Interactive Object Displacement in Building Walkthrough Models

by Thurman A. Brown

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Carlo H. Séquin

Professor Carlo H. Séquin
Research Advisor

December 18, 1992

(Date)

* * * * * * *

Lawrence A. Rowe

Professor Lawrence A. Rowe
Second Reader

18 December 1992

(Date)

Abstract

In the context of a Building Walkthrough Program, we investigate methods for interactively placing and rearranging furniture within the building model while in an interactive real-time Walkthrough session. We have created a simple 3D editor that allows the user to move objects, while observing some object-dependent constraints. Different objects may have different alignment properties. For example, desks and chairs would be given the property that they will remain aligned with the floor polygons, while pictures and white boards might be confined to be coplanar with the vertical walls of the building. When objects with such alignment properties are placed or moved in the model, the program searches for a nearby suitable face, and if an object comes within a certain distance of such a face, the object is snapped to that face. These alignment properties can be used by our interactive editor as well as by programs that populate the interior of a building model procedurally.

Of course, within the given alignment constraints, the user is allowed to rotate, scale, and duplicated objects (*operations well-known from 2D graphics editors*). A good graphical user interface with iconic handles makes such operations easy and unambiguous in a 3D model with only the 2D mouse pad as an input device. This editor should make it much easier for the developers and users of a building model to place furniture and other objects into natural positions without the need to change views or to zoom into the space between an object and its supporting surface. By reducing the tedium of placing thousands of objects inside a large and complicated building model, we will be able to populate such models to a degree that makes them look realistic and that makes for exciting Walkthrough experiences.

Contents

1 Introduction

The goal of this project is to create an environment in which a user can easily populate a large building with objects in a variety of complicated arrangements. Reducing the tedium of this task will allow the user to create a more realistic Walkthrough environment.

Before creating this tool, the process of placing furniture in building models was very time consuming. The user was required to do all of the positioning of objects in either an AutoCAD or UNIGRAFIX file. Verification of object position was done in the Walkthrough phase, but any required positioning changes had to be done in the original AutoCAD or UNIGRAFIX file.

Our editor program allows the user to perform all object positioning and verification within the Walkthrough phase. The user is not required to use AutoCAD or UNIGRAFIX, other than to initially define the objects and position them in the scene. The user can do the final placement of the objects in the database environment, while interactively verifying their positioning. The user is given additional positioning help by placing constraints on objects. With the earlier system, the user had the problem of aligning furniture exactly on the floor or of placing white boards exactly on the walls. Needless to say, we ended up with many floating white boards and desks sunken into the floor. These problems are removed with the use of the editor. The user can simply walk around in the building, moving some objects and turning others, without worrying about leaving objects floating, thus creating a more natural looking object population within the building.

Another useful application of the editor is as a verification program. With the use of object properties, we have created a verification program that we can run on our building models. The program checks every object in the database. If the program finds an object that does not conform to one of its property constraints, the object is translated to its correct position. Thus, we can run this program on our building model, and it will correct many of the modeling problems such as floating picture boards and sunken chairs.

1.1 Problem

In the 3D world, most of the existing editors seem to fall into one of two realms. In one realm are the 3D editors or CAD tools, such as AutoCAD. CAD tools are designed to help the user in creating a particular type of object. For instance, AutoCAD is designed to help the user model buildings or other objects with mostly axial faces. It is not a general modeler, so it would be difficult to model objects such as plants or animals. Other CAD devices are designed to model electronic circuits. It is impossible to model a building using a circuit design CAD tool. In the other realm of the 3D world, there are editors and tools that focus on manipulating or altering a particular object. There has been work at Brown University to implement 3D widgets [4]. The 3D widgets they created, give the user easy access to a set

of transformation operations that are used to manipulate individual objects. Other environments help the user to create interesting 3D objects procedurally. At UC Berkeley many tools have been created to generate or modify objects described in the UNIGRAFIX language [3], for example, *animator* allows the user to display UNIGRAFIX objects and manipulate the vertices, edges, and faces of that object, or *UGtess* allows the user to tessellate an object in a variety of ways. However, no tools exist that work on scene descriptions with large groups of diverse objects.

For the Building Walkthrough Project, we need a 3D editor that is designed to help the user with the placement of objects within a scene and to specify how these objects interact with each other. We need an easy way to compose interesting and realistic scenes. At present, the process of placing objects in the database consists of a combination of UNIGRAFIX, CAD, and database operations. Desks, tables, chairs, and other objects are modeled in UNIGRAFIX or by other editing sources. Then a CAD tool is used to populate one of the offices in the building by adding various objects. The resulting CAD file is then converted into UNIGRAFIX with help of an AutoCAD-to-UNIGRAFIX converter [7]. The database file is created by running various database routines on the UNIGRAFIX file. The database routine, *wkadd*, creates the necessary data structures to represent the walls and other objects in the scene [1]. By a semi-batch process of applying transformations and rotations to an entire UNIGRAFIX office description, the remaining offices are populated. Each office is then inspected by running the Walkthrough program and visually verifying the position of each object in the building. When positioning problems arise, due to irregularities in the building, such as support beams in some of the corners, individual office files must be corrected either by changing the UNIGRAFIX text file or by altering the CAD descriptions of the office and re-applying the conversion process. Then a new database file must be created and the building must be re-inspected.

This batch method of populating the building results in many objects being positioned incorrectly and too much regularity in the building. Most of the offices in the building have the same amount of furniture, all placed in the same manner. This regularity diminishes the realism of our models, because, typically, no two people organize things in the same way. Under the existing system, the only way to solve this regularity problem is to convert each office description to its UNIGRAFIX or CAD representation and separately edit each office file. What we need is an easier way to populate the building, and once the building is populated, we need to be able to reorganize the objects in any office without leaving the database representation of that office. With this we will better be able to represent the "natural looking" disorder of everyday life.

1.2 Solution

To solve this problem, we have created an interactive 3D object displacement program. The goal of the program is to allow a user to manipulate any objects encountered in the Walkthrough and to have the changes updated into the database in real time. The user can define alignment constraints on types of objects in the

database. Then these constraints can be used to help with the placement of the objects. For example, a desk or a chair could be constrained to be aligned with a floor polygon. When moving an object with such a constraint, the user does not have to worry about accidently leaving a desk floating in mid air. White boards and pictures can be constrained to remain coplanar with wall polygons, preventing them from floating away from a wall. Objects with constraints snap to other suitable objects when they are within a certain distance of that object, similar to Eric Bier's Snap-dragging paradigm [5]. The user is also allowed, within the given alignment constraints, to rotate, scale, or duplicated objects, operations well-known from 2D graphics editors.

It would be nice if the user had the power to manipulate the walls and floors of the building as well. For instance, the user might want to see what a larger or smaller room might look like or what would happen is he/she removed one of the walls in an office. But to remove walls and floors in the Walkthrough model requires many calculations. Visibility and subdivision precomputations have to be re-calculated. These calculations are done in batch mode, before the creation of the database, to allow us to display the Walkthrough model in real time [2]. For large building models, these pre-calculations take many hours. We would thus need to find a way to perform these pre-calculations incrementally. This problem is beyond the scope of this work. Thus, currently the editor does not allow the user to perform transformations on the wall and floor polygons of the model. (see Transformations section for details on constraints).

With our editor, the user can easily place and add objects in a scene. Each user can arrange the objects in a way that feels comfortable to him/her. Once many users are allowed to modify different rooms to their preference, the building model will be a step closer to reaching the desired realism.

1.3 Overview of Report

The remainder of the report discusses various features of the editor and their implementation. The following section is a tutorial on how to use the editor. It explains the editor both from the user's and from the system operator's point of view. Then there is a discussion on the integration of the editor's user interface with the rest of the Walkthrough program. Next, the paper goes on to discuss the issues involved with selecting objects in the Walkthrough environment and displaying them while they are being manipulated interactively. Section 4 explains object properties. It begins by discussing the properties that are available in the current version and discusses the effects that these properties have on the placement of objects. It details what is required to define these properties both from the user's and from the system operator's perspective. Section 8 discusses how the editor handles object transformations and constraints in the interactive mode and in the batch mode. The final two sections discuss some suggested extensions to the editor system and the conclusions reached from this project.

2 User Interface

This section gives a description of how to use the editor and what options are available to the user. The first sub-section is designed for the novice user. It is a tutorial on how to access the features of the editor and some examples of how to use the editor. The second sub-section is a discussion of the decisions made in creating the user interface for the editor.

2.1 Tutorial Description from End-User's Point of View

For this section, it is assumed that the user is running the Walkthrough program with the editor operations properly installed. It is also assumed that the current database model contains at least one object instance of any type of object that might be placed by the user.

2.1.1 Selecting an Object

Holding down the shift key puts the system into editing mode and activates dynamic highlighting. With the shift key pressed, the object that is currently under the cursor position is highlighted by displaying its bounding box in white. Pressing the left mouse button while holding down the shift key will perform static selection. The object that is currently highlighted by dynamic selection (a white bounding box) will now be highlighted by a black bounding box. If the user releases the mouse button, while continuing to press the shift key, then the white bounding box will continue to dynamically highlight the object under the current cursor position while the black bounding box will remain with the statically selected object. *(NOTE: For the remainder of this section when I refer to pressing one of the mouse buttons I am implying that the shift key should also be pressed, unless otherwise stated.)* To statically select a different object, the cursor is moved over the new object, using dynamic highlighting if desired, and the left mouse button is pressed. This will unselect the first object and select the new object, represented by moving the black bounding box from the previously selected object and to the newly selected object.

To select multiple objects, the *ALT* key is pressed during the selection. If no objects are currently selected, the multiple select has the same effect as the normal select. It will select the object that is currently being dynamically selected. If other objects were previously selected, a new object can be selected without unselecting these objects by using the *ALT* key. This is represented by displaying a black bounding box around each of the selected objects. *(Note: Transformations done with multiple selection are done with respect to the last object selected (see Transformation section for more details).* A previously selected object is unselected by re-selecting the object with the multiple select function. The static select function (without the *ALT* key) overrides multiple selection. If the user has multiple objects selected, but wants to select only the new object, the new object is selected with

static selection. This results in the new object being selected and all other objects being unselected.

2.1.2 Transformation Modes

In the editor, there is a general transformations mode as well as three constrained transformation modes. In the general mode, the user can easily translate or rotate an object. In the constrained modes, the user is given interaction handles that allow him/her to perform more specialized transformations (including scaling). The editor starts in the general transformation mode. In this mode, the user can use the left mouse to translate an object, and the middle mouse to rotate the object. The right mouse button is used to change transformation modes. When the user presses the right mouse button for the first time, the program changes into the constrained translation mode. In this mode, a triad of translation handles *(see transformation section)* are displayed with the selected object. Pressing the right button again, changes the program into the constrained rotation mode. In this mode, the great circles and handles for rotation are displayed with the object. Pressing the right mouse button a third time changes the program into the constrained scaling mode, provided that scaling has been enabled in the dialog window. In this mode, the eight scaling handles are displayed along with the object. Each mode displays the bounding box in a different color to help the user distinguish the modes. Pressing the button a fourth time restores the program to the default transformation mode.

