The WALKTHRU Editor: Towards Realistic and Effective Interaction with Virtual Building Environments

Richard Bukowski

Master's Project
under the direction of Carlo Séquin

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

November 2, 1995

Abstract

This thesis describes the development of WALKEDIT, an object placement editor for the Berkeley architectural WALKTHRU system. In addition to incorporating editing operations commonly found in 2D and 3D model editors, two new major results were achieved.

First, a system for simple and natural direct manipulation of 3D objects was created. This system, which we call Object Associations, is a software framework that provides a unified method for designing and implementing convenient direct manipulation behaviors for objects in a 3D virtual environment. A combination of nearly realistic pseudophysical behavior and idealized goal-oriented properties is used to disambiguate 2D mouse actions on the display screen into appropriate and natural object motion in the 3D virtual world, and to determine valid and desirable final locations for objects being manipulated. Objects selected for relocation actively look for nearby objects or structures to associate and align themselves with. An automated implicit grouping mechanism falls out of this process. Concept, structure, and our implementation of this framework are presented.

Second, the realism of the WALKTHRU real-time rendering system was enhanced by the addition of physical simulation software. A first set of routines provides the virtual user with an adjustable eye height, which is dynamically maintained through a combination of feedback control techniques and discrete time physical simulation. This gives a much more natural feeling to moving through the building, allowing the user to look up and down while walking, and permitting realistic use of stairs and elevators. In a separate experiment, the Lin-Canny closest features algorithm and a fast contact force computation algorithm was integrated with the object associations system, allowing true collision detection and pseudo-static simulation of moving objects. The implementation, tradeoffs, and success of this experiment are presented and discussed.

Contents

1 Introduction

> Easy to use, comprehensive modeling systems remain elusive and difficult to devise. In fact, the modeling problem provides many more difficulties than the rendering problem. People have frequently asked me "How long did it take to render the picture?" Rarely do questions relating to the duration of modeling tasks arise.
>
> *Donald Greenberg, Jan. 1991* [26]

Berkeley's building walkthrough program (WALKTHRU) was intended to provide the user with the ability to move in real time through models of large buildings fully populated with furniture. By 1991, that goal had been realized; however, the conceptual design of WALKTHRU did not include techniques for creating the building model in the first place. The modeling process for our one base model, that of Berkeley's new Soda Hall, was a long and difficult one, requiring months of manual calculations to determine precise floating point locations and dimensions for walls and furniture. These building elements were entered into text files by hand, compiled with a UniGrafix compiler, pre-processed with the visibility routines, and, finally, displayed on the screen. Often, the result contained poorly placed walls and objects that intersected each other or floated in space. These problems were resolved by manually estimating the errors in object positions and correcting the numerical offsets in the text file. The entire hours-long recompilation process would then be repeated, after which more alignment problems would inevitably be visible, requiring further iterations of the modify/recompile process.

Given the level of modern interactive technology, there is no reason why model construction should be such an arduous text-based process. This thesis describes the development of WALKEDIT, an interactive editor for the WALKTHRU system which allows users of the WALKTHRU to manipulate the contents of building models interactively within the virtual building environment. The WALKEDIT project began several years ago with an effort by Thurman Brown to add standard editing operations to the WALKTHRU environment [11]. Since then, many new techniques have been added that allow the user to pick up, move, copy, and otherwise manipulate furniture and other objects interactively and intuitively from within the real-time rendering environment used in the original WALKTHRU application [23]. Such interaction is facilitated by a newly developed 3D manipulation paradigm, called *object associations*, which provides intuitive object behaviors. Under the influence of object associations, books and cups move in tandem with the desks they are stacked on, furniture slides along the floor rather than floating through the air, and wall hangings re-orient themselves as they move to lie flush against supporting walls.

User motion in WALKEDIT has been modified so that the user's "virtual body" behaves in a more physically realistic fashion. The virtual body obeys a set of pseudo-physical rules that simulate normal movement through buildings, allowing the user to walk up and down staircases and use elevators naturally. This has been found to contribute a great deal to the realism of the walkthrough experience. We are also experimenting with further enhancements such as dynamic simulation and solid-object interactions which will further enhance the realism and usability of the user interface.

2 Background and Previous Work

Creating a fully equipped and accurate model of a 3D environment for any purpose (graphics, CAD, modeling, etc.) is an arduous task. Even assuming the availability of a good interactive 3D geometry editor with a friendly and efficient user interface, such tasks are inherently much more difficult than drafting and editing in only two dimensions. In the first portion of this section, we will discuss the challenges inherent in the 3D direct manipulation task with modern computer hardware, and approaches that have been taken in the past to meet those challenges. Later, we will focus on our chosen problem domain, that of manipulation of furnishings in an architectural environment.

2.1 General 3D Interactive Direct Manipulation Techniques

2.1.1 3D Hardware

A seemingly obvious solution to the problems of 3D direct manipulation is the use of 3D displays and 6 degree of freedom (DOF) input devices. This approach eliminates the mismatch between the dimensionality of popular display and input technology (2D screens and mice) and the dimensionality of the virtual environment. Unfortunately, at their present level of development, these technologies do not yet provide a cost-effective and practical solution to the 3D direct manipulation task.

Current 3D display technology tends to rely on CRT-based "shutter goggles" or head mounted stereo displays. Shutter goggles provide almost no sense of immersion in the virtual world, and give no immediate perceptual benefit because the user still cannot move his or her head to realistically examine the object from different angles. Affordable head-mounted displays are slow and have poor resolution and field of view for their proximity to the eye. Tracker radius limitations, wiring and cords, bulky and heavy equipment, and physical interference of the user's real body with the objects in the real world make them physically cumbersome.

Most 6 DOF input devices such as the "SpaceBall" [5], "DataGlove," or 3D mice [47] are awkward, tiring to use for an extended period of time, and too expensive for the typical user. They tend to have much lower sampling rates and far higher noise and jitter than 2D devices, due to the limitations of today's 3D tracking technology. Even more fundamentally, they fail to solve the problems of 3D placement. Precise placement of objects in three dimensions is hard – even in the real world – unless we get help from the physical or conceptual interactions of the objects we want to place with their surroundings. Consider positioning a picture frame one millimeter in front of a wall without touching the wall with the frame or with your hands; visual feedback alone cannot do a satisfactory job. With a noisy, slow, inaccurate tracker and a picture frame that will sink into the wall surface with no tactile feedback, the user can quickly become frustrated.

2.1.2 On-Screen Widgets

Due to the limitations of 3D displays and 6 DOF devices, a great deal of research has been directed toward software-based techniques for manipulating 3D objects with standard, low-cost 2D input and output devices. The challenge in this approach is to design a method for controlling all six DOF of a 3D object with only 2D user input.

