh with $(m-1)(n-1)$ polygons. If the mesh is closed (in either m or n direction), an additional row or column of polygons are created to close the surface in a cylindrical manner.

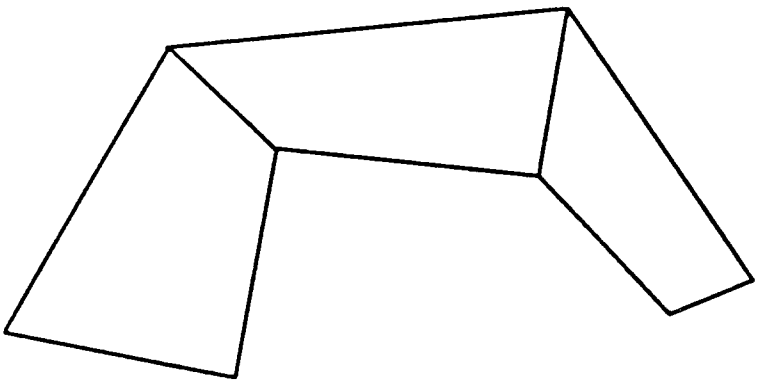
If the PEDIT command is used to smooth the mesh, additional vertices are created by AutoCAD. Group codes 73 and 74 specify the number of these vertices on each side. This feature is not implemented in our converter.



(a) Line segments as defined in POLYLINE DXF record.



(b) *Side-lines* used by converter to compute final corners.



(c) Final wide POLYLINE decomposed into three polygons.

Figure A.2- Conversion of a wide POLYLINE

A.7. INSERT

The groups in an INSERT are :

8	[layer name]	
2	[block name]	
10,20	[*x* and *y* coordinates of insertion point]	
38	[elevation , -optional]	
41	[X scale factor -optional]	
42	[Y scale factor -optional]	
43	[Z scale factor -optional]	
50	[rotation angle around *z* axis]	
66	[attributes follow flag]	{*not used*}
70, 71	[row and column count for array insertion]	{*not used*}
210, 220, 230	[vector along ECS *z* axis, -optional]	

For each INSERT entity, a UNIGRAFIX instance statement is generated and stored as a string in the current definition structure. In an INSERT record, the name under group code 2 specifies the block to be inserted . The *x* and *y* coordinates of the insertion point of the block are stated under group codes 10 and 20. The *z* coordinate, if non-zero, is stated under code 38. These coordinates correspond to a translation (UNIGRAFIX -t format). The scaling factors for the block (UNIGRAFIX -s format), if not equal to one, are stated under codes 41, 42, and 43. The rotation (UNIGRAFIX -r format) is always around the *z* axis and is recorded under group code 50. If ECS is used, the corresponding matrix is created. This matrix is stated in UNIGRAFIX format using the -M format.

Occasionally, scale factors are negative, resulting in mirrored blocks. Mirroring reverses the orientation of the faces. A negative scale factor is converted to a positive scaling followed by mirroring (UNIGRAFIX -m).

INSERT records can have attribute values if their corresponding block contains attribute definitions. These values are stated after the group ATTRIB. The converter ignores these attributes.

A.8. 3DLINE

The groups in a 3DLINE are :

8	[layer name]
10, 20, 30	[coordinates of first corner]
11, 21, 31	[coordinates of second corner]
39	[extrusion in *z* direction, -optional]
210, 220, 230	[vector along ECS *z* axis, -optional]

The 3DLINE and LINE entities are very similar. Their only difference is that a 3DLINE cannot have an elevation value. The conversion of a 3DLINE is exactly like the LINE entity. Despite the fact that the 3DLINE is not a planar entity, it can be extruded or constructed on an ECS. If the 3DLINE is extruded it will be extruded in the direction of the current UCS plane *x* axis.