2.1.3 Performing Transformations

Once an object is selected, the left and middle mouse buttons are used to interact with the object. When the user selects an object with the left mouse button, it is highlighted by a black bounding box. By selecting the object again with the left mouse button, the user grabs the object for a translation within its defined constraints. By grabbing the object with the middle mouse the user can rotate the object, within any object placement constraints, based on the crystal ball paradigm used in many interactive viewers.

If the user desires to use one of the more specialized transformations, he/she presses the right mouse button to enter the desired mode. In the translation mode, the object is displayed with the triad of handles used for translation. To move the object in the XY-plane, the user positions the cursor over the handle that represents movement in the XY-plane. He/she then presses and holds the left mouse button to select this handle. As long as the user holds down the left button the object will be dragged around in the XY-plane. When the user releases the left button, the object will be dropped in its new position. Similar actions are required to interact with the scaling or rotation handles.

2.1.4 Menu Operations

The user is given additional editing options in a dialog menu. The dialog menu consist of a number of option windows and buttons that give the user some control over the currently allowable options and the current mode of the editor. The following parameters can be specified (Figure 1 and 2):

- Current Viewing mode. This option allows the user to toggle in and out of birds-eye-view. The birds-eye-view transformation is controlled by a combination of the control key and the mouse buttons. Control key and left button are used to scale, control key and middle button are used to rotate, and control key and right button are used to translate. For the birds-eye-view operations, the shift key must be released.

- Object selection mode. This allows the user to switch between performing object selection by searching for face intersections with the ray created from the cursor position and performing object selection by searching just for bounding box intersections. The later mode is advised only when speed of the face intersection mode becomes intolerable slow.

- How to display the selected object (or objects) during interactive manipulation. The choices are: "transparent", "wire frame", "normal", or "bounding box only". Simply selecting the desired option, propagates the changes to the display.

- How to handle object property constraints. The user has the choice of: "No constraints", in which the constraints have no effect on the motion of the object; "Near Constraints", in which an object only snaps to a suitable face if it is within a certain distance of that face, i.e. half the distance of a bounding box dimension; and "ALL Constraint", in which an object can snap to any face in the same room, regardless of its distance from that face.

- How to draw hint lines. These are the axial lines drawn from the interaction point to make positioning in 3D easier. Hint lines are only used in the translation mode. The user has the option to not draw the lines at all, to draw the lines to the extent of the object's bounding box, or to draw the lines to the extent of the database.

- Enable Scaling. This options enables or disables scaling to protect the user from unintentionally doing hard-to-correct damage to the database. When scaling is enabled, the scaling mode will appear as one of the transformation modes accessed by using the right mouse button.

- The Transformation button. Pressing this button pops up a dialog menu that is used to perform exact transformation of objects. There are three buttons: a translations button, a rotation button, and a scaling button. Under each button there are three input fields that allow the user to enter the desired X, Y, and Z values for the respective transformations. In order to perform an exact

transformation, the user selects the desired object or objects, enters the X, Y and Z values for the corresponding transformation, and then presses the button. For example, if the user wants to move a picture exactly 5 inches up, he/she would first select the picture. Then he/she would press the transformation button in the main dialog menu. Next the user would enter 5 in the Z field above the translation button, making sure the the X and Y values are set to 0. Then the user presses the translation button and the picture is moved up 5 inches in the Z direction.

- The UNDO button. The UNDO button allows the user to UNDO the last transformation.
- The COPY button. The COPY button produces a duplicate of all objects that are currently selected. *(Note: The duplicated object is not added to the database until the original object is moved. This prevents the introduction of multiple objects in the exact same position.)*

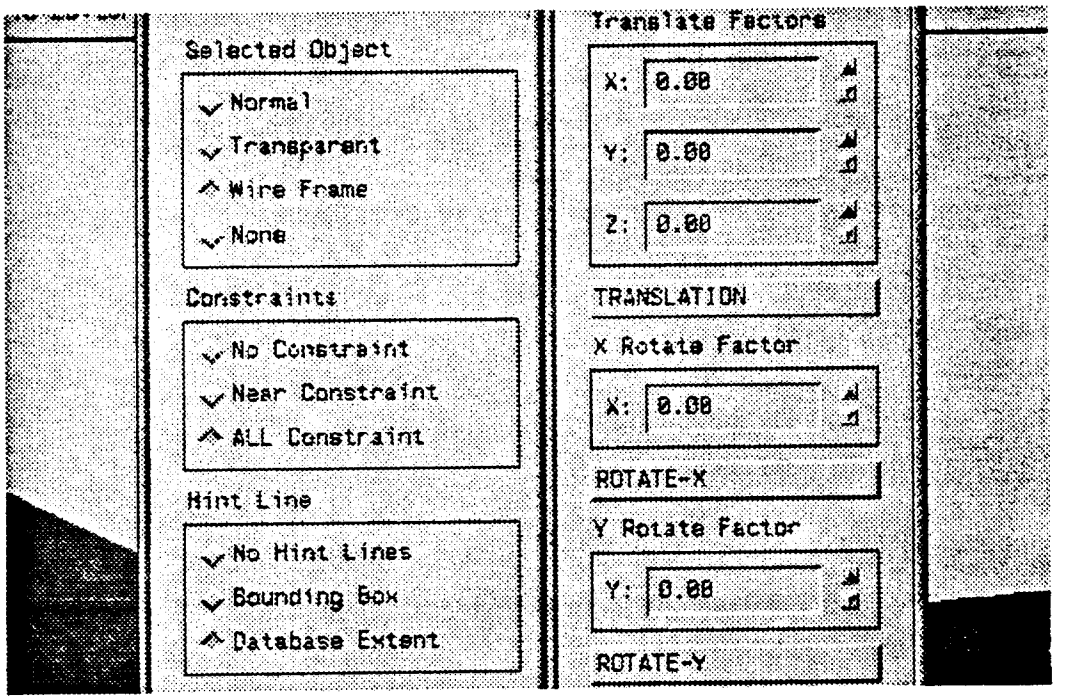

Figure 1: Top half of main dialog menu and exact transformation menu.

2.2 Discussion of Choices Made

When designing the editor for the Walkthrough, our goal was to create a simple yet powerful user interface to casually arrange the objects within the building without creating lots of physically non-sensible situations such as objects floating in mid air. The two main concerns of the interface were to make a few operations very simple and to intergrate all commands harmoniously into the large Walkthrough

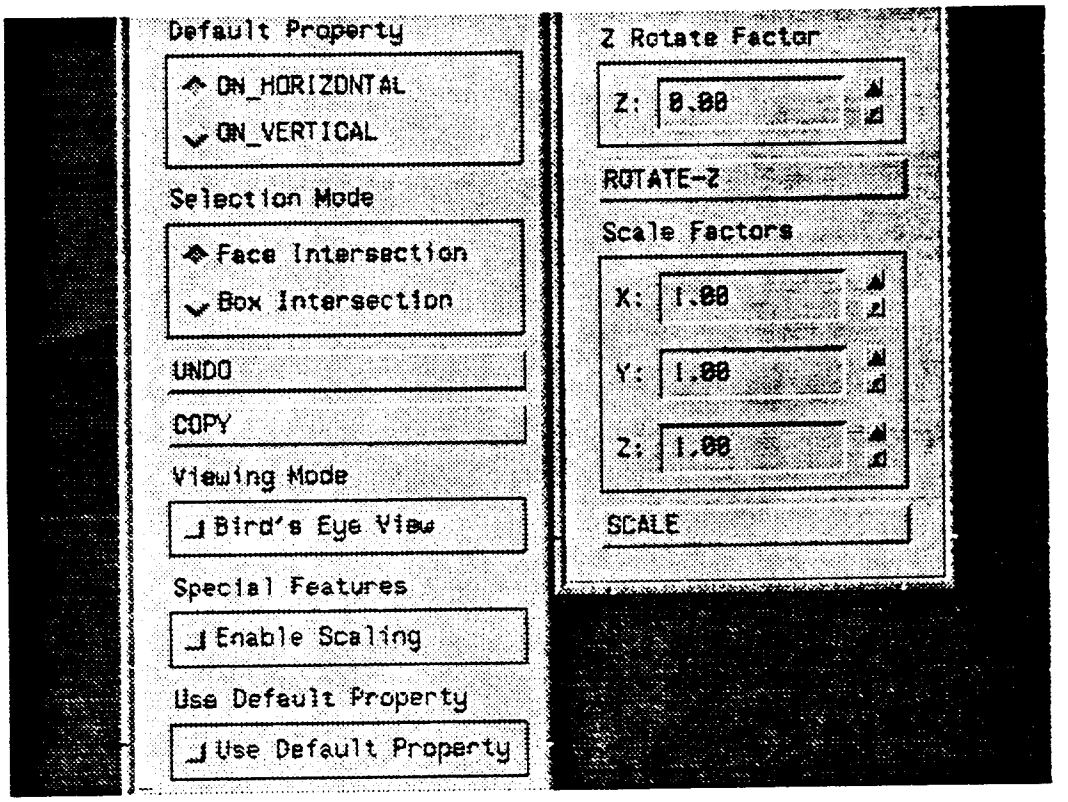

Figure 2: Bottom half of main dialog menu and exact transformation menu.

system. The following sections describes some of the decisions made about the user interface.

2.2.1 A Discussion of Modality

The two extremes of a modal user interface are a "completely modal" interface and a "non-modal" interface. In a completely modal interface, the user only has access to the operations related to the current mode. This type of interface is usually the simplest to use but limits the power of the system. In a non-modal interface, the user has access to all operations of the system at all times. This type of interface is generally more powerful, i.e., it takes fewer key strokes to achieve a desired effect, but it is also more complicated to use. For our editor, we decided to implement a semi-modal interface to get the best of both extremes.

The main reason for not implementing a completely modal interface is its limited power. If the interface for our editor were completely modal, the user would not have access to the Walkthrough and birds-eye-view operations while in the editor mode. The user would have to toggle between the editor and Walkthrough modes to perform many common sequences of operations. For example, if the user wanted to select an object, change his/her position, and then move the object, he/she would be required to switch in and out of editor mode at least twice. When performing multiple

editing operations, this switching of modes becomes very tedious and reduces user productivity. With a semi-modal interface, this switching is not required. While in the editor mode, the user still has access to all of the moving operations, so he/she does not loose much of the power associated with a non-modal interface. Some power is lost since the user cannot access the operations of the other modes while editing, but it is not clear why the user, for example, would want to edit the scene and view the cell subdivision at the same time.

To implement the semi-modal interface, we developed a core set of operations that remain consistent throughout all Walkthrough modes. The non-modal core operations, which are always available to the user of the system, consist of the operations that the user performs on a regular bases: the standard walking, turning, and birds-eye-view operations of the existing Walkthrough system [1]. All operations not included in the core operations are mode dependent. When the user enters a mode, for example editor mode, he/she will have access to the core operations plus the operations associated with that mode. When the user enters another mode, visibility mode for example, the user will have access to the core operations and to the visibility operations, but not to the editing operations.