Most 3D direct manipulation solutions implicitly or explicitly assume the use of 3D widget sets. A widget is an auxiliary graphical construct, such as a set of axis-aligned arrow handles or a rotation sphere, which is added to the on-screen display of the object (see figure 2 for some examples of 3D widgets). This construct is directly manipulated with the mouse to move the object in a clearly constrained way specific to the widget type. Neilson and Olsen provide an overview of many of the earlier widget-based techniques such as transformation and rotation handles [37]. Later work either improved the widgets themselves, such as the ARCBALL work which removed the hysteresis from the earlier "crystal ball" manipulator [39], or explored new, specialized 3D widget types, such as recent work at Brown on 3D widget classes based on deformation and constraint linkages [51, 41]. Most commercial direct manipulation systems that are not true CAD systems, such as Silicon Graphics' Inventor modeling system [42], use 3D widgets to provide their direct manipulation capabilities.

The tendency of widgets to dominate 3D direct manipulation seems to be a legacy of the older 2D direct manipulation problem, for which we have an established and proven set of 2D constraints and widgets that provide easy-to-use and easy-to-generate 2D direct manipulation interfaces [17, 25]. The implicit assumption is that the techniques that worked in 2D will work equally well in 3D. Chen's work in 1988 supports this idea by showing that a combination of 2D input devices with these 3D widgets is no less accurate or harder to use than modern 3D, 6 DOF input devices for most 3D manipulation tasks [14]. However, even the latest work concedes that 3D widgets have a long way to go before a 3D widget set in a 3D virtual environment can rival the completeness of a 2D UI toolkit and direct manipulation widgets in a 2D application domain [41]. Standard 3D widgets perform well if the manipulation tasks are geometrically simple, involving only translations along or rotations around clearly defined axes, such as local or global coordinate axes. If more complex actions are required, widgets can be cumbersome: either the widget set is simple, requiring the user to use many widgets sequentially or simultaneously to perform the task, or the widget set is large and complex, requiring the user to find and select, from a set of menus or lists, the widget or widgets that are appropriate to a given situation. Widget sets can also inherit some of the same problems as 6 DOF input devices; they often fail to consider interactions between objects when the user is performing manipulations, or when they do, they are over-generalized, requiring use of multiple widgets and actions for conceptually simple manipulations. In an attempt to address these shortcomings, two other major categories of direct manipulation techniques have been explored: constraint based systems and physics/dynamics based systems.

2.1.3 Constraint Systems

Constraints are a second lasting theme in manipulation design. Whereas widgets attempt to disambiguate 2D mouse motion to 3D motion by forcing the user to point to auxiliary "handles" on the object, constraints attempt to perform the disambiguation by allowing the user to specify, at the object or object class level, which degrees of freedom the user is permitted to manipulate, and how changes in those degrees of freedom will alter the other, constrained degrees of freedom. This allows the system or the user to identify how a particular object can move without attaching widgets to it. Furthermore, if the constraints are properly specified, they automatically take care of local alignments that 6 DOF manipulators and most 3D widgets fail to deal with. Pictures whose backs are constrained to lie in the plane of the wall and whose angle is constrained to be vertical are always properly aligned in those dimensions; the user can manipulate the pictures' 2D position in the only "real" degrees of freedom, those parallel to the wall plane.

The two major problems with this approach have been well documented. First, constraint systems can be difficult to solve at interactive rates, forcing the designers to use limited solution techniques and constraint sets that provide less functionality than one might want. Work on improving the techniques behind constraint solutions is an open mathematical problem [28, 4, 9], one that is outside the scope of this thesis.

Second, it is an onerous task for the user to manually specify and maintain the constraint networks needed for direct manipulation [35, 31]. Just the internal constraint structure needed to create a cube contains 24 distinct constraint relationships; the number of constraints necessary to specify the relationships between the objects in an average office can be large, and the user may have to make and break them individually in order to move objects. Some work has attempted to address this problem. Brad Myers attempts to solve the constraint maintenance problem by having the system try to guess which constraints the user wishes to impose; ideally, this reduces the problem to a matter of the user saying "yes" or "no" to the computer's guesses [35]. Unfortunately, answering the computers' constant requests for confirmation can still be annoying, and the system will inevitably miss some constraints, which sends the user back to manual entry. Kurlander takes a different approach, allowing the user to feed the system "snapshots" of the configuration of the objects to be manipulated, from which the system infers and applies all possible constraints that are not invalidated by any snapshot [31]. Myers' approach is prone to underestimation, if the computer doesn't guess a constraint; in this case, the user must go back and explicitly add it. Likewise, Kurlander goes to the other extreme and overestimates. With the first snapshot, the system is constrained completely; if the user wishes to break a constraint, he or she must deactivate all constraints, go to a CAD mode, move into the desired alignment, take another snapshot to give the system an example of what it can do, and re-activate the engine. It is probably impossible to create a system that will automatically determine and maintain all constraints for the user in all situations; these systems, as well as our own object associations system, attempt to make the best possible guesses to minimize the amount of dialogue necessary for the user to correct the computer's assumptions and guesses.

2.1.4 Physics and Dynamics Systems

The third major theme in manipulation is the addition of partial or complete physics simulations to force objects to behave "properly" when pushed, pulled, or otherwise manipulated. These techniques attempt to take advantage of the user's real-world intuition about what an object will do when the user's virtual "hand" applies a force to the object. Some interesting work has been done to determine which aspects of the real world are the most helpful to direct manipulation in virtual environments. Smith identifies the balance of functionality between physical and "magic" (non-physical) behavior in Xerox PARC's Virtual Reality Kit [40]. He notes that real-world behavior comes for "free" in a UI, in that the user is familiar with physical behavior from real life and needs very little training to use it well in a virtual environment. More virtual, "magic" tools, such as Eric Bier's snap-dragging [7], come with a learning curve, requiring users to familiarize themselves with a new paradigm for interacting with objects.

Figueiredo identifies fast collision detection as a critical physical aspect of intuitive interaction with virtual objects [21], presenting some usability results for various techniques of interacting with objects using collision detection in toy environments. However, it is important that any direct manipulation scheme address the *balance* between physical simulation and magic approaches like snap-dragging. A purely physical system can force the user to go through unnecessary contortions to accomplish a task that can easily be expressed by a simple, nonphysical constraint: an example is the Zashiki-Warashi system by Yoshimura [50], which allows users to arrange furniture by "dropping" items with a gravity simulation. The approach worked well for tables, chairs, and books, but made it difficult to affix pictures or shelves to walls. Physical simulation *augments* other methods, it does not replace them.