A.9. 3DFACE

The groups in a 3DFACE are :

8	[layer name]	
10, 20, 30	[coordinates of first corner]	
11, 21, 31	[coordinates of second corner]	
12, 22, 32	[coordinates of third corner]	
13, 23, 33	[coordinates of fourth corner]	
70	[invisible edge bit-coded flag]	{*not used* }

The 3DFACE is the entity that we prefer to use for constructing polygons in our database. A 3DFACE is defined by its 4 corners and is directly converted to a UNIGRAFIX face. The third and fourth corners coincide if the 3DFACE is triangular.

Appendix B - The CS Building Database

The CS building database consists of 8 DXF files. There are seven floors; each floor is described in a file (csb-1.dxf through csb-7.dxf). The description of the roof (csb-r.dxf) resides in a separate file. Floors are constructed on the *xy* plane and most ceiling and floor polygons are parallel with this plane. Building elements in each floor are associated with a layer. All layer names used in the DXF file appear in the TABLE section of the file (Section 2.2), and are used to determine the color of the converted polygons. We have extracted them from the files and deleted the whole header section of the DXF files. Although layer names are to a large extent self-explanatory, a file (csb-layer.description) is provided that contains the layer descriptions provided by the architectures. To view the whole building, all files should be converted individually and then assembled.

Appendix C - Manual pages

Several programs and interactive tools are provided to prepare an AutoCAD model for the walkthrough display. The following manual pages provide a brief description of each program and its usage.

NAME

acad2ug — converter from AutoCAD DXF format to UNIGRAFIX

SYNOPSIS

acad2ug [-c <colorfile>] [-f1 <viewfile>] [-f2] [-f3] [-l <layerfile>] [-p] dxffile

DESCRIPTION

Acad2ug converts AutoCAD DXF files to UNIGRAFIX format preserving the hierarchy in the DXF file. Inputs to the converter are a DXF file (.dxf), an optional layer file, an optional color table file, and a set of optional arguments. Outputs are a hierarchical UNIGRAFIX (.ug) file; an error file (.errors extensions); an image file describing the plan view (.planv) and its corresponding UNIGRAFIX file (.planv.ug) if the option -p is used.

–c <colorfile>

This option allows the use of a color table to define the color values that are assigned to layers in the DXF file. The <colorfile> contains an rgb color table. In the absence of a color file the converter will use a default color table. The format of each line of the color file is:

[*Index*] [*Red*] [*Green*] [*Blue*]

-f1 <view file>

This option can be used to apply a first algorithm for fixing face orientations before the conversion results are written out. This algorithm uses a set of view points defined in the <viewfile> to render the scene and to determine the orientation of visible faces. The view points in the <viewfile> are specified as UNIGRAFIX instance statements.

-f2 This option can be used to apply a second algorithm that tries to recognize wall, ceiling and floor polygons in order to fix face orientations. It is assumed that the building is constructed on the *xy* plane and ceilings and floors are parallel to this plane.

-f3 This option is the same as -f2. In addition it will look for the string "CL" and "FL" in layer names to identify ceiling and floor polygons. This option should be used instead of -f2 if ceilings and floors are modeled without thickness, e.g. when the upper and lower surfaces of a floor slab appear in different files.

–l <layerfile>

If conversion of only specific layers in the AutoCAD file is desired this option should be used with a <layerfile>. The <layerfile> consists of the name of all layers in the DXF file along with a tag indicating whether the layer is on(Y) or off(N) and an index into a color table. The format of each line of the layer file is:

[*name of layer*] [*Y*(on) or *N*(off)] [*color index*]

-p This option indicates that a plan view file should be generated. The plan view file consists of pixel color values corresponding to an orthogonal projection of the floor on the *xy* plane. The accompanying layer file should specify the layers that pertain to plan views. In addition to the plan view file (.planv), a corresponding UNIGRAFIX file (.planv.ug) is generated.

EXAMPLE

acad2ug -l *building.layer building*
will convert the file *building.dxf* using the layers in file *building.layer* and outputs files *building.ug* and *building.errors*.