A problem with a non-modal interface approach is that the complexity of key combinations needed to access the operations of the system grows rapidly. Since the user can access any operation at any time, all key combinations used to access operations must be unique. With a modal interface, only the key combinations for operations in a particular mode need to be unique, so they do not become as complex. With our semi-modal interface, the key combinations used in a particular mode must be unique from each other and from the key combinations used for the core operations. Thus the key combinations needed for the semi-modal interface are more complex than those needed in a completely modal approach, but simplier than those needed in a non-modal interface.

Another issue concerning the choice of key combinations, is maintaining consistency of operations associated with them throughout different modes. It is confusing to the user to have a key combination perform one operation in one mode and perform a completely unrelated operation in another mode. With our interface, the operations associated with the key combinations used for the core operations are inherently "consistent" across the different modes. These are the most important operations, since they are the most often used. The key combinations used for the modal operations have been careful designed so that the operations associated with them are as consistent as possible in each mode.

The semi-modal approach to the user interface will also make it easy in the future to develop an on-line help system. The Walkthrough program will always know the present mode of the user. This information can be used to display context sensitive help files discussing the operations and key combination available in the current mode.

2.2.2 Discussion of Object Selection

In implementing our user interface, we have decided to follow the culture exemplified by programs such as MacDraw whenever possible. Performing object selection with the left mouse button and performing multiple selection with the addition of a modifier key are examples of this culture. The dynamic highlighting operation was added to give the user the necessary interactive feedback during the selection process in a cluttered 3D scene. Without dynamic highlighting, it is difficult for the user to select small objects or to select an object from a group of many objects. It is a process of trial and error to position the cursor correctly in order to select the desired object. With dynamic highlighting, the user simple moves the cursor around until the bounding box of the desired object is displayed, then presses the select button. This interface is more user friendly, since the user knows before he/she presses the select button which object is going to be selected. One possible problem with dynamic highlighting is that situations could occur in which this feedback would be intolerably slow. Such a situation arises if there exists a large number of very complex objects with intersecting bounding boxes. For such a case, the user is given the option to perform the selection process by bounding box intersection test instead of face intersection test.

3 Object Selection

To perform object selection we need to: find the "right" object quickly, give the user some visual feedback, and give some assistance in making the new placement easier.

3.1 Finding the "Right" Object

The process of object selection requires mapping the cursor position to a 3D line in the scene and finding the intersection of this "pointing" line with the object closest to the user.

The cursor position is mapped to a 3D line with the help of the function *map2d* which was created at Silicon Graphics. This function uses the cursor's current x and y position and the current system viewing matrix to create a line, which originates at the observers eye position and projects through the cursor position into the scene. The "right" object is found by processing the objects in the building model to find the one closest to the observer and intersected by this line. With thousands of objects in our database, it is not feasible to simple intersect every face of every object with the "pointing" line to find the "right" object. To implement this selection in real time, we take advantage of visibility and subdivision precomputations that have been calculated for the building [2]. The visibility precomputations allow the Walkthrough program to know which objects are visible to the observer, from his/her current position in the building. The Walkthrough maintains a list of these objects. We can restrict the selection test to this small subset of all objects and greatly increase the speed of our selection.

3.1.1 Use of Abstraction

Additional speedup in the selection process can be achieved by using various types of object abstractions to speed up the intersection test. Object bounding boxes are used to trivially reject some of the objects without performing any face intersection test. When face intersection tests are required, they can be done with objects represented at lower "levels of detail". "Levels of detail" is an object abstraction maintained by the database, which contains multiple descriptions of objects each with decreasing amounts of detail. These descriptions are created by removing some vertices and faces from the original object description. Object at lower "levels of detail" have fewer faces and may be slightly smaller than the original object. Performing the selection process by intersecting the "pointing" line with the faces of objects at a lower "level of detail" increases the speed of the process since it requires fewer intersection test. But, when using lower "levels of detail", there is a slight possibility of missing objects during the selection, because of the smaller size of the object's description. Thus when doing face intersection, it is possible that an object, which could have been selected based on its full description, will not

get selected. But since the selection is dynamic this situation is easily corrected by a slight movement of the mouse.

3.1.2 Use of Sorting

Sorting plays an important role in computer graphics. We investigated whether an appropriate use of sorting can speed up the determinations of the "right" object along the "pointing" line. After rejecting some of the objects based on bounding box test, face intersections must be performed on the remaining objects in order to determine the "right" object. If the remaining objects are not sorted, the only way to determine the "right" object is to intersect the "pointing" line with every face of every object and select the object with the closest face intersection point. With sorting, the "right" object can be determine by starting the face intersection test on the closest object, then testing each object until a face intersection is found that is closer to the user than the closest face of the next object. The object with this face intersection then becomes the selected object.

Sorting with Every New Mouse Position

It was determined to be most efficient to sort the objects remaining after the bounding box test every time a new static or dynamic selection occurs. Thus every time the dynamic or static selection routine is called, the bounding box test is performed and the remaining objects are sorted based on their distance from the observer's eye. Next, the face intersection test is performed on each object in this list, starting with the object closest to the user. When a face intersection point is found, it is checked against the bounding box of the next object in the list. If the intersection point is closer to the user than that bounding box, the test is complete. We can be sure that no other objects have face intersection points closer than the present one. With this algorithm, there is a small overhead cost to sort the list of objects with bounding boxes intersected by the "position" line every time the position line is moved. But this overhead time is smaller than the overhead encountered in other approaches considered and briefly discussed below. In those approaches, the sorting algorithm has to be performed on all visible objects, before the bounding box test, and in some cases, it has to be performed more than once for a selection.

Sorting by Observer's Viewing Direction

In order to take advantage of the locality of the user, we considered an approach to sort the objects based on the users current position. In many situations, the user will remain in one position while performing a sequence of editing operations. The program can sort the objects based on the users current position and major viewing direction. So, when the user initiates an editing operations, the list of visible objects is sorted by the distance of each object from the user's eye in the major viewing direction. Then all selections performed from the users current position and viewing direction are performed on this sorted list. If the user moves, the list is recalculated for the next selection. This approach would pay off if many

selections are performed for the current position.

The problem with this approach is that it does not guarantee to the correct result. It is possible for the user's viewing frustum, the angle between the maximum direction that the user can see to his/her right and the maximum direction the user can see to his/her left, to be as large as 180 degrees. In such cases, the users major viewing direction could be along the x-axis while the direction in which he/she is selecting an object could be along the y-axis. One could imagine a situation where the users is looking straight ahead, at something on the wall, but wants to select the chair that is to his/her right. Because of the user's viewing direction, the objects are sorted along the x-axis, but the user's selection is occurring along the y-axis. The object closest to the user along the x-axis is not necessarily the object that is closest to the user along the y-axis.

Presorting in X, Y, and Z directions

It is clear that sorting the objects based on the observer's viewing direction is not adequate for selection, since the "pointing" line is based on the cursor position as well as the user's position. In a further attempt to take advantage of the user's locality, we considered an algorithm to presort all visible objects along each of the major axes. Thus, when the user initiates a selection operation, the program sorts the visible objects for the six directions: one each for the positive and negative directions of each major axis. *(NOTE: The sorted lists could be created in a preprocessing phase and stored with each cell description.)* The program must maintain separate list for the positive and negative directions since the list for the positive direction is not always the inverse of the list for the negative direction due to the different sizes of the bounding boxes. Next, the selection routine determines the major direction of the "pointing" line and performs the selection algorithm with the sorted list corresponding most closely to that direction. The same sorted list is used for all selections in a particular direction within a cell.

The problem with this algorithm is that it requires a lot of overhead every time the user moves into a different cell, which happens quite often in a building model, or it requires a lot a space to store six sorted list for each cell, if preprocessing is used. Furthermore, since a sorted list contains all objects visible from a cell, even the ones behind the observer, after any movement within a cell, the user's position relative to the objects in the list must be calculated. This is required so that only the objects which are in front of the user will be used in the intersection test.

3.2 Visual Feedback

Once an object is selected, it must be displayed in a way that allows the user to easily interact with it and to move it around quickly. The bounding box of the object will always be displayed as a first level of highlighting. Displaying the bounding box is an inexpensive operation that effectively highlights an object. It also gives the user information about the positioning of the object relative to other objects and about the overall size of the object. We have had much debate about what other

information should be displayed along with the object. Our goal was to display enough information to enhance the users interaction with the object but not so much information that the users view of the object will be obscured (Figure 3).

3.2.1 Displaying the Object Itself

An object can be displayed on the screen in many different ways. Some ways better highlight the object and other ways allow for better manipulation of the object. An object can be displayed as it normally appears in the Walkthrough, usually as a shaded solid object. This allows for easy manipulation of the object. The user knows exactly how the object looks in its new position. Also, displaying the object in this manner better emphasizes depth cues, created by the change in the object's color due to the lighting model, than other drawing options, for example wire frame. The disadvantages of displaying the shaded object is that the object can hide other objects or reference lines that the user might want to use to perform a precise transformation. For example, if the user is trying to place a trash can flush against the leg of a desk, displaying the trash can in its normal form may hide the desk leg from the user. The user may then have difficulties performing this operation without selecting a new viewing direction.

An object can be displayed semi-transparently. The advantage of transparency is that the object does not hide any information that is behind or under it. In the situation discussed above, the user will be able to see when the trash can is being pushed into the desk leg by observing the disappearance of some portions of the semi-transparent back surface of the can. The user still has a good idea of how the object looks in relation to other objects in the scene. Furthermore, the user can still see some of the depth cues from shadding of the object when it is displayed transparently. Unfortunately, not all workstations have hardware support for transparency and software implementation of transparency is too slow for interactive manipulation.

Displaying an object as a wire frame has the advantage that it does not hide any information below or behind the object. The user has to use some imagination to see exactly what the object looks like, but with most objects this is not very difficult. Wire frame has the problem that the depth perception of the viewer could be distorted if the object is composed of a few large faces. Also, the depth and positioning cues generated by a lighting model for a wire frame object are not as helpful as those generated from shading an object in normal or even transparent mode. The advantage of a wire frame display is that it is easy to implement and can be manipulated quickly on almost all machines.

For very complex objects, such as a tree with thousands of leaves, the user may not want to display the whole object. Interactivity and speed may be more important than accuracy of placement. Bounding box mode allows the user to manipulate the tree based solely on its bounding box. In the Walkthrough world, many objects have a topology that is mainly rectangular, resulting in bounding boxes

that give a reasonable representation of the object. Not displaying the object at all obviously does not hide any information, but the user has to use a lot of imagination to visualize the object in its new position. In general it is difficult for the user to interact with only the bounding box of an object. However, ofter some initial placement, the user can quickly de-select the object and see it fully rendered in its new position. One or two more selections and position adjustments will typically lead to the desired placement.

In the editor the user is given the option to display the object in any of the ways discussed above, since in different situations different options will be most appropriate. The default is to display the object transparently. Upon implementation and inspection, we have determined that transparency gives the user the most information. If the machine does not support transparency, then the program will change the default setting to wire frame display.