A major problem with physical or dynamics-based systems is that they tend to be limited to small environments (one room or a few objects) in order to maintain interactive speeds. The relationship of usability to speed is well established [49, 23]; in order to maintain usable frame rates for interaction, the fastest of modern CPUs and most advanced simulation techniques must be used. It is only recently that available hardware and software components have become powerful enough to support dynamics in extended interactive models [16, 19].

2.1.5 Combined Techniques and Virtual Environment Frameworks

Some approaches combine subsets of these three basic techniques to provide a more complete, structured direct manipulation system. For example, Van Emmerik combines CAD views with both constraints and "jacks," a type of 3D widget, to provide a complete solution for 6 DOF positioning via direct manipulation [45, 46]. Bier also combines constraints and snapping behavior with 3D widgets to provide an alignment solution for 3D direct manipulation [7]. Both methods have the advantage of not only providing the user the ability to manipulate 3D objects with the 2D mouse, but also permitting precise relative positioning by aligning lines, planes, and other geometries more easily than a pure constraint system. However, they also have disadvantages: in Van Emmerik's system, a CAD-style orthographic display is necessary, making it unclear how to generalize his techniques to a perspective-window virtual environment, and both systems suffer from a profusion of modes and auxiliary jacks that the user must be familiar with in order to perform basic manipulation tasks.

Other 3D direct manipulation design efforts concentrate on providing VR frameworks within which direct manipulation techniques can be embedded in a reasonable fashion. Unfortunately, most non-trivial VR systems so far have either no interaction methods at all (such as our basic WALKTHRU [18, 23, 44], the UNC walkthrough [1, 2, 10], or the various commercial walkthrough systems [48, 6]) or have an "open interface" that *supports* attaching widgets to them, without many widgets actually *implemented* in a nontrivial context [38, 15, 19]. A virtual environment system that fully supports direct manipulation in a nontrivial world has yet to be presented.

2.2 Applications of Direct Manipulation to Architectural Environments

The body of work on direct manipulation of mechanical or architectural models is relatively limited. We do not consider CAD environments such as AutoCAD [3] to have direct manipulation interfaces; such tools are based on manipulation via numerical entry or operations such as offset vectors and constraint alignments, and make no pretense of offering natural manipulation of building contents or structure.

Architectural environments offer a number of contextual advantages to direct manipulation. Objects in a building tend to have a very specific, context- and object-dependent set of "valid" positions in the environment. For example, a picture never floats freely in space, nor does it rest sticking 90 degrees out into the room; it rests with its back flush against some wall surface. Likewise, cups, books, and other common objects always sit on a supporting surface. Users have shown a marked tendency to be comfortable with restrictions placed on 3D manipulation based on how an object "should" behave [29]. Houde's user interface study, for example, describes users who, when told that they could not lay a chair on its side, were undisturbed, responding with the phrase "well, it *is* a chair." Taking advantage of these *expected* properties can allow the system to better disambiguate user intentions, better anticipate user needs, and limit the command set to something easier for the user to deal with.

One example of an applied system is Gleicher's Briar system [24], which provides an intuitive method for "rubberbanding" objects together in natural ways. Unfortunately, Briar requires extensive descriptions of how the objects work before they can be manipulated, and those descriptions may only be defined in terms of a limited set of linear constraints . Briar can do some interesting things with well-defined, simple objects like gooseneck lamps; however, movements of objects such as chairs and cups, which normally involve plane-to-plane constraints that change depending on when and how they are moved from place to place, are beyond the capabilities of the system to adapt to new constraint types. Some systems have gotten around this problem by going to physical simulation, such as the Zashiki-Warashi "virtual room construction" software [50], which provides the user with a 3D, 6 DOF tracked wand pointer with which objects can be "picked up," then dropped under gravitic force, which enforces a "reasonable" resting pose for the object. Unfortunately, their gravity-only system fails for such objects as light fixtures (which are supposed to sit on the ceiling, not the floor) or pictures (likewise, should sit on the wall). In such cases, the system forces the user to go back to a CAD style, non-direct manipulation interface.

Houde at Apple's user interface research group provides perhaps the most easy-to-use approach [29], one which to some degree parallels our object association system. Houde does a series of user interface studies which attempt to develop a manipulation technique suitable for arranging furniture in virtual rooms. They identify a number of ease-of-use properties for manipulating objects and furniture in a virtual room, including:

1. Most objects want to move in some given plane, such as tables and chairs (the floor plane) or pictures (the wall plane).

2. Most interactions with furniture are of 3 types: *translation* along the aforementioned plane, *orientation* (rotating about the normal of the plane), and *lifting* (for stacking objects or placing them atop one another).

3. "Creating a composite mode for allowing easy access to rotation [about the plane normal] and translation [in the plane] allows users to make repetitive position adjustments in a smooth manner." [29]

4. "Reducing the number of possible degrees of 3-D manipulation freedom via context specific constraints, contributes to ease of use and a user's feeling of control in a 3-D environment." [29]

Houde uses a bounding box with attached "narrative handles" (iconic handles with shapes that represent the handle's action) as a 3D widget. The user manipulates the object via these handles, implicitly selecting the manipulation mode based on the chosen handle. Houde's results mirror some of our results from the object associations research. We have extended the research by devising new techniques that build on these precepts. She did not deal with objects other than simple, on-a-surface gravitic objects, and glosses over how the user would stack up objects or move piles of objects. We have solved those problems, as well as providing a more flexible and complete framework for development of additional context-specific direct manipulation properties.

Finally, note that none of the systems mentioned here have been used with more than "toy" environments (e.g. a "small" collection of objects or a single room). The physical simulation systems slow down when dealing with more than a handful of objects, and the other systems have simply not been tested in a large environment. One of the important aspects of the WALKEDIT project is the fact that we are manipulating a database built from the specifications of a real building, so our results prove the applicability of our techniques to "real-world" manipulation problems, as well as demonstrating their robustness in the context of a very large architectural model that contains 2.2 million polygons and 14 thousand objects.

2.3 The Walkthrough Editor Prototype

Thurman Brown wrote the first prototype of WALKEDIT as a masters' project. The prototype attempted to provide basic techniques for direct manipulation of detail objects in the WALKTHRU model; however, it ended up raising as many issues as it solved. Some of the more complex operations were not fully implemented, and the initial approach to a user interface proved less than satisfactory. The simple transformations necessary for the basic editing operations worked, but the constraint system operated neither properly nor intuitively. Often, objects would move in an unpredictable manner, and in many cases, detailed knowledge and experience with the system was required in order to perform simple tasks like stacking books on a desk.

This section discusses the capabilities and shortcomings of the original system; it is provided as a basis for the discussion of the new solutions developed in the current WALKEDIT environment.