FILES

~delnaz/csb/src/acad ~delnaz/ug/ug3lib

SEE ALSO

acadlayers ugitools ucbwalk

DIAGNOSTICS

Several AutoCAD entities such as POINT, VERTEX, TEXT and POLYLINE bulge factors are not implemented. Planar non-extruded entities are converted to UNIGRAFIX face statements only if they are closed.

AUTHOR

Delnaz Khorramabadi

NAME

acadlayers — extracts the layers used in AutoCAD DXF files.

SYNOPSIS

acadlayers < dxffile > layerfile

DESCRIPTION

Acadlayers reads an AutoCAD DXF file on stdin and outputs the list of layers used in the file to stdout. The format of the outputfile is:

[*name of layer*] [*Y*] [*color index*]

This is the basis from which one starts to create the customized layer files used with the options -p or -l of **acad2ug**.

EXAMPLE

acadlayers < *building.dxf* > *building.layers*

FILES

~delnaz/csb/src/acad

SEE ALSO

acad2ug ugitools ucbwalk

AUTHOR

Delnaz Khorramabadi

NAME

ugcleanup — invokes several programs to clean a UNIGRAFIX file.

SYNOPSIS

ugcleanup [**-a**] [**-c**] [**-i**] [**-m**] [**-n <epsilon>**] [**-p <priorityfile>**] < inputfile > outputfile

DESCRIPTION

Ugcleanup is a filter to clean hierarchical UNIGRAFIX files. It reads from stdin and calls "ugvlmerge" "ugisect" and "ugcopl" as requested to eliminate T vertices, incorrect intersections and coplanar overlapping faces in each definition. The output is written to stdout.

-a Use area criteria to discard one of the overlapping coplanar polygons.

-c Calls **ugcopl** to find coplanar overlapping faces. If the overlapping polygons have the same color, one is discarded.

-i Calls **ugisect** to eliminate incorrect intersections.

-m Calls **ugvlmerge** to eliminate T vertices.

-n <epsilon>
Calls **vmerge** to merge vertices that lie within epsilon of each other.

-p <priorityfile>
Use a priority file to discard one of the overlapping coplanar polygons. The priority file contains pairs of color names used in the UNIGRAFIX file. The first member of a pair has priority over the second one. If the file is not specified, a list of overlapping colors will be generated on stderr.

FILES

~delnaz/csb/src/cleanup ~delnaz/ug/ug3lib

SEE ALSO

ugisect ugcopl ugvlmerge ugclean ugitools

DIAGNOSTICS

The slits created by **ugisect** are eliminated. **Ugcopl** will not have satisfactory results when used without **ugvlmerge**.

AUTHOR

Delnaz Khorramabadi

NAME

ugitools — collection of interactive tools to view and edit UNIGRAFIX files.

SYNOPSIS

ugitools [-(r/h)] [-v (viewfile)] < inputfile

DESCRIPTION

ugitools is a tool-kit for UNIGRAFIX files designed for a Silicon Graphics IRIS 4D. It reads hierarchical UNIGRAFIX files from stdin and creates a scene window for viewing and editing definitions. A panel with five push buttons is provided to invoke the following tools:

ugview:
for modifying viewing, rendering and picking parameters; stepping through definitions and saving files.

ugcolor:
for editing color values and assigning colors to instances and faces.

ugedit:
for editing the geometry of definitions.

ugorient:"
for fixing face orientations.

ugplace:
for creating instances and editing their transformations.

–r/h used for **ugcolor** to determine the color format (rgb or hls) for the output file (default is rgb).

–v <viewfile>
used for **ugorient** to specify the file containing view points.

FILES

~delnaz/ug/ugitools ~delnaz/panel ~delnaz/ug/ug3lib

SEE ALSO

ugview ugcolor ugedit ugorient ugplace

DIAGNOSTICS

Only global colors that are defined outside definitions can be edited.