3.2.2 Displaying a Hot Point

Besides displaying the object, the program can display additional information to assist the user in manipulating the object. We decided to add the idea of a hot point. This is a highlighted interaction point on the selected object where the "pointing" line first intersects the object. Displaying a hot point gives the user extra feedback of what has been selected.

The debate about where exactly this point should be displayed originally concentrated on three options. The first option considered was to place the hot point at the intersection of the "pointing" line with either the floor or the wall polygon that the object is placed on. The advantages of placing the hot point in this location is that it can give the user feedback about the intended placement of the object. For example, when moving an object aligned on a floor polygon, displaying this hot point gives information of the objects location relative to other object on the floor. A problem is encountered if the object is being displayed normally (see above): the hot point will typically be obscured by the object, and is then of no use to the user. Furthermore, if the selection line is close to horizontal, then the intersection of the "pointing" line and the floor polygon may be far away from the object that is being manipulated. Worse, if the "pointing" line has an upward slope the intersection point will lie behind the sight of the user and will not be visible. Since situations like these occur fairly often in our Walkthrough, this type of hot point did not seem suitable for our program.

The second option is to display the hot point at the intersection of the "pointing" line and the first intersection of the bounding box of the object (front or back face). This position has the advantage that it is easy to calculate and will never be behind the observers position. The disadvantage of displaying the hot point at this location is that it will not always be intuitive to the user how this point relates to the object. Some object have a very large bounding box in relationship to the actual size of the object. Selecting such an object could result in a hot point that is

far away from the actual surface of the object. It will appear to the user that the hot point is just floating somewhere in space, making the relationship between the hot point and the object unclear.

For our Walkthrough program, we decided that the best place to display the hot point is on the face of the selected object. We place the hot point at the intersection of the "pointing" line and the face of the object that was used to select the object. This point has already been calculated during the selection routine. The advantage of this approach is that the hot point will always be located on the surface of the object and will always be visible. It is the actual point that was used to select the object, so the user's attention is already focused on this position. The hot point is drawn as an octahedron, thus part of it may be drawn inside of the object giving additional hints about the contour of the object.

Implementation and experimentation with all three approaches proved that the face intersection point was the most consistent and intuitive location to place the hot point. Thus, by default, the hot point will be displayed at the face intersection point, but if the user selects the option to perform object selection by bounding boxes, then the hot point will be displayed at the bounding box intersection point and not at the face intersection point.

3.2.3 Displaying Reference Lines

The editor gives the user the option to display refence lines to provide additional information about how the object is related to the things around it. Reference lines are axial lines originating at the hot point and extending out some distance. The reference lines can give the user information about the size of the object, its distance relative to other objects, or its distance from the nearby floors and walls of the building. We have given the user three options of how to display these lines, in an attempt to find a simple configuration that gives the user the most information. The user has the option of not displaying the reference lines at all, of displaying the reference lines from the hot point to the extent of the object's bounding box, or of displaying the reference lines from the hot point through the entire database. In each case, these lines exist as real 3D lines appropriately obscured by other objects.

The option of not displaying the reference lines at all, is useful when the user is manipulating small objects. Sometimes it is more important not to obstruct the user's view than it is to give the user additional information. The option of drawing the reference lines only to the extent of the object's bounding box is useful when the user is trying to place an object so that it abuts or sits on another object. For example, if the user was trying to position a desk in the corner of a room, he/she would manipulate the desk so that the end points of the reference lines are flush with the walls of the corner. Octahedrons are drawn at the ends of the reference lines to give the user an indication when they are touching another object. If a reference line is not touching a wall, then the desk is too far away, and if a reference line goes through a wall, then the desk is too close. The options of displaying the reference

lines to the extent of the database are helpful when the user is doing some type of global positioning. They allow the user to register an object with other objects which might even be in other rooms. Also, these extended lines help to find an object that might have snapped out of sight.

Before making the final decision on reference lines, we considered other options for drawing the reference lines. For example, we considered drawing axial lines from the hot point to the point where the line intersects the objects bounding box and then drawing additional perpendicular lines from these new intersection points to the points of intersection between the line and the nearest wall or floor polygon. The reference lines can also be draw axial from the hot point to the point where the line intersects another object, but this is functionally equivalent to drawing the lines through the entire database. We decided that the three choices given to the user, discussed above, are an adequate subset of the many possible reference lines that could be drawn. By switching between the three modes, the user can obtain most of the desired effects that might be obtained from the other approaches considered.

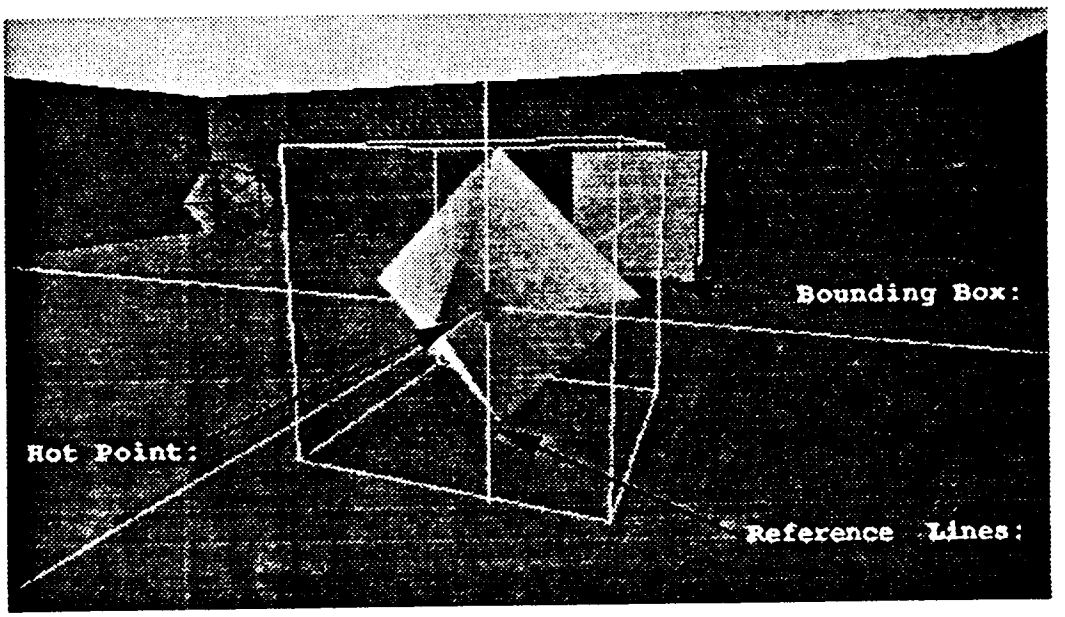

Figure 3: Selected object displayed with highlighting information.

4 Object Properties

Object properties are a way for the user to specify placement constraints for some objects in his/her model. For example, desks can be given the property that they must stand on the floor. It would be strange to walk into an office and see a desk floating a few inches above the ground. The property definitions simplify the user's interaction with the editor. If an object has the property that it is suppose to be on the floor, then the program can place the object on the floor and the user does not have to worry about getting the object at exactly the correct height, which is not a trivial task. Also when translating the object, the editor knows that the object only needs to be moved in the XY-plane, thus the editor can implement translation of that object in a 2D space rather than a 3D space. If the object did not have such a property, the editor would have to find a way to allow the user to move the object anywhere in a 3D space with a 2D pointing device.

Placement properties also form the basis for grouping of objects. This is useful if the user wants to move an object such as a desk with many other objects on top of it. Ideally the user only has to select and move the desk and all of the objects on the desk will move as well.

Property definitions could be used to define many other object characteristics, and future editors may well incorporate many of these new properties (see Conclusion section). The following sub-sections discuss: the available prototype object properties, how a user defines object properties, and what is required by the system operator to implement new properties.

4.1 Two Placement Properties

For our prototype editor we have implemented two properties, "ON_HORIZONTAL" and "ON_VERTICAL". Giving an object the "ON_HORIZONTAL" property, tells the editor that the object should rest on a floor or another horizontal surface. For example, if the user places a desk into the model, which has the "ON_HORIZONTAL" property and is positioned within a reasonable distance of a horizontal floor polygon, the desk will snap to the correct position on the floor. The editor then constrains object's with this property, which have been successfully bound to a suitable surface, to move only in the XY-plane as long as it remains above that surface.

An object's constraint can be interactively changed while in the editor, to allow arbitrary movement and positioning. While moving an object with the "ON_HORIZONTAL" property in the scene, the editor will snap the object to the closest suitable surface below the object's current position. (Note: the "ON_HORIZONTAL" property is currently the default object property used by the editor for all objects that do not have another positioning constraint specified.)

Giving an object the "ON_VERTICAL" property, tells the editor that the object should rest on a wall or another vertical surface. This property is useful for

objects such as pictures and white boards. With this property, object movement is constrained to a plane parallel to the Z-axis. To determine which plane the object is to be moved in, the editor chooses the surface that the object is currently snapped to, or, if the object is still unattached, looks for a suitable surface nearest to the object. This movement plane will change as the object is moved from one wall to the next. Again, the user can interactively override an object's constraints to allow movement and positioning in a horizontal plane. For a discussion on how the editor constrains the user's transformations of object that have the "ON_HORIZONTAL" or the "ON_VERTICAL" property, see the section on Transformations.

4.2 Object Grouping

Grouping of objects is useful when the user wants to move an object and all objects that are related to that object. For example if the user wants to move a desk, then the objects on the desk are the related objects, and they should move with the desk. Without object grouping, the user would first have to select every object on the desk, then grab and move the desk. With this approach there is the possibility that the user might miss an object. Then he/she would have to go back, grab the missed object and try to correctly re-position that object on the desk.

To bind an object to another object, the user has to place an object so that its property constraint is satisfied by snapping to the desired object. For example, if a user defines books to have the "ON_HORIZONTAL" property, then such a book that has snapped onto a desktop will be related to that desk. When the desk is moved, the book will move with it.

4.3 Defining Object Properties

Object property definitions are created by the user in a separate text file. In a future system, property definitions may be integrated directly into an extended UNIGRAFIX file, but for the present system, we decided it would cost too much time to modify the UNIGRAFIX structures. In this user-created text file, the *property file*, the user can define object properties and assign these properties to object types. The name of an object type is equivalent to the name used to create the object definition in the UNIGRAFIX file.

Property definitions are defined by user-specified object property sets. The user defines object property sets to contain all of the properties that characterize an object type. Object property sets are enclosed between the key words **def prop** and **end;** and consist of the property set name and a list of the properties that make up the desired set. The property list can include the prototype placement properties currently supplied by the editor system, "ON_HORIZONTAL" and "ON_VERTICAL", as well as properties related to the other aspects of the object, such as rendering. For example, there could be a "ROUNDED" property to specify that the object should be rendered as a rounded surface. A property definition can be used by any number

of object types. The syntax for object property set definition follows the convention used in UNIGRAFIX for object type definition:

def prop *property_name*
property;
... ;
... ;
end;

Object property sets are bound to specific object types by a user-defined assignment statement. The user can use the assignment statements to bind an object property set to any number of object types. Object property sets must be bound to an object type before they have any effect on the objects of that type. The keyword **give** is used to define a property assignment statement. This key word is followed by the name of the object type then the name of the object property set. The syntax of the assignment statement is a follows:

give *object_type property_name;*

In the following example, there are three object property set definitions and four bindings of those properties sets to object types. Notice that in the second object property set definition, the user has defined two properties to be used for the cupprop property set.