2.3.1 Selection

An object must be selected before it can be manipulated. When the user holds down the shift key, a white bordered bounding box is dynamically generated around the current object at which the mouse pointer is pointing. This box indicates exactly which object will be chosen if the selection button is pressed; for selecting very small objects or a single object in the midst of a large number of other objects, this is a very helpful feature.

The user selects an object by shift-left-clicking on it in the view window. The selection point on the object is highlighted with a small octahedron (chosen for its visibility from any angle), and the bounding box becomes black. If the user re-grasps the object via another shift-click, the selection point is changed to the newly indicated point. Multiple selection is implemented via the alt key, which is used in the same fashion as the shift key. However, shift selection implicitly deselects any other selected object or objects. Alt selection simply toggles the state of the object between "selected" and "not selected." This design allows the user to select sets of objects. When the user initiates a manipulation, the last selected object that was clicked on is the "focal" object; the selection point for the group is defined as that of the focal object (Figure 1).

2.3.2 Transformations

In the WALKEDIT prototype, three special-purpose modes for rigid body transformations (planar translation along a major axis plane, a "crystal ball" style rotation mode, and a scaling mode; see Figure 2) complement a general transformation mode. In each of the special modes, a specific constraint is used to map 2D mouse motion on the screen to 3D object motion in model space. The *major axis translation*

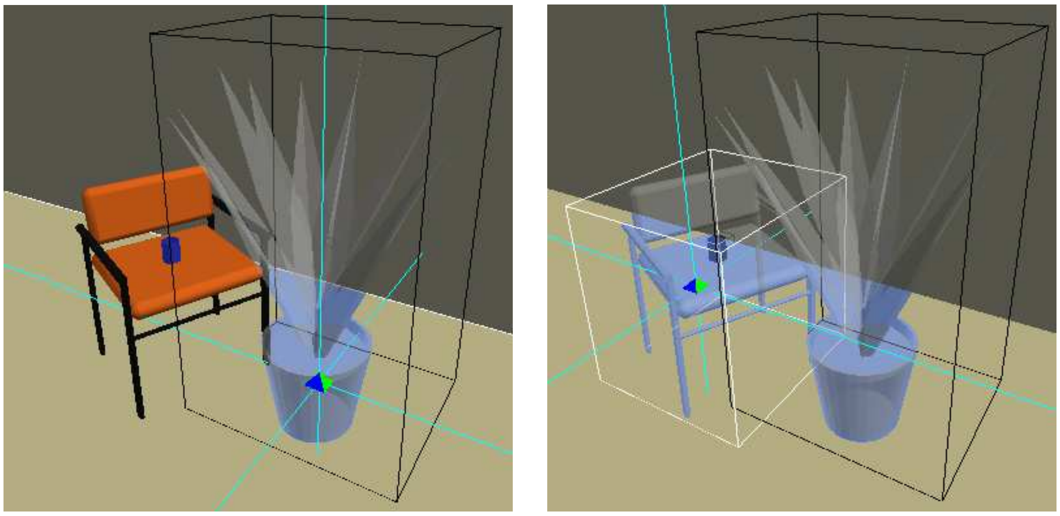

Figure 1: Selection of single or multiple objects. On the left, the plant has been selected; note the octahedron indicating the selection point and the axis-aligned hint lines. On the right, the chair has been alt-clicked to add it to the selected group; the chair is now the focal object carrying the selection point.

mode maps mouse motion on the screen to motion in a selected major axis plane by projecting a ray from the eye point through the mouse pointer into the 3D world. This ray is intersected with a constructed plane parallel to the selected major axis plane passing through the initial selection point, and the object is moved such that the selection point is brought into coincidence with the intersection. The resulting object transformation satisfies two properties: the object translation vector lies within the axis plane, and the selection point on the object will always remain coincident with the mouse pointer (Figure 3). The two other special transformation modes are somewhat simpler: the *crystal ball rotation* mode maps X and Y motion of the mouse to theta and phi rotation of the object, and the *scaling* mode maps X motion of the mouse to scaling of the object.

In the general transformation mode, the system applies special motion constraints dynamically depending on which mouse button is used to drag the object. These “constraints” are not mathematical, but

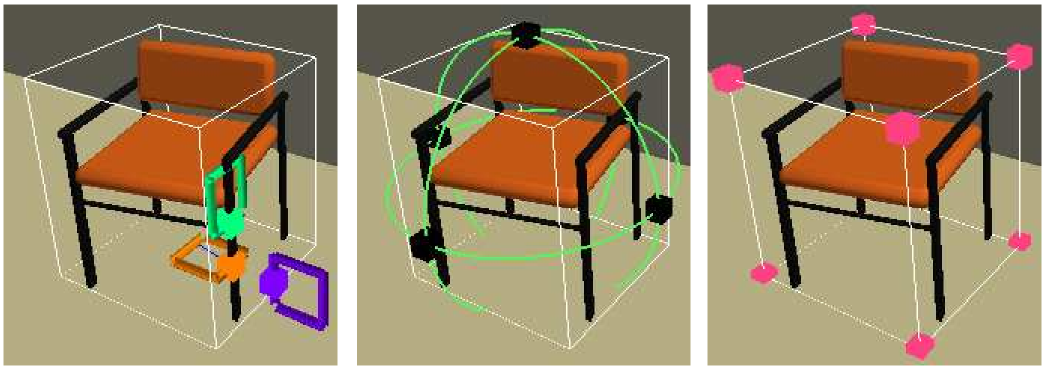

Figure 2: 3D manipulation mode widgets available in WALKEDIT. On the left, the *major axis translation* mode handles are visible on the chair; in the middle, the *crystal ball rotation* mode handles are shown; and on the right, the *scaling* mode handles are applied.

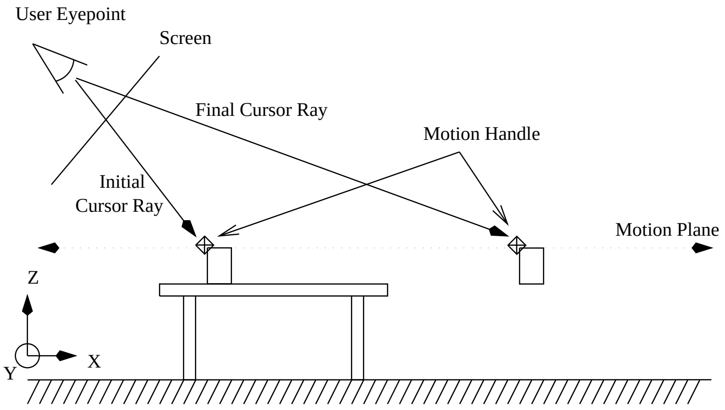

Figure 3: A schematic drawing showing how a cup would move under the major axis translation mode. The cup is grabbed by the XY-plane motion handle; the system moves the cup parallel to the XY axis plane to maintain the coincidence of the handle with the mouse pointer ray. Note that interaction of the cup with other objects (i.e. the table) does not affect the motion.