AUTHOR

Delnaz Khorramabadi

NAME

ugview — interactive tool to view UNIGRAFIX files

SYNOPSIS

ugview < inputfile

DESCRIPTION

Ugview is an interactive viewer for UNIGRAFIX files designed for a Silicon Graphics IRIS 4D. It reads hierarchical UNIGRAFIX files from stdin and creates a scene window for displaying UNIGRAFIX definitions. Mouse buttons are used directly on the scene window to pick items and modify viewing transformations. A panel with pushbuttons and a color selection chart is provided that allows manipulation of viewing, rendering and picking parameters. The panel also provides for selective rendering of groups of objects with similar colors.

Mouse Actions:

Picking/Unpicking: To enable/disable picking of instances, faces, wires, edges, and vertices push the corresponding button on picking options. To pick an item move the mouse on top of the item and click with the left mouse button while holding the control key. The picked item will be high-lighted and added to the selection list. To remove the item from the selection list, hold the control key and click with the middle mouse button. To distinguish picked instances from other items, only their contour is high-lighted.

Scaling: To scale the scene hold down the left mouse button and drag it towards right (scale up) or left (scale down).

Rotating: To rotate the scene hold down the middle mouse button and drag it towards left/right (rotation about Y axis) or up/down (rotation about X axis).

Translating: To translate the scene hold down the right mouse button and drag towards the desired direction.

Panel actions:

By clicking at buttons in the panel, you can change rendering, viewing, and picking parameters, advance to the next definition, and render parts of a definition. Each button has an associated keyboard key that will perform the same action.

Rendering Options: By pushing one or more of the buttons in this group you can allow/disallow rendering of instances, faces, backfaces(black), wires, edges vertices and the display of the definition's coordinate axes.

Picking Options: By pushing one or more of the buttons in this group you can enable/disable picking of instances, faces, wires, edges and vertices. The "clear_s" button will remove all the picked items from the selection list.

Viewing Options: By pushing one of the first four buttons in this group you can have four different views of the object: front, top, side, and perspective. By pushing the "spin" button, the object will constantly rotate about its z axis in addition to other mouse transformations. The scene can be reset by pushing the "reset" button.

Color Chart: This is a color selection chart that displays the names of all the colors that are globally defined in the file. The chart can be scrolled by a slider on its right hand side border. Colors can be selected by clicking at a name on this chart or selecting an item with the desired color on the scene window. The selected color will be high-lighted in the chart. The items with the selected color can be removed from the display or added back to it by pushing the "on/off" button. The two buttons "all_on" and "all_off" will remove all colored items from display or add all back to the display.

Advancing to the next definition: The scene always starts by displaying the top of the UNIGRAFIX hierarchy. You can step through and examine all definitions in the order of their appearance in the file by pushing this button.

Saving the results: To save the input file with all modifications press the "Save" button. This will

create the file output<number>, where number indicates the number of times an output has been written during the current session. The items that are removed from display by turning their colors off will not be written to the file.

Plan view generation: To generate a plan view of the file push the "Planview" button. This will create the file output<number>.planv, where number indicates the number of times a plan view output has been written during this session. This file is a binary file containing the image produced by rendering an orthogonal view of the flattened file. The items that are removed from display by turning theirs colors off will not be written to the file.

Exit: Exits the program.

FILES

~delnaz/ug/ugview ~delnaz/panel ~delnaz/ug/ug3lib

SEE ALSO

ugitools ugcolor ugedit ugorient ugplace

DIAGNOSTICS

Only global colors that are defined outside definitions can be edited.

AUTHOR

Delnaz Khorramabadi

NAME

ugcolor — interactive tool for color editing and assignment in UNIGRAFIX files.

SYNOPSIS

ugcolor [**-(r/h)**] < inputfile

DESCRIPTION

Ugcolor is an interactive viewer and color editor for UNIGRAFIX files designed for a Silicon Graphics IRIS 4D. It reads hierarchical UNIGRAFIX files from stdin and uses **ugview** to examine each definition and pick items for editing. Ugcolor provides a color pallete with red, green, and blue sliders to edit the value of colors defined in the input UNIGRAFIX file. Colors can be assigned interactively to individual instances, faces and wires.

–r/h determines the color format (rgb or hls) for the output file (default is rgb).