Property Definition File: office.pr

{ Definitions of office properties. }
def prop deskprop;
ON_HORIZONTAL;
end;

def prop cupprop;
ON_HORIZONTAL;
ROUNDED;
end;

def prop pictureprop;
ON_VERTICAL;
end;

{ Assignment of properties to object. }
give desk deskprop;
give cup cupprop;

give picture pictureprop;
give white_board pictureprop;

In the first two object property sets the "ON_HORIZONTAL" property is used. Any object type bound to use the property set deskprop, will have the properties exhibited by "ON_HORIZONTAL". The property set cupprop has an addition property, "ROUNDED", so any object bound to cupprop will have the properties characteristic of "ON_HORIZONTAL" as well as those characteristic of "ROUNDED". The third property set only contains the "ON_VERTICAL" property. In the four assignment statements the various object types are bound to the various property sets. For example, objects of type desk are bound to the properties defined by deskprop. Notice that the picture board and the white_board use the same property set.

4.4 Implementation of Properties

In order for the editor to use object properties, certain files need to exist. There are three files required: the *property set definition file* discussed above, which contains the object property set definitions and the "give" assignment statements, the *object type table file*, which contains the names of all available object types and is created by the database routine, and the *property table* file, which contains the name of the available properties and is created by the editor routine. The system operator needs to ensure that the property set definition file is properly created. The type table file and the property table file can be used by the system operator to verify naming of object types and property types if needed.

4.4.1 Use of Property Definitions

When manipulating a selected object, the editor uses database information stored for that object along with the current state of the editor to determine which movement constraint property the object should satisfy. Procedures that define the properties are built into the editor program. The editor uses the property table file and the type table file *(discussed below)* to create lookup tables for internal property names and for object type names. When the property set definition file is loaded, each property definition set is given a unique id value, and a property set table is create containing this id value, and a list of property id values, which are obtained by replacing the property names with their corresponding id values in the property name lookup table. Another lookup table is created for the assignment statements. This lookup table contains the id value of the object type, obtained from the object type name lookup table, and the newly created id value of its assigned object property set. When the editor performs a transformation of an object, the object's id value is obtained from the object's stored database information. Then the object property set id value is found, by finding the corresponding type id value in the assignment lookup table. With the property set id, the associated property id is found from the

property set table, and then the proper handling routine is called (see Transformation section for handling properties).

4.4.2 The Object Type Table File

In the Walkthrough program, integers are used as identifiers for objects and other constructs. The table files are used to create lookup tables that map the ascii object and property names in the property set definition file to integer identifiers. These files were made external to insure consistent use of id values throughout the various phases of the Walkthrough program. A large database model can be created in parts, so objects of the same type can be added into the database at different times. With external mapping files, we can ensure that objects loaded by different routines at different times will obtain the same id value.

The database routine uses the type table file to set an object's type id value whenever a new object is added into the database. The same type table file is used by the editor to load the id values for object names in the property set definition table. This ensures that an object type id value stored in the database is the same as an id value of the same object type used by the editor. The object type table file contains a list of available object type names and their corresponding id value. This file is created by *wkadd* when the database is first created. *Wkadd* is a database routine that creates the necessary object data structures and adds the object descriptions into the database. When an object is added, its type id is set to the id value in the type file corresponding to that object type. If there is no entry yet in the file corresponding to that object type, then a new entry is created with a unique id value. When an object is selected in the editor, the program retrieves this type id value and uses the type table to determine which type of object was selected. Following is an example of a type table file:

Type file: buildingX.ty

DESK 1
TABLE 2
PICTURE 3
SPHERE 5
CUP 6

4.4.3 The Property Table File

The property table file contains the names and corresponding id values of all properties available in the current version of the editor program. In the present system, the property table file could be internal, but it was made external because in future implementations, when the property definitions are in a UNIGRAFIX file, this table

file might be needed. The property table file is used by the editor to replace property names in the property definition file with unique id values. Following is an example of the currently used property table file:

Property definition table file: editor92.pr

```
ON_HORIZONTAL 1
ON_VERTICAL 2
```

5 Transformations

Translating and rotating objects is the main use of the editor. Therefore it should be easy for the user to access and perform these operations. To implement translation, we had to solve the well known problem of moving an object unambiguously in 3D with a 2D mouse. We took advantage of the object properties that constrain the positions and thus the movements of objects. For example, if an object is constrained to be on a horizontal surface, in general, it can only be moved in the XY-plane. With these constraints, object movements are typically limited to a 2D space. The user has the ability to override these constraints whenever necessary. We also give the user access to other translation features that give him/her more control over an object's movement.

Rotation around a preferred axis is also a commonly used function when rearranging furniture. A simple rotation function, typically constrained to lie within a plane defined by the object's placement property, is easily accessible by the user. Again the user can access an extensive rotation function if desired. We have also implemented some absolute keyboard-controlled transformation functions, to allow the user to do things such as rotate an object by exactly 90 degrees.

Scaling, on the other hand, is an operation that the user does not have easy access to. We do not want to make it too easy to scale objects, since the objects have typically been created at the correct scale. Once the user scales an object, we can no longer guarantee that our model is realistic. (We could have giant pencils laying all over the place!) But again, the user can gain access to the scaling operation if necessary.

5.1 Defining an Interaction Point

An interaction point is a reference point for performing a transformation. For example, when rotating an object, the interaction point is the point that the object is rotated around. In our editor, we define an interaction point for each type of transformation. If the user were required to define the interaction point, it would greatly reduce his/her productivity. For translation, the position of the interaction point does not affect the result of the translation, so we use the hot point as the interaction point. For rotations, the interaction point is defined as the center of the object's bounding box. If the rotation interaction point is not at the center of the object's bounding box, then the object will be translated as well as rotated during the rotation. This side effect, while useful in some situations, is not the general rotation we desire. For the exact scaling mode, the interaction point is the center of the objects bounding box. For scaling with specialized handles, we give the user some control over the interaction point. The interaction point is determined by the interaction handle the user grabs and is the corner of the object's bounding box opposite from the handle selected.

An interaction handle is used to distinguish between the actions of selecting

an object and of grabbing an object to perform a transformation. If the user grabs an interaction handle, the program assume that he/she wants to perform a transformation on the object, otherwise the program assumes that he/she wants to select the object. For the general transformations, translation and rotation, the interaction handle is simply the selected object. If the user grabs an object that is already selected, the program assumes that he/she wants to transform that object. For the more constrained transformations, the program displays explicit handles. These handles represent different transformations and, depending on the handle selected, the user can perform these specific transformations (see Transformations with Specialized Handles).

5.2 General Transformations

In the general transformation mode, the user can perform translation and rotation operations with little overhead. The user simply selects an object, then grabs it with the correct mouse button to translate or rotate the object. Object transformations are constrained by the object properties, as mentioned above. If the object does not have a constraining property or if the editor cannot find a suitable face for the object, then the object is translated in a plane perpendicular to the "pointing" line. For the general translation, the handle is the object itself, so the user simply selects any point on the object, and that point becomes the handle as well as the hot point (or interaction point). The user can then drag the object around in the scene by this handle. The handle for the general rotation is the same as for the translation. However, the interaction point for the general rotation is always the center of the object's bounding box. So the user grabs an object at any point, and the object will be rotated around its center as the user moves the cursor around.

5.3 Transformations with Specialized Handles

When performing the more constrained transformations, the user is given specific handles to interact with. These handles are displayed as cubes and have visual aids attached to them to help clarify their function. Different transformation will be performed depending on which handle is selected.

5.3.1 Translation

For translation, the hot point is replaced by three interaction handles. The position of the handles are determined just as the position of the hot point was determined above (see *Displaying a Hot Point*). Using the hot point location has the advantage that the mouse is already located at this position, so the user does not have to move the mouse very far to grab a handle. To grab a handle, the user places the mouse over the handle, then presses and holds the left mouse button. With the handle grabbed, the user can drag the object around the scene. The three handles are located at some

offset from the hot point location. The size and distance of the offset are determined by the size of the selected object's bounding box. Thus larger objects have larger handles and smaller objects have smaller handles (Figure 4). This is similar to real life in that you would not put a large handle on a pencil in order to help you move it more easily. Varying the handle size has the effect that the user must be closer to smaller objects in order to perform the more constrained transformations. For example, we do not want the user to be able to move a pencil around on the desk from two rooms away because he might not be able to see if he stuck the pencil partly into a cup.

Each handle is used to move the object in a particular plane and has a larger flat square connected to it, which is parallel to one of the three major planes: one parallel to the XY-plane, one to the XZ-plane and one to the YZ-plane. To move an object in a particular plane, the user grabs the handle that has the square parallel to that plane attached to it. As long as the user holds down the left mouse button, the object will be dragged in the desired plane. For example, if the user grabs the handle that is connected to the square parallel to the XY-plane, he/she will be able to move the object around in the XY-plane only. Thus the problem of moving objects in a 3D space with a 2D mouse is reduced to moving objects only in a 2D space which can be done conveniently with a mouse.

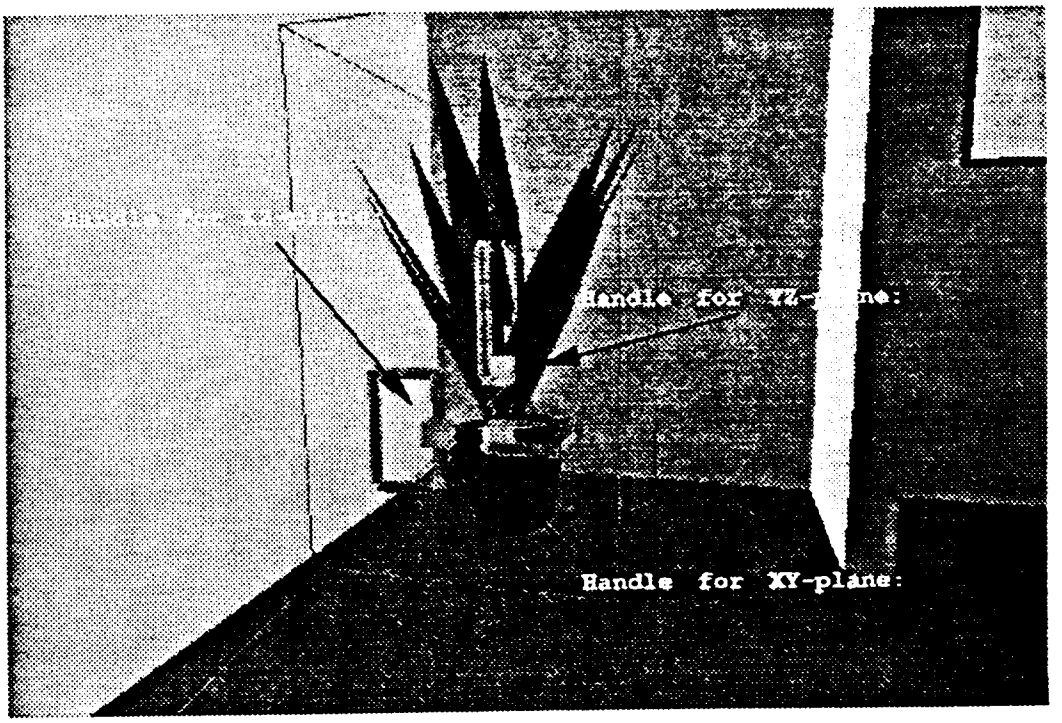

Figure 4: Specialized translation handles.