functional in nature; they are intended to simplify manipulation of specific types of objects. The prototype implemented two special constraints: one called *on-horizontal*, which was designed for objects that normally sit on the floor or on other objects, and *on-vertical*, designed for objects that hang on wall surfaces. Under one of these special constraints, when the object is first selected, a *motion plane* is established through the selection point. The motion plane is either a horizontal plane parallel to the floor (in the case of on-horizontal) or a vertical plane parallel to a nearby wall (in the case of on-vertical). The constraint forces the object to move in the motion plane in the same way that the major axis translation mode moves the object in a selected major axis plane. However, the special constraint will also move the object *out* of the motion plane under certain circumstances. In on-horizontal, if the object's translation will move it such that there is no supporting surface under the object, the special constraint will add a motion straight up or down so that the object "rests" on some nearby support surface (Figure 4). In on-vertical, if the translation will move the object through a corner where two walls meet, the special constraint will rotate the motion plane to be parallel to the new wall, and rotate the object so that its back remains parallel to the motion plane (Figure 5). These special constraints can automatically provide desirable alignments that would ordinarily have to be applied manually: objects like desks and books moved under on-horizontal will always sit on a support, and pictures moved under on-vertical will rotate properly when moved from one wall to another. Unfortunately, the special constraints suffered from some problems. The rules used by on-horizontal to determine whether an object should move up or down were unintuitive and buggy, making it difficult to get objects to align properly. On-vertical was never made to work in the prototype, so testing was difficult. Finally, the routines were hard-coded into the system and the menus, making it difficult to add or change them.

A simple automated grouping mechanism accompanied the special motion constraints. When applied to an object, on-horizontal returned the object identifier for the supporting surface on which the selected object came to rest. When the user selected an object in general transformation mode, the system would search local space, applying one of the special motion constraints to the objects in the vicinity, and try to determine which local objects were "sitting on" the selected object. All such objects were then grouped with the selected object and moved with it (Figure 6).

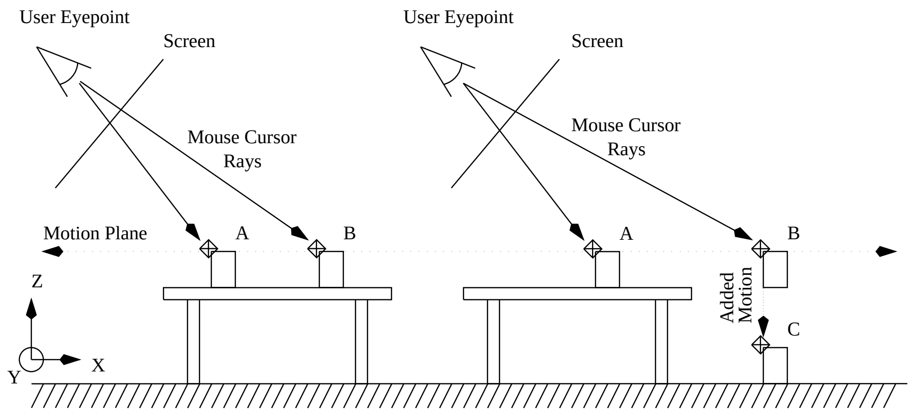

Figure 4: The *on-horizontal* general transformation mode. On the left, a cup on a table is selected and moved along the surface from point A to point B by dragging the mouse cursor up on the screen. The motion plane is shown as a dotted line. On the right, the cup moves from point A over the edge of the table (point B); the mode adds a vertical component to the cup's position to make it "fall" to the floor (point C).

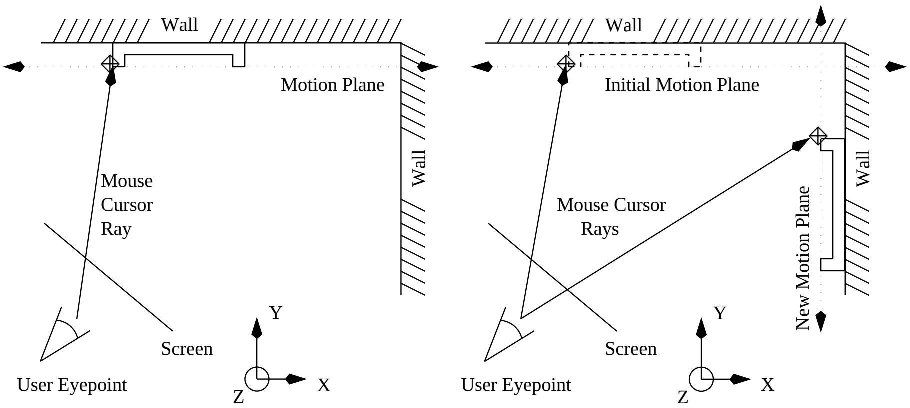

Figure 5: The *on-vertical* general transformation mode. On the left, the user selects a picture; the system defines a motion plane parallel to the nearest wall. On the right, the user moves the picture around a corner; the picture rotates and a new motion plane is established on the new wall.

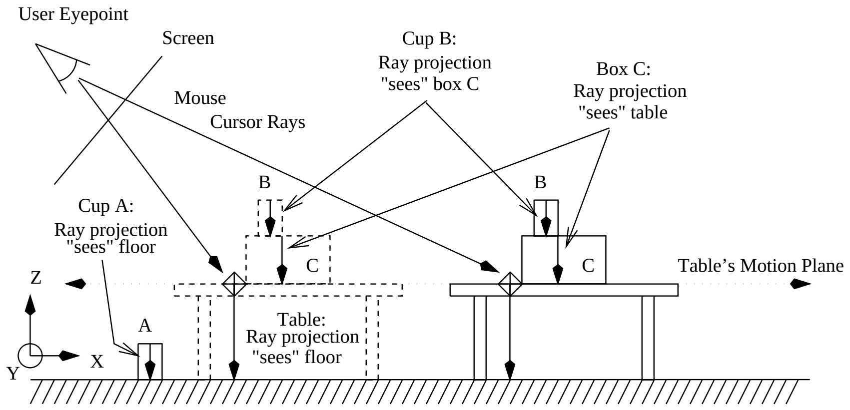

Figure 6: Automatic grouping process. When the on-horizontal procedure projects a ray from cup A downward, it hits the floor; when the procedure projects a ray from box C downward, it hits the tabletop; when the procedure projects a ray from cup B downward, it hits box C. Thus, when the table is moved, cup B is grouped with box C, and box C is grouped with the table; box C and cup B move with the table, while cup A remains stationary.