Editing Color Values:

To edit the value of a color, select it by clicking directly on the color chart or by picking an item with the desired color in the scene window. In either case the name of the selected color will be high-lighted on the color chart and its value will be painted on the color pallete. Move the r, g, and b sliders to edit the color. When the desired value is obtained push the "Color_ok" button. The results will take effect immediately and all the items with the selected color will be displayed with their new value.

Color Assignment:

To assign a new color definition to one or more instances, faces, or wires select them in the scene window (for selection refer to ugview). The selected items will be high-lighted in the scene. Use the slider on the border of the color chart to scroll the chart to find the name of the new color that should be assigned to the selected items. Select the color and push the "color_change" button. The selected items will be assigned the new color and the results can be viewed on the scene window.

FILES

~delnaz/ug/ugcolor ~delnaz/panel ~delnaz/ug/ug3lib

SEE ALSO

ugitools ugview ugedit ugorient ugplace

DIAGNOSTICS

Only global colors that are defined outside definitions can be edited.

AUTHOR

Delnaz Khorramabadi

NAME

ugedit — interactive tool for editing geometry of definitions in UNIGRAFIX files.

SYNOPSIS

ugedit < inputfile

DESCRIPTION

Ugedit is an interactive viewer and geometry editor for UNIGRAFIX files designed for a Silicon Graphics IRIS 4D. It reads hierarchical UNIGRAFIX files from stdin and uses **ugview** to examine each definition and pick items for editing. Ugedit provides a panel with several push buttons and typein windows to modify the geometry of each definition.

Adding a vertex:

Type in the coordinates of the new vertex in the x, y, and z coordinate typein windows and push the "New_V" button.

Deleting vertices:

Select some subset of vertices in the scene window (refer to ugview for selection), and push the "Delete" button. All edges attached to these vertices are deleted first, then all contours that end up with less than three edges are deleted, then all faces that end up with no contours are deleted, and finally the selected vertices are deleted.

Moving a vertex:

Clear selections and select the vertex in the scene window (refer to ugview for selection). The coordinates of the vertex will appear on x, y, and z typein windows. Edit these values and push the "Move_V" button. The coordinates of the vertex will be updated by the entered values.

Merging vertices:

Select the vertices in the scene window (refer to ugview for selection), and push the "Merge_V" button. All the selected vertices will be deleted and a new vertex will be created at the position of the first selected vertex that has a topology that is the union of all the selected vertices.

Adding a face or a wire:

Select the vertices of the new face or wire in the scene window (refer to ugview for selection), and push the "New_F" or the "New_W" buttons. The order in which the vertices are arranged in the new face or wire corresponds to the order in which they were selected. Default names are chosen for new faces and wires. If a color is selected in the color chart, it will determine the color of the newly created face or wire. Otherwise the newly created faces or wire will have no color.

Making faces double-sided:

Select the faces that should be double-sided in the scene window (refer to ugview for selection), and push the "Double_F" button. All selected faces will be duplicated with the order of their vertices reversed.

Deleting faces and wires:

Select the faces and wires that should be deleted in the scene window (refer to ugview for selection), and push the "Delete" button.

Extruding faces and wires:

Select the faces and wires that should be extruded in the scene window (refer to ugview for selection), enter the extrusion amount in the extrusion typein window and and push the "Extrude" button. Wires will be extruded along the z axis and faces will be extruded along their normal.

Subdividing faces and edges:

Select the faces and edges that should be subdivided in the scene window (refer to ugview for selection), and push the "Subdivide" button. A new vertex will be created at the centroid of each selected face or the midpoint of each edge, and will be connected to all of the vertices of the face or edge. The original faces and edges will be deleted.

Collapsing faces and edges:

Select the faces and edges that should be collapsed in the scene window (refer to ugview for

selection), and push the "Collapse" button. A new vertex will be created at the centroid of each selected face and all the boundary vertices of each face will collapse into this vertex. For an edge, a new vertex is created at its midpoint and the two endpoints of the original edge are merged into this new vertex.