5.3.2 Rotations

There are six rotation interaction handles. They are located at the intersections of the three great circles, originating at the center of the object's bounding box with a radius equal to the distance from the center of the bounding box to the center of the furthest edges. To help the user visualize what is happening, the three great circles are also drawn (Figure 5). The handles lie at the crossings of these circles. When the user selects one of the handles, he/she is allowed to rotate the object in either of the directions represented by the two circles intersecting that handle. The rotation is performed by dragging the handle in the plane of the desired circle. Once again, the problem of moving an object in 3D space with a 2D mouse is simplified to a sequence of 2D operations.

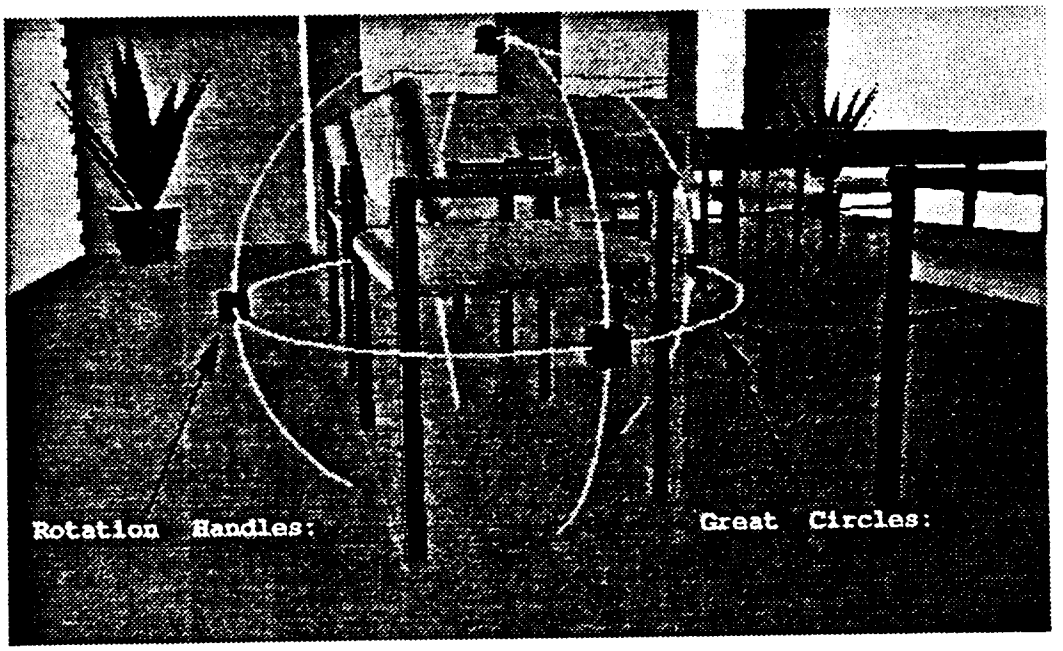

Figure 5: Specialized rotation handles.

5.3.3 Scaling

For scaling, there are eight interaction handles located at the corners of the object's bounding box (Figure 6). When a users selects one of these handles, the object is uniformly scaled around the opposite corner of the bounding box. For example, assume that the user is looking at the the face of the cube that is in the XZ plane and he/she grabs the handle on the front, upper right corner of the bounding box. Dragging the cube to the right will uniformly enlarge the object relative to the distance that the handle is moved. Dragging the handle to the left will uniformly shrink the object in a similar fashion. During either of these operations, the handle at the back, bottom left corner of the cube will remain stationary. This is similar to scaling operations in existing editors such as MacDraw. Non-uniform scaling

of objects can only be achieved by the use of the exact transformation operations discussed below.

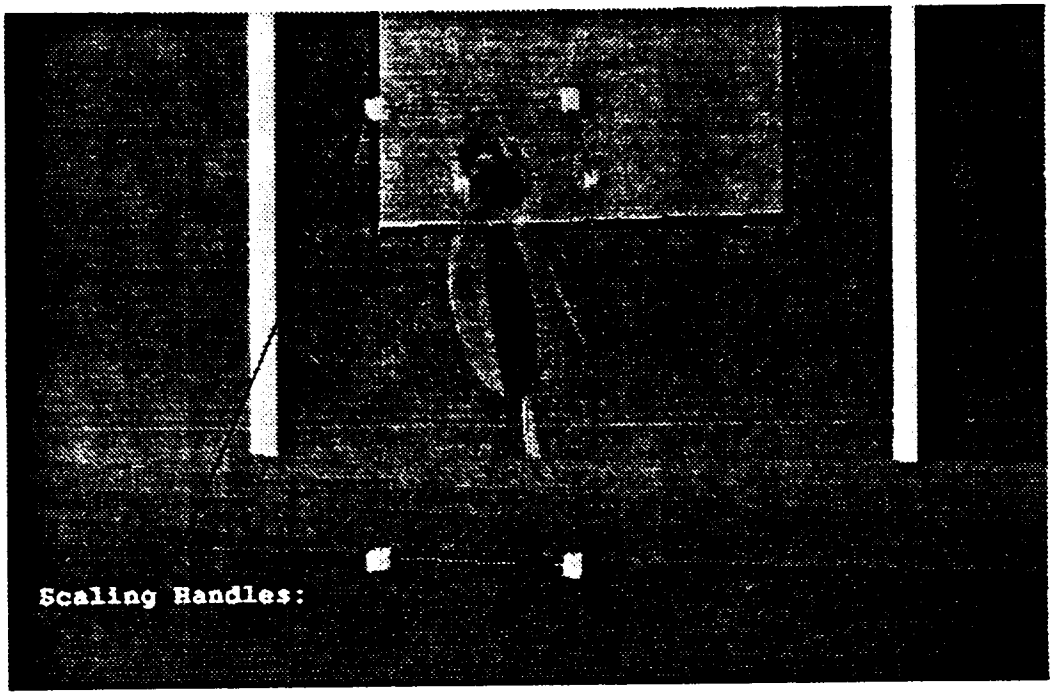

Figure 6: Specialized scaling handles.

5.4 Exact Transformations

The user must use the dialog menu to perform exact, numerically defined, transformations of objects. The user presses the Transformation button on the dialog menu to bring forward a new menu that allows keyboard entry of distances, rotation angles, and scaling factors (see Tutorial section for details). When the user presses one of the transformation buttons, the selected object is transformed by the current values for that transformation. For example, in order for the user to rotate an object by exactly 90 degrees around the Z axis, he/she first selects the desired object. Then the user sets the Z rotation value in the transformation dialog to 90, and presses the Z-rotation button. This rotates the object by 90 degrees around the Z axis. The same type of actions are performed to rotate around the X or Y-axis. To non-uniformly stretch an object by a factor of 2.5 in the X-direction only, the user would set the X-scale factor in the dialog menu to 2.5 and set the Y and Z-scale factors to 1.0 (the default value), then press the scale button.

5.5 Transforming Multiple Objects

Even when multiple objects have been selected, the user only interacts with one of them. In the general transformation mode, this is the one that the user grabs to

execute the transformation. In one of the constrained transformation modes, it is the last object selected. The interaction handles associated with the present mode will move from object to object as the user adds new objects to the selection.

There are two possible ways to perform transformations on multiple objects. The program could either transform each object around its own interaction point or transform all objects around one common interaction point. When performing translations, there is no distinction between the results of the two implementations.

For rotations, we chose to implement the latter of the two methods. The common rotation point will be the center of the bounding box of the last object selected. To implement this type of rotation, we perform: a translation of each object by the -X, -Y and -Z values of the interaction point, a rotation operation relative to the user's mouse movement, and then another translation by +X, +Y and +Z. This is useful for rotating groups of object, since it will result in all selected objects rotating jointly around the one last-selected object. In the other approach, rotating multiple objects would result in each object staying in place and rotating around the center of its own bounding box, which could lead to interference among them.

For scaling, we have chosen to implement the former of the two. When scaling multiple objects with our editor, each object will scale by the same factor, but one corner of each object will remain stationary. The stationary corner for each object is the object's bounding box corner that is in the same relative position as the corner opposite from the selected handle. To implement this, we perform: a translation of each object by the -X, -Y and -Z values of the object's stationary corner, a uniform scaling based on the users mouse movement, and a translation by +X, +Y and +Z. If we were to implement the other method of transformation, each object would be scaled around the same common point, possible resulting in substantial translations for some objects.

5.6 Constraint Based Translations

We have created the first implementation of a constraint based placement system for use in our editor. The constraint system will affect how objects are moved and positioned in the scene. This first implementation is a prototype for a future extension to UNIGRAFIX that will allow objects to have many properties (see Future section). The properties that the editor knows how to handle are stored in the system file *editor92.pr* (see *Property Definitions* section). For this prototype, we have implemented two properties "ON_HORIZONTAL" and "ON_VERTICAL", thus allowing objects in our system to have one of two possible properties.

5.6.1 Handling the "ON_HORIZONTAL" constraint

The first constraint "ON_HORIZONTAL", is inteded for use by pieces of furniture and objects that lie on top of desks, shelves, or tables. This property informs the program that an object should be positioned with respect to a horizontal surface. This property is considered whenever an object is manipulated by a user with the editor or when an object is placed in the database by some batch program. To enforce the constraint, the editor creates a search line parallel to the Z-axis, that extends directly below the object. In interactive mode, the line starts at the hot point of the selected object. If the editor is being called by a batch program, then the line starts at the center of the object's bounding box. A suitable snapping face is the face closest to the object that is intersected by the search line and is parallel to the XY-plane.

If the object comes within a certain distance of a suitable face, then the object is snapped to that face. The distance, to determine whether or not an object snaps, is dependent on the snapping mode. There are two available snapping modes: local mode, in which an object only snaps if it is within the distance of half its bounding box to a suitable surface, and global mode, in which an object will snap if any suitable face can be found inside the room that the object is currently in. The snapping is done by translating the object in the Z direction so that the bottom of the object's bounding box is moved to the point of intersection between the search line and the suitable surface (Figure 7) (see [5] for more information on snapping).

During a transformation of multiple objects, only the constraint of the object that was grabbed will be enforced. When the user ends the transformation, the program will check the constraints of the other objects and reposition them if necessary.