2.3.3 Menu Operations

The prototype editor implemented several other operations, available from pulldown menus. The user could toggle between selecting objects by face intersections or bounding box intersections; the latter could be used to increase interaction speed. Simplified representations of the selected object, from transparent (the optimal, but most expensive rendering technique) to wireframe to bounding box, allowed the user to adapt the display for slower computers. The user could elect to have "hint lines" drawn from the selection point parallel to each of the major axes; these provide positioning feedback while the object is being moved. There is a toggle to select whether or not the scaling mode is available; normally scaling is disabled, since it is neither a rigid body transformation, nor is it a "normal" operation to perform when manipulating furniture in the real world. For exact positioning operations, there is a dialog that allows the object to be moved numerically by entering offset values from its current position.

In the prototype, there was a copy button for making multiple copies of an object. The user would select an object and press "copy"; the next time the object was moved, a copy of the object would be left behind at the location where the copy button was pressed. Both the original and the copy would then be available for transformations. Also, a simple undo function was provided that allowed the user to undo the last rigid body transformation.

2.3.4 Deficiencies in the Prototype Editor

In many ways, the prototype editor raised more questions than it solved. The general transformation mode was buggy and quirky; it wasn't clear when objects should move out of their major plane of motion, so on-horizontal would often unpredictably stack or unstack objects during manipulation. There was no provision for long-distance motion of objects. In order to move a chair from one side of the building to the other, the user had to slide it step by step to the door, down the hallway, and into the new room, repositioning the eyepoint at every step. There was literally no way to move objects between floors without entering a numerical offset for the object in the "explicit transformation" dialog box. The grouping mechanism unnecessarily duplicated a large amount of computation, did not cache work from selection to selection, and was implemented in a batch mode that forced the user to wait for the process to finish before the object could be moved, which was often frustratingly long in heavily populated areas of the model. The copy-in-place scheme caused a number of problems: besides being difficult and error-prone to execute, it created problems with the grouping mechanism (if you copy a desk with a cup on it in place, which copy of the desk should the cup be grouped with?) and the selection mechanism (the copy begins exactly aligned with the original; how can you point to one or the other? Furthermore, it is not obvious that there are two objects in that position). The undo system was incomplete: it could only undo a single button-down-and-drag operation, which was usually inadequate. The prototype was missing provisions for writing changes in the run-time model back to the original, ASCII-based UniGrafix database from which the model was constructed. This rendered the prototype essentially useless as a real tool, since any changes made with the editor could not be propagated back to the building model. The prototype focussed our attention on the crucial issues that would make such an editor a practical and easy-to-use too; a great deal of both research and programming work had yet to be done.

3 The New Walkthrough Editor

Based on the lessons learned from the original prototype, a new editor was developed. We had three major goals for our second generation WALKEDIT. First, we wanted to complete and debug all of the functions of the prototype. Second, we wished to add new capabilities to make the system more useful and intuitive. Third, we wanted to increase the realism of the walkthrough experience. The following sections present the new methods and algorithms that have been implemented to enhance the overall functionality of the system, as well as the tradeoffs that were explored in the transition from the old prototype to the new, improved walkthrough editor.

3.1 The General Transformation Mode

3.1.1 Motivation for Having a General Transformation Mode

Most of the editing work is done in the general transformation mode. In the early stages of development, we decided to make the interface as non-modal as possible so that the system would be easier to use. The advantages of choosing a non-modal over a modal interface are twofold: it can provide fewer commands for the user to remember, and requires less physical activity (button presses, menu selections, etc.) to execute those commands. However, designing an intuitive non-modal interface can be more difficult, because a non-modal system must be more carefully designed to disambiguate a more terse form of user input.

In the process of developing this non-modal "general transformation mode," we examined different methods for helping the user to move objects in 3D with 2D input devices. We wanted the process of moving furniture in a 3D virtual environment to be as quick and easy as moving cut-out cardboard pieces on a floorplan. However, it should also be possible to force objects to align themselves nicely to walls and to one another, if the operator chooses such an option. Our goal was to bring as much of this capability into the general transformation mode as possible, making it a straightforward task to do the most common alignments with simple, single mouse actions (such as left-click-and-drag). We accept the possibility that the general mode may not be able to handle all cases well; however, if we can encapsulate the *majority* of actions the user would wish to perform in the general mode, we can succeed in reducing the effort required to perform modeling tasks by a large percentage. The remaining minority of operations can be handled by more traditional, CAD-style object manipulation tools.

3.1.2 Properties of a Desirable General Transformation Mechanism

Balancing Realism with Goal-Oriented Behavior

There are many solutions to the 3D direct manipulation problem in the literature. Unfortunately, these solutions often overlook an important aspect of the problem: the tradeoff between *physical* object behavior and *teleological* (goal-oriented) object behavior in the user interface. Previous work tends to focus on one "end" to the exclusion of the other (generalized/nonphysical vs. constrained/physical).

Physical behavior involves properties like gravity and solidity, which, through our experiences in the real world, help us disambiguate the location of an object in a 2D projection without complete 3D information. For example, if we see the top of a table in a floorplan, we know the exact position of the table in the real world because, physically, the table must sit squarely on the floor. If you want to position a real table in a real room, you don't worry about pushing down on the surface of the table to force it to sit on the floor; you rely on gravity to handle that DOF and only push the table sideways. If a virtual environment does not include this property, we have unnecessarily added a DOF that the user must explicitly control. Approaches like snap-dragging provide more degrees of freedom and more generality of motion than necessary for the "task at hand;" controlling these extra DOFs forces unnecessary cognitive overhead onto the user (aligning, setting up tugboats and orientation frames, grouping objects by hand, etc). This is a consequence of treating objects as wholly nonphysical geometries. We would like to build useful physical properties into the objects we are manipulating. Note that the properties are primarily local; they come from interactions with other objects in the proximity of the object being moved, or from natural effects like gravity which do not involve other objects. Ideally, a 3D manipulation paradigm would allow us to take advantage of these expected properties to avoid unnecessary user control overhead.

Often, however, there is a downside to enforcing physical properties. They may remove certain unnecessary DOFs from user control, but they simultaneously limit the generality of the motions allowed. If the user wishes to place a virtual grand piano in a virtual room, they shouldn't have to worry about fitting the piano through the doorway. In such cases, we may wish to impose *teleological* rather than physical behaviors on objects. These behaviors treat the object as merely a geometric entity, providing actions which are convenient but not necessarily realistic. This can make certain operations much simpler; we don't have to worry about being strong enough to move an object, we can introduce snapping behavior to force objects to align, and so on. On the other hand, if object behavior becomes too nonphysical, we may begin making life more difficult instead of less. Physical simulation based methods such as the Zashiki-Warashi system provide automated alignment (a table aligns with the floor via forces/torques caused by its mass), automatic grouping (objects on the table stay on it while the table is moving because of friction), and other effects that make objects behave in expected ways. However, they make other, conceptually simple operations difficult or impossible. For example, the Zashiki-Warashi system does not allow pictures and shelves to be moved along or attached to walls, since the physical simulation doesn't allow unsupported objects, and modeling nails or hooks is beyond the system's capabilities. Even if nails were modeled, there is no reason to force the user to go through elaborately realistic procedures such as affixing objects with nails when a simple, "magical" "glue-surfaces-together" operation solves the problem elegantly.