FILES

~delnaz/ug/ugedit ~delnaz/panel ~delnaz/ug/ug3lib

SEE ALSO

ugitools ugview ugorient ugcolor ugplace

AUTHOR

Tom Funkhauser, Delnaz Khorramabadi

NAME

ugorient — interactive tool for fixing face orientations in UNIGRAFIX files.

SYNOPSIS

ugorient [**–v <viewfile>**] < inputfile

DESCRIPTION

Ugorient is an interactive viewer and face orientation editor for UNIGRAFIX files designed for a Silicon Graphics IRIS 4D. It reads hierarchical UNIGRAFIX files from stdin and uses **ugview** to examine each definition and pick faces for editing. Ugorient provides a panel with several push buttons to apply different algorithms for fixing the orientation of one or more faces.

–v <viewfile>

specifies a file containing a series of view points to be used for fixing face orientations in building models.

Flipping orientation of faces:

To flip the orientation of one or more faces, select them on the scene window and push the "Flip-Face" button. To fix faces in a particular definition, the scene should be advanced to display the definition.

Propagating orientation of faces:

To propagate the orientation of one or more faces to neighboring faces through their shared edges, select them on the scene window and push the "P_edge" button. To fix faces in a particular definition, the scene should be advanced to display the definition.

Propagating orientation in a plane:

To propagate the orientation of one more faces to all other faces that lie in the same plane, select them on the scene window and push the "P_plane" button. To fix faces in a particular definition, the scene should be advanced to display the definition.

Fixing orientation of faces in buildings:

Two algorithms are provided to fix face orientations in buildings:

The first one uses building properties, such as wall thicknesses, and tries to identify the two sides of walls, ceilings and floors. It operates on the faces in each definition and has produces the best results when a definition contains the complete structural elements of a building. To apply this algorithm on the faces in a definition, advance the display to the definition and push the "Fix1_B" button. To apply this algorithm in steps and see the results of each step, keep pushing the "Fix1_Bstage" button. The criteria used at each step and statistics on the number of faces that were fixed will be printed.

The second algorithm uses a set of view points to render the building and to fix the orientation of visible polygons. It operates on all definitions in the file at the same time by rendering the flattened file. To apply this algorithm, the view points should be provided in a file that appears after the -v option. This file is a UNIGRAFIX file containing instances of the definition VIEWPNT; the translation values in each instance statement determine the position of each view point. To invoke this algorithm, advance the display to the top of the UNIGRAFIX hierarchy and push the "Fix2_B" button.

Viewing the results:

To view the results on the scene window, press the "Show_results" button. The faces that were fixed as a result of the last action, will be high-lighted in the scene; errors will be shown in the color red.

FILES

~delnaz/ug/ugorient ~delnaz/panel ~delnaz/ug/ug3lib

SEE ALSO

ugitools ugview ugedit ugcolor ugplace

DIAGNOSTICS

The algorithms to fix buildings are designed for building models with walls of finite thickness that are constructed on the *xy* plane.

AUTHOR

Delnaz Khorramabadi

NAME

ugplace — interactive tool for placing definitions in UNIGRAFIX files.

SYNOPSIS

ugplace < inputfile

DESCRIPTION

UgPlace is an interactive viewer and placement tool for UNIGRAFIX files designed for a Silicon Graphics IRIS 4D. It reads hierarchical UNIGRAFIX files from stdin and uses **ugview** to examine each definition and to pick items for placement. Ugplace provides a panel for the user to control the placement of definitions consisting of a chart for selecting definition names; a chart for selecting a transformation from the current instance or adding a new one to it; sliders, dials, buttons and a typein window for entering or modifying the value of the current transformation; and pushbuttons to copy, create and delete instances.

Adding a new transformation to an instance:

Clear the selection list and select the instance in the scene window (for selection refer to ugview). The transformation list of the instance will appear in the transformation chart followed by the string "New transformation". Select the string "New transformation" from the transformation chart. To specify the new transformation, either enter it in the typein window, or use one of the sliders (scale, translate) , dials (rotate) or button (mirror).