5.6.2 Handling the "ON_VERTICAL" constraint

The "ON_VERTICAL" property is intended for use by objects such as white boards and wall pictures. This constraint will be enforced in a similar fashion as the "ON_HORIZONTAL" constraint discussed above. The difference is that the search line for the suitable face will be drawn in the XY plane. The line will start at the hot point, or at the center of the object's bounding box for batch programs, and will extend in the same (X,Y) direction as the "pointing" line (defined in the object selection section). When handling this constraint in batch mode, there is no "pointing" line, so we create two search lines: one parallel to the X-axis and one parallel to the Y-axis, and then we use the search line that results in the closest suitable face. The snapping of the object will occur by a translation of the object in both the X and the Y directions. The distance of the translation is the distance from the intersection point of the search line with the suitable face to the intersection of the search line with the object's bounding box face that is closest to the suitable face (Figure 8).

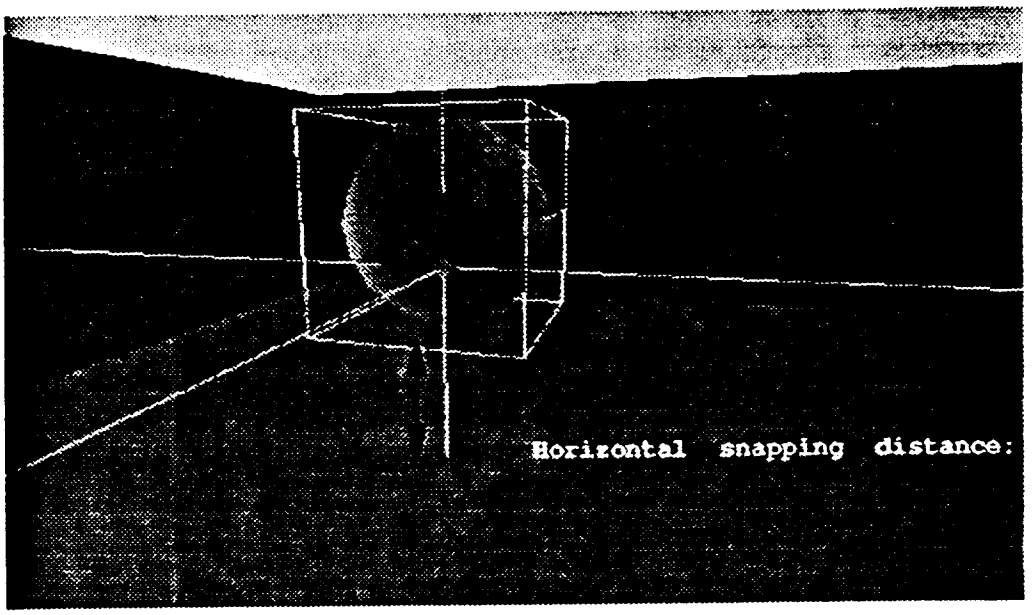

Figure 7: Handling the ON_HORIZONTAL property.

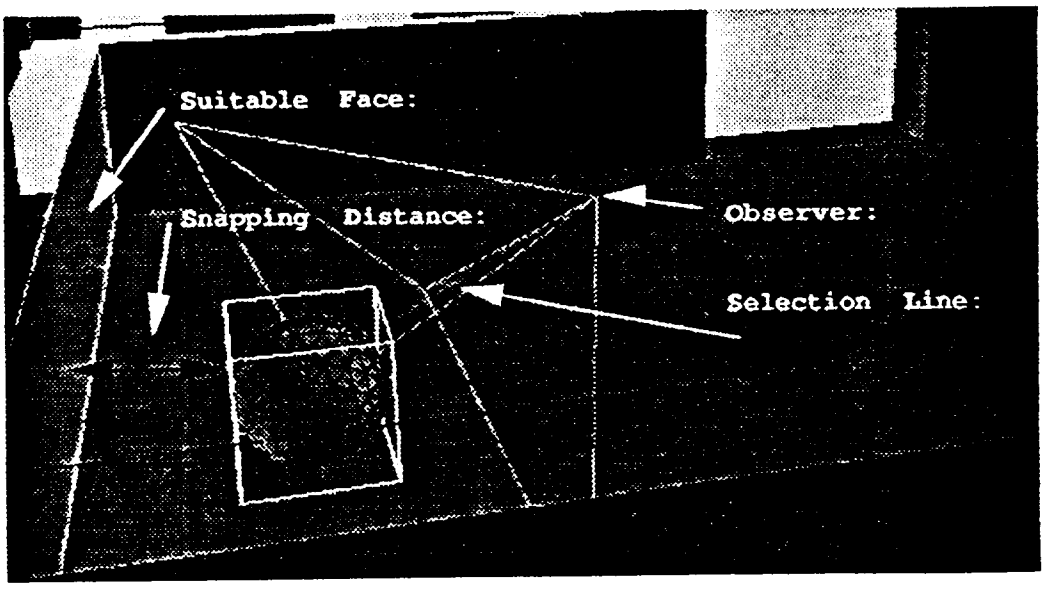

Figure 8: Handling the ON_VERTICAL property.

We encountered additional problems in implementing the "ON VERTICAL" constraint that we did not have to worry about with the "ON_HORIZONTAL" constraint. When a relatively flat object snaps from one wall to an adjacent wall, a common occurrence in building models, the object will be oriented incorrectly on the new wall. The flat face or back of the object will not be planar with the wall, resulting in part of the object sticking into the wall and part of the object projecting into the room. For example, if the user is sliding a picture along a wall and it snaps to a perpendicular wall, the picture will be oriented incorrectly. The back of the picture will be perpendicular to the new wall. In this situation, the user would want the object to be rotated by 90 degrees so that it remains correctly oriented on the wall. In general, an object with the "ON_VERTICAL" property should always be rotated so that the back of the object is coplanar with its suitable surface. In the Walkthrough database, we do not have the information to determine the back of the object, so we assume that the object is initially positioned with the correct orientation close enough to a suitable face. To handle this problem, we added an additional check to the "ON_VERTICAL" constraint handler. Whenever an object with the "ON_VERTICAL" property is selected for manipulation, the program checks to see whether or not the object is within snapping distance of a suitable face. If so, the program keeps track of the normal of this face. Then whenever the object snaps to a face with a normal different from the previous normal, we rotate the object around the Z-axis. The rotation is around the center of object's bounding box by the angle corresponding to the difference of the two face normals. With this solution, the object will only stick into the wall when the user positions it close to a corner, but since the editor is an interactive system, the user can easily notice and correct this situation.

5.6.3 Both Constraints

Particular types of furniture are characterized to exhibit both of the positioning constraints defined for the "ON_HORIZONTAL" and the "ON_VERTICAL" properties. For example, bookshelves should be placed on a floor as well as against a wall. To handle objects with both properties, we have implemented a greedy algorithm that simple handles each of the properties in the order that they appear. So, if bookshelves are defined to have the "ON_HORIZONTAL" and the "ON_VERTICAL" properties, in that order, the program first snaps a bookshelf to a horizontal suitable surface, if one can be found, then snaps it to a vertical suitable surface, if one can be found from the new location.

5.6.4 Additional Constraints

Besides the user defined constraints, we have implemented an additional constraint which prevents the user from moving an object inadvertently through a wall or floor polygon. This means that the user cannot grab an object in one room and push it through the wall to into another room. Thus, in general, if the user wants to move

an object from one office to another, he/she is required to drag it through the doors of each office. However, in some cases, there may be a quicker way, if the object is large enough (e.g., a desk) and the walls are thin enough. The user could push the desk partly through the wall (the hot point is not allowed to go through a wall) then go into the other room, reselect the desk and pull it into the new room. The reason for enforcing this constraint is to prevent the user from accidently misplacing an object. If the program allowed objects to be moved through walls freely, the user would be able to move an object into a position where it could not be seen and then he/she could only guess about its location and orientation. Also, if the user accidently moved an object through a wall and released the selection button, the object would be removed from the users current visible object list and could no longer be selected from the users current position. The user would be required to search for the object in the other rooms of the model in order to re-select it; and worse yet, if the entire object is inside of a wall, it might be permanently lost.

6 Copy and Undo

In the editor we have also implemented the commonly used copy and undo operations. The copy operation allows the user to populate the building model more easily. By simply selecting an object and pressing the copy button, a new object is added into the database. The undo operation will undo the user's last transformation. This allows the user to move an object back to its original position if he/she makes a mistake or does not like the new position of the object.

6.1 The Copy Function

The copy operation is used to add an object into the database. When the user selects one or more objects, the editor maintains a copy of these objects. To perform the copy operation, the editor creates a new object in the database by calling a database routine to add a new object. This routine takes the object definition and creates the new pointers and identifiers necessary for a database object. When a new object is added into the database, the cell-to-object visibility of all cells visible from the cell the object was inserted into are updated to determine whether or not the new object is visible to them. In order to prevent the user from accidently making multiple copies of an object in the exact same position, the new object is not actually created until the original object is moved. This is done by setting a copy flag, then creating the new object after a transformation has been completed.

Before implementation of the editor, a copy operation consisted of creating a new instance of the object, either by editing the text UNIGRAFIX file or by using a CAD tool. Then the desired transformations were applied to the object again either with UNIGRAFIX or a CAD tool. Next, the new object was added to the database by using a batch program, which creates the object structure, sets the correct pointers and performs the visibility calculation. Finally the the new object was inspected by

running the Walkthrough program. The copy function provides a way to increase the furniture density in a building model much more quickly.

6.2 The Undo Function

We have implemented a one-level undo for the editor. At the end of every transformation, marked by the selection of a new object, the editor updates its undo structure. The undo structure contains an object definition, a transformation matrix, an undo flag and X, Y, and Z variables for rotation, translation and scaling to keep track of the current transformation. The transformation matrix is a composite matrix of all transformations that have been applied to the currently selected object. This matrix is created by keeping track of the user's mouse movements and the current type of transformation. For example, if the user is currently in the translation mode, the undo structure keeps track of the delta X, delta Y and delta Z between the object's original position and the object's new position. These values are used to create a translation matrix which is multiplied into the undo transformation matrix. When the undo button is pressed, the undo flag is toggled between true and false. If the undo flag is false, then all selected objects are first transformed by the matrix in their undo structure and then drawn. If the undo flag is true, then the objects are not transformed by their undo transformation matrix, they are drawn at their original database position. This has the effect of undoing any transformations that have occurred since the object was selected. If a new object is selected while the undo flag is true, then it is not required to update the position of the old object in the database. Once the user selects a new object, the previous transformations can no-longer be undone. Before the user "accidently" looses the current object selection, when trying to grab the object from a new interaction point, he/she would see the white highlighted bounding box of another object to indicate that on clicking the mouse a different object would be selected.

7 Handling Constraints in Batch Mode

The batch version of the editor is a pre-processing program that can be used on a building model to verify that objects satisfy their property constraints. An object that does not satisfy its property constraints is translated, if possible, to a suitable surface so that it does. This batch program is especially useful on large building models. In order to perform the same function without the batch program, the user would have to walk through the entire model and select every object. Each object selected would then snap to a suitable face and satisfy its property constraints. With large building models, this would be very time consuming. The user would need to keep track of which object have already been selected and which rooms have already been verified. With batch mode constraint processing, the user simple runs the program which checks every object in the model. The batch program produces an output file which contains the original and final positions of all transformed objects and a list of objects that have placement constraints but were not close enough to a suitable face to be translated. This file can be used by another utility (not yet implemented) to update the UNIGRAFIX files for later models (See Updating the UNIGRAFIX Master File).