Hence, overly generalized mechanisms lose the intuitive and easy positioning mechanisms and disambiguating knowledge we have all learned since we were children; overly physical systems bring those mechanisms back, but force us to deal with the problems of the real world that we are using a computer to get away from. We would like to try for the best of both worlds: a mechanism that will provide the structure for, and encourage the interface programmer to provide, the proper tension between realism and virtual-world magic [40].

Applying the Physical/Teleological Balance in WALKEDIT

In WALKEDIT, we are primarily concerned with keeping objects supported against gravity, having them attached to – and thus properly aligned with – the ceiling, walls, or vertical surfaces of other objects, or having objects aligned with respect to each other. All this can be achieved with a remarkably small set of physical and pseudophysical paradigms of motion. Some of the key paradigms of 3D manipulation and some of the behavioral aspects of objects in a building that we found desirable when populating our Soda Hall model with furniture are summarized below:

- User-selected objects should follow the mouse pointer, so that "point and place" becomes an integral, intuitive operation.
- Objects typically should not float in mid-air but rest on some supporting surface. If the cursor points to the surface of a desk or to a bookshelf, it can be implied that the user wants to move the selected object to that particular surface.
- Alternatively, many things, such as picture frames or light fixtures are attached to walls or other vertical surfaces.
- Such implicit associations of objects with reference objects should be maintained even when the reference object moves or is changed in other ways; however, they must also be breakable so that objects can be lifted off a surface easily and moved somewhere else.

On the software engineering side, other key aspects of a general transformation mode are:

- Ease of use: it should be easy for the user to assign behaviors to objects and classes of objects. Furthermore, it should be easy for the programmer or user to create new behaviors.
- Flexibility: Given the description of a behavior, it should be nearly as easy to encode that behavior as it is to describe it.
- Generality: Any desired behavior should be representable.

3.1.3 Desired Manipulation Process in the General Transformation Mode

The generic editing operation in an interactive environment is to "grab" an object and then to "place" it (and any objects grouped with it) somewhere else. In WALKEDIT, a user selecting an object explicitly chooses both the object itself and a *selection point* on the object which makes a natural *handle* for object manipulation. Once the object and its selection point have been established, the user can apply either a *local* motion by dragging the mouse pointer, or a *remote* operation such as "picking up" the object and "placing" it at a different location.

We find it useful to match the manipulation mechanism to the human's natural tendency to use a combination of gross and fine positioning. A human in the real world follows three basic phases when repositioning an object. First, the object is picked up or grabbed by some convenient handle. Second, a very quick, gross positioning step is used ("put that there"); this is often a large hand or body motion to get the object into place [20]. This step is characterized by *avoidance* of collisions and physics; the intent is to get the object to the vicinity of the goal position as quickly and easily as possible. Third, a fine tuning step finalizes the motion: very small motions are applied, adjusting the final position and allowing physical behavior such as gravity to "take hold" of the object at rest. In this final step, physical properties and interactions become more applicable and useful as the movement nears completion.

Following this model of manipulation, we derive a similar three-phase process for virtual world manipulation. The selection phase has already been described. We then implement a teleological phase, where the intermediate path of the object to the vicinity of the goal position is as direct as possible, relaxing physical restrictions, to follow the goal-oriented directive of the user. As the object nears the goal position, in the pseudo-physical phase, the object's resting pose should be tuned to be "realistic" within the defined simulation constraints of the virtual environment. Figure 7 shows an example of how this model would handle the task of moving a piano into a room.

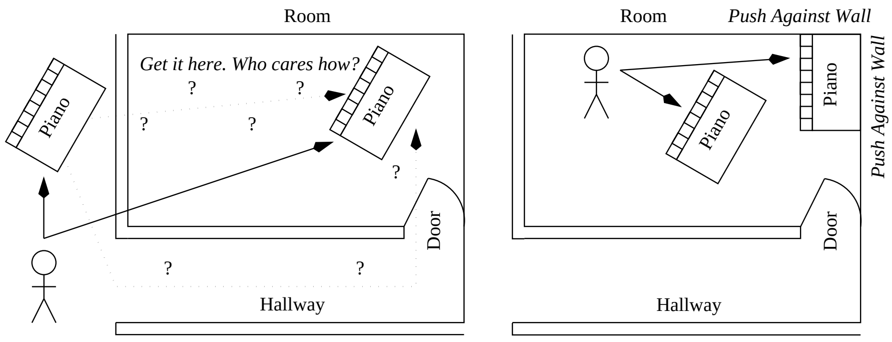

Figure 7: A slightly exaggerated example of our model of object manipulation at work for moving a piano into a room. On the left, the user has selected the piano; the first *teleological* step is to move the piano through the doorway. The user doesn't care how the piano gets there; physical interaction with the doorframe is not only useless but actively annoying. On the right, the piano is close to its desired end position; the *pseudo-physical* step is entered to push the piano the last few inches against the wall. In this step, interactions with gravity and collision with wall and floor are essential to easily place the piano flush against both.

A non-interactive motion admits a similar solution; the teleological step consists of the user pointing to a location with the intent to "put the object there." Our interpretation of this is to move the object directly to that location (i.e. "warping" it there without regard to physical obstructions or trying to "lift" against gravity), and then applying any desired local physical behavior to determine the valid final resting position.

The ability to rotate an object is often desired, but these rotations are generally about some preferred axis. If we can identify this axis *a priori* for a given situation, it is more intuitive to provide the user with a simple "click and slide" type of rotation, where the mouse X or Y offset is directly proportional to the amount of rotation.

3.2 Object Associations

This section describes a software framework, which we call *object associations*, developed to meet the needs of our general transformation mode. Object associations support simple and practical manipulation of 3D objects with 2D I/O devices via two special types of programmer-supplied procedures and an implicit grouping behavior. This model gives the programmer the ability to specify object-dependent methods of disambiguating 2D gestures in a 3D world and allows association of suitable local behavior with database objects to make precise default placement easy. These associations usually fall somewhere between physical simulations and mathematical constraints, but can be less formal and more flexible than either.