Editing transformations of an instance:

Clear the selection list and select the instance in the scene window (for selection refer to ugview). The transformation list of the instance will appear in the transformation chart. To edit one of the transformations pick it on the transformation chart. The selected transformation will be high-lighted and will appear in the typein window. You can either edit it by modifying its equivalent slider/ dial/button or by editing its value in the typein window. As you start modifying the value of a slider (translation/scaling) or a dial(rotation), the picked instance will reflect the modification on the scene. Once the slider or dial is released the change will become permanent.

Editing transformations of a group of instances:

Clear the selection list and select the instances one by one in the scene window. (for selection refer to ugview). The transformation chart will display only the string "New transformation" as soon as the selection list contains more than one instance. Determine this new transformation as explained above. The new transformation will be appended to the transformation lists of all selected instances.

Copying an existing instance:

Clear the selection list and select the instance in the scene window (for selection refer to ugview). Push the "Copy" button. A copy of the instance is added to the current definition; the selection list is cleared, and the copied instance is placed in the list. Note that the copied instance lies exactly on the original one and can't be distinguished until its transformations are modified.

Copying several instances:

Clear the selection list and select the instances one by one in the scene window. (for selection refer to ugview). Press the "Copy" button. A copy of all instances in the selection list are made and added to the current definition; the selection list is cleared and the copied instances are placed in the list. Note that the copied instances lie exactly on their originals and can't be distinguished until their transformations are changed.

Creating a new instance:

Pick a definition either by picking an instance of it in the scene window or by selecting its name in the definition chart. In either case the name of the selected definition will be high-lighted in the definition chart. Press the "New" button. An instance of this definition will be made and added to the current definition; the selection list is cleared and the new instance is placed in the list. The new instance has initial scaling, rotation, and translation values such that it will appear on the top right hand side of the window. Modify these values as needed.

Deleting instances:

Clear the selection list and select the instances one by one in the scene window. (for selection refer to ugview). Push the "Delete" button. The selected instances will be deleted from the current definition.

FILES

~delnaz/ug/ugplace ~delnaz/panel ~delnaz/ug/ug3lib

SEE ALSO

ugitools ugview ugedit ugorient ugcolor

DIAGNOSTICS

Transformations in matrix form (-M3 -M4) and array statements are not handled.

AUTHOR

Delnaz Khorramabadi

NAME

ucbwalk — interactive walkthrough program for buildings

SYNOPSIS

ucbwalk [**-a <x y z>**] [**-d <datafile>**] [**-f <x y z>**] [**-h <helpfile>**] [**-l <floorfile>**]

DESCRIPTION

Ucbwalk is a display program that takes a database (created from UNIGRAFIX files) containing the description of a building and lets the user navigate through the building and explore it interactively. It uses the mouse buttons and the arrow keys to control motion in the scene window. A panel with several push buttons is provided to modify rendering, lighting, and motion parameters. A plan view of the current floor with the current position and viewing direction of the user is displayed at all times. Information on usage can be displayed by pushing the help button.

-a <x y z>:
Initial viewing direction of the viewer.

-d <data file>:
This file contains a description of the building at various levels of detail and its visibility information in a tree structure.

-f <x y z>:
Initial position of the viewer.

-h <help file>:
This file contains information on how to use the walkthrough. It appears on the walkthrough scene when the "help" button is pressed. The default file is "ucbwalk.help".

-l <floor file>:
The floor file stores information about each floor in the building. It contains the number of floors in the building, their names (to be displayed during walkthrough), their elevation and corresponding plan view file. Its format is:

<number of floors> <initial floor>
<name of floor i> <elevation of floor i> <plan view file for floor i>

EXAMPLE

ucbwalk -a 0. 1. 0. -f 1000. 1000. 3000. -h *ucbwalk.help* -l *csb.floorfile* -t *csb.database*

FILES

~delnaz/csb/src/walk

SEE ALSO

acad2ug wkcreate wkadd wksplit wklist

AUTHOR

Delnaz Khorramabadi