7.1 Implementation of Batch Mode

Handling constraints in the batch program is implemented in much the same way as in the interactive program. To verify an object's placement constraints, the program must know all objects visible to that object. The database does not maintain object-to-object visibility information, but it does maintain cell-to-object visibility information. To take advantage of this information, the batch program processes the database in a cell by cell mode, trying to satisfy the constraints of all objects "centered in" a cell before moving to a new cell. An object is defined to be "centered in" a cell if the center of its bounding box is in that cell. With this method, the same list of visible objects can be used for all objects centered in a cell.

After finding an object centered in the current cell, the program defines its search line. In the interactive editor, the search line is defined to start at the user selected hot point, but since there is no user input in batch mode, the program defines the search line to start at the center of the object's bounding box. Thus for the "ON_HORIZONTAL" property, the search line is defined as the line starting at the center of the object's bounding box and going in the negative Z direction. Defining the search line for the "ON_VERTICAL" property is slightly more complicated, since the editor does not know which face of the object is suppose to be against a wall. The editor defines four search rays in the plus and minus X and Y directions, originating at the center of the object's bounding box. The intersection test for a suitable face is performed along the search line or lines. The suitable face selected for snapping the object to is the closest face to the object intersected by the search line(s). If no suitable face is found, the program writes the object's id and an indicator that the object has no suitable face to the output file. If a suitable face is

found, but is too far away for the object to snap, the program writes to the output file the object's id and the distance to the suitable face. *(Note: This occurs when the program is run with the local snapping option. With this option, the object will only snap if it is within a distance equal to half the width of its bounding box in the direction perpendicular to the suitable face).* If the program finds a suitable face that is close enough to snap to, the object is translated so that its bounding box is moved to the intersection point of the suitable face and the search line.

Whenever an object is moved by snapping, to maintain object grouping, all objects that are constrained to that object must also be moved. Thus for each object that is to be snapped, the program must determine which objects are constrained to that object. Since, at the current time, it was considered impractical to change the database to accommodate the links describing the grouping of objects, groups are formed dynamically by a recursive process which determines the suitable surface for each of the visible objects, just as above, then creates a temporary "group" list (a list of the objects whose suitable surface is a face of the original snapping object.) If the created group list is not empty, the routine creates a new group list by processing the visible objects to determine if any object, not already in the group list, is constrained to any of the objects in the group list. This process is repeated with the newly created group list until the newly created group list is empty. Object grouping is maintained by snapping the original object and translating the objects in the group lists by a distance equal to the snapping distance.

8 Updating the Permanent Records

In our Walkthrough system, building models are represented in two forms: a UNIGRAFIX text description and a Walkthrough database format. When the editor is used to make changes to the building model, the changes have to be updated in the files representing the above mentioned formats. The Walkthrough database file is required to be updated, so that, any changes to the building model made in a Walkthrough session or batch routine will be reflected in later Walkthrough sessions of the model. The UNIGRAFIX file is required to be updated, so that, if new UNIGRAFIX object descriptions are to be added to the building model, they can be added to an UNIGRAFIX file that has been modified to represent the edited version of the building model.

8.1 Updating the Walkthrough Database

When the user selects an object for manipulation, a copy of the object is maintained by the editor. All transformations, highlighting, and drawing of the object is handled by the editor. The editor keeps track of the current user operations. When the user ends his/her current sequence of transformations, marked by the selection of another object, the transformations are updated into the database. To do this the editor uses the current 4x4 transformation matrix created from the user's actions. This matrix and the object definition are passed to the database which applies the transformation matrix to the matrix associated with its current copy of the object.

After transformation of an object, the database checks to see in which cell the object is now located, so that the cell information can be updated. If the object remains in its original cell, the cell-to-object visibility of all cells visible from that cell must be updated, since movement within a cell can result in an object becoming visible to some cells and not-visible to others. If the object has moved into a different cell, the cell-to-object visibilities of all cells visible from the object's original cell, as well as the cell-to-object visibilities of all cells visible from the object's new cell must be updated. This additional update is required because the object first has to be removed from any cell-to-object visibility lists that it was previously in and then added to the cell-to-object list of any of the new cells that it is visible to (See [1] for more information).

8.2 Updating the UNIGRAFIX Master File

The UNIGRAFIX *Update* file is a file that represents the difference between the UNIGRAFIX *Master* file, the original building model descriptions, and the edited building model. The Update file contains a list of all changes, in UNIGRAFIX format, that have been made to the building model by the editor. This file is updated by both the batch program and by the interactive editor. The file is composed of a list of tuples that contain the identifier of the object that was transformed and the

transformation matrix that was applied. The batch program writes a tuple to the file whenever it snaps an object to a suitable face or moves an object because of a grouping constraint. The interactive mode writes a tuple to the file whenever a sequence of transformations have been completed. The UNIGRAFIX update file can be used by a post-processing routine to Update the original UNIGRAFIX Master file.

9 Conclusion

As the time test of our editor, we used two existing building models, fully populated with different amounts of office furniture. Some of the furniture was floating or sunken throughout the various rooms. First we ran the batch verification program on the smaller model to correctly position it's objects. 63 objects out of a total of 546 objects were adjusted in 18.96 seconds. Casual visual inspection showed no obviously misplaced objects. Then we repeated the above procedure on the larger model. 1296 objects out of a total of 19209 objects were adjusted in 1767.80 seconds (29 minutes). Without the batch verification program it would take a user many hours of interactive editing to go through the entire model and verify that each object is properly positioned.

This report describes the first implementation of an interactive editor for the Walkthrough program. The whole notion of object properties will require additional research. The property mechanism can be a very powerful tool, with many uses. Eventually the property definition structure should be added to the UNIGRAFIX data structures and handled by the UNIGRAFIX parser. Future Walkthrough editors will need enhancements to increase a user's editing power and to aid in the development of new Walkthrough features. For example, the editor might need features to enhance interaction with animated objects or to specify a time dependent transformation for a particular piece of a movable object. Below are a few remarks on extensions to the idea of placement properties; they are a result of our experience with implementing the two properties ON_HORIZONTAL and ON_VERTICAL.

There are additional properties that seem desirable from a user's point of view. There should be a position property that allows the user to specify that a specific object should be on another specific object, for example, cup A should be on top of table B, instead of simply specifying a cup should be placed on a horizontal surface. We would like a property that allows the user to specify that an object should abuts another object, for example a desk should abuts a wall. It would also be useful to allow the user to define the interaction point of an object, so that the user can define the point around which an object should be rotated.

Of course, the idea of properties goes beyond the specification of placement constraints. As one example, properties could be used to affect the rendering style of objects, i.e., whether the object should be rounded or left in its polyhedral form. Also, it would be useful to be able to make more complex property definitions, such as "picture A should be on a wall and five feet above the floor."

The interactive mode of the editor makes it ease to significantly modify the furniture layout in the various rooms of a large building model. As a result we feel that the editor is a valuable addition to the Walkthrough project. The interaction gives the user more of a feeling that he/she is actually in the building. The editor allows the user to add much needed disorganization to the building in order to create a model that better simulates every day life.

A APPENDIX: Runing the Program

This section is intended for the system operator whose purpose is to install the editor to run with the Walkthrough program. It explains the files needed to initialize the editor and the commands used to run the editor. This section assumes that the operator is familiar with the routines used to create the database.

A.1 Initial Files.

A number of files must exist in order to initialize the program and to assign properties to object types. The property set file is created by the user, while the other files are for internal use by the system and are generated automatically.

- Property set file *(user created)*. To use the property constraints the user must create a property set file *name.pr* which contains the property set definitions and the necessary assignments (See *Property Definitions* section for more details).
- Property file (*automatically created*). This file contains a list of the properties that the editor knows how to handle. This file should already exist and should be named *property.pr (See section on Property Definitions for more details).*
- Object type file (*automatically created*). This file contains the names of the available object types. Its default name is *databasename.ty* . This file is created by the wkadd routine (See section *Property Definitions* for more details).
- Database file (*automatically created*). The database file describes the polygons that compose the building and contains at least one of every type of object that the user wants to have in the final building description *[see Funkhouser for more information of creating the database file]*.

A.2 Running the Interactive Editor

To start the program, the user types "wkedit *database file* [-p *property file*] [-t *type file*]. This opens on the screen the normal GL window used for the Walkthrough program. Along the top of this window is the menu bar which allows the user to open up the various dialogue boxes. In the remainder of the window will appear the objects of the scene visible from the user's current position. The Walkthrough features are accessed in the same way as in the normal Walkthrough program. To activate the editor capabilities, the user selects the toggle button "editor" from the file menu at the top of the screen. When in editing mode, the user can access the Walkthrough operations by the same key and mouse combinations as before. All of the editing operations are accessed either by a combination of the shift key and a mouse button or from the editor menu.

A.3 Running in Batch Mode

To run the batch program, the user types "wkcheck_edit *database file* [-p *property definition file*] [-t *property*] [-local] [-global]. Notice that the same files that are needed to run the interactive program are used to run the batch program. The *-local* or *-global* options, allow the system operator to specify whether objects can snap to a local face or to a global face. Setting snapping to a local face prevents an object from snapping through more than a distance equal to the size of its bounding box in the corresponding direction.

B APPENDIX: References

References

[1] Funkhouser, Thomas A., Séquin, Carlo H. and Teller, Seth J. *Management of Large Amounts of Data in Interactive Building Walkthroughs*. ACM SIGGRAPH Special Issue on 1992 Symposium on Interactive 3D Graphics, 11-20.

[2] Teller, Seth J. and Séquin, Carlo H. *Visibility Preprocessing For Interactive Walkthroughs*. Computer Graphics (Proc. SIGGRAPH '91), volume 25(4), (August 1991), 61-69.

[3] Séquin, Carlo H. *Introduction to the Berkeley UNIGRAFIX Tools (Version 3.0)*. Technical Report UCB/CSD 91/606, Computer Science Department, U.C.Berkeley, 1991.

[4] Conner, D. Brookshire, Snibbe, Scott S., Herndon, Robbins, Daniel C., Zeleznik, Robert C. and van Dam, Andries. *Three-Dimensional Widgets*. Computer Graphics (Proc. SIGGRAPH '92) Computer Science Department, Brown University, Providence, RI.

[5] Bier, Eric A. *Snap-dragging in three dimensions*. Computer Graphics (Proc. SIGGRAPH '90), volume 24(4), (March 1990), 193-204.

[6] Strauss, Paul S. and Carey, Rikk. *An Object-Oriented 3D Graphics Toolkit* Computer Graphics (Proc. SIGGRAPH '92), volume 26(2), (July 1992), Silicon Graphics Computer System, Mountain View, CA.

[7] Khorramabadi, Delnaz A. *A Walk through the Planned CS Building*. Masters Thesis UCB/CSD 91/652, Computer Science Department, U.C.Berkeley, 1991.