Our approach borrows heavily from several paradigms developed in the realm of interactive computer graphics over the past several decades. It first has notions of *snap-dragging* [7], but without the need of explicitly dealing with visible alignment manifolds; most alignments are provided automatically by the association procedures rather than explicitly by the user. Second, while it can emulate some of the behavior of a *physical simulation* of the objects in the environment [4, 27], it can be less constraining than our every-day world; objects can pass through one another and remain in physically impossible non-equilibrium positions under the control of suitable associations, which may be application-specific or may depend on the editing mode. Third, while some associations can be described as *constraints*, our system does not require the rigid formality and associated solution machinery found in mechanism editors based on underlying geometric constraint systems [36, 8, 35, 28, 24].

A novel feature that emerges naturally from our approach is an automated *implicit grouping mechanism*; it uses the relationships established between objects as they reposition themselves with respect to their environment.

3.2.1 Overview

The object associations mechanism is based on a two-phase approach to moving a selected object. During a first *relocation phase*, the object follows a trajectory free of physical or behavioral restrictions and which is a suitable disambiguation of the 2D path specification in screen space into a 3D motion in world space. During a second *association phase* the object uses its (possibly physical or pseudo-physical) association rules to determine a good nearby position which best satisfies the stated behavioral conditions of the object in a rest state.

Object associations are implemented with two types of small procedures that are invoked when an object is selected. Each object is assigned one *relocation procedure* but may have a number of prioritized *association procedures*. The relocation procedure is used during local, interactive motion to disambiguate gestures made with the mouse pointer; it defines a mapping of incremental 2D mouse motion to incremental 3D object motion. Association procedures are used for both local and remote placement; they apply additional motion components to an object, based on the other objects in the area, in order to preserve the desired object behavior. In addition, objects will dynamically link themselves to the reference objects with respect to which they have aligned themselves; they will typically follow any movements of these reference objects. While an object can be assigned only one relocation procedure, they may carry any number of (ordered) association procedures. The user can add or remove extra association procedures from the existing set to selected objects during the interactive walkthrough mode. When such an object is selected, its attributes will determine which relocation procedure applies to the object and which association procedures are used to determine the final placement of the object. We have integrated these procedures with the user interface layer that controls all the major editing functions: selection, dynamic grouping, dragging, and detailed placement.

So far we have implemented two relocation procedures: the *on-horizontal* procedure, designed for objects that move primarily horizontally, and the *on-surface* procedure, designed for objects that are attached to a potentially non-horizontal support surface. In both routines, the object moves along a piecewise continuous, polyhedral 2D manifold in space; the specific shape of the manifold is both view- and object-dependent. The left mouse button *translates* the object along the manifold without changing its orientation relative to the manifold. The middle button *rotates* the object about a line through the center of its bounding box normal to the manifold section on which the object rests.

In addition, there are four association procedures: the *stop-at-wall* procedure, the *pseudo-gravity* procedure, the *anti-gravity* procedure, and the *on-wall* procedure. Stop-at-wall forces objects being manipulated to remain within the current view. Pseudo-gravity simulates objects that normally rest on a horizontal support surface. Anti-gravity attaches light fixtures, smoke detectors, sprinklers, and other such objects to a ceiling. On-wall is used for pictures, white boards, wall clocks, and other objects that hang on vertical surfaces. All of the currently implemented WALKEDIT association searches use the same type of ray-based probing mechanism to find alignment objects. These ray-probes find nearby objects that affect the alignment of the selected object.

3.2.2 Relocation Procedures

The *local* motion paradigm – dragging the object with the mouse – is the basic editing move for fine-tuning the position of an object, or for moving objects over short distances. The user selects the object and moves the mouse pointer in the desired direction. To generate each frame of the motion animation, the *relocation procedure* is first called to convert the cursor position into a constrained position on a suitable auxiliary manifold that depends on the type of association carried by the selected object. The relocation procedure moves the object along the manifold in such a way that the selection point maintains coincidence with the cursor. After the relocation procedure determines the base motion, any relevant *association procedures* are run to determine additional motions that the object must perform to maintain its desired behavior. The association procedures will normally move the object in degrees of freedom not controlled by the mouse; however, if a more constraining motion is desired, it may further restrict the motion on the surface of the 2D manifold. For instance, the association procedure may force an object to move along a 1D path as if dragged by an invisible rubber band between the mouse and the selection point.

When the user initiates an interactive motion by holding down a shift/control key and clicking a mouse button, the relocation procedure is called with arguments corresponding to the current screen coordinates of the mouse, the user's view frustum, the particular drag mode being used (translate or rotate), the selection point on the object, and the original mouse screen coordinates where the object was selected. It first makes an *a priori* selection of one or two preferred DOFs that can be controlled directly and unambiguously with a mouse or with another 2-parameter input device, and which most naturally reflect the basic motion of the selected object. A simple, invisible, auxiliary 2-dimensional manifold, such as a plane, cylinder, or sphere, is established; the only requirement for the auxiliary manifold is that its projection into the view window maps points on the screen 1:1 onto points on the manifold, but the manifold will typically go through or near the selection point. The object is then moved under mouse control in such a way that its selection point stays on the manifold. The mapping between the cursor motion on the screen and the relocation of the selection point in the 3D virtual world is obtained by intersecting the cursor ray from the eye point with the auxiliary manifold. This gives an intuitive behavior for direct control; the object, grabbed by the user-selected handle, will follow the projection of the mouse movement on a reasonable restricted manifold. In general, these manifolds should be piecewise continuous so that the object will move in a predictable local way for small movements of the mouse. Figure 8 shows an example of what the manifold for the *on-surface* procedure, described below, would look like.

The manifold used in our *on-horizontal* procedure is simply a horizontal plane through the selection point. In the translation mode, the eye-cursor ray is intersected with the plane equation $z = \mathbf{s}_z$, where $\mathbf{s}$ is the original coordinate of the selection point. The ray-plane intersection returns some point $\mathbf{i}$; the procedure returns translation vector $\mathbf{i} - \mathbf{s}$. In the rotation mode, the eye-cursor ray is ignored; the x offset of the mouse pointer on the screen is used as an angle. A rotation by that angle about the plane normal is returned.

On-surface uses a more complex manifold composed of piece-wise planar offset surface segments situated in front of the faces of visible surfaces in the scene. In the translation mode, the procedure uses the geometric database to intersect the eye-mouse ray with the first surface it hits. The intersection point $\mathbf{i}$ of the ray with the surface is determined, and the translation vector $\mathbf{i} - \mathbf{s}$ is returned (where $\mathbf{s}$ is, again, the initial coordinate of the selection point). However, the algorithm also computes the rotation angle between the manifold's surface normal at the selection point and at the new point, and returns that rotation to maintain the orientation of the object's "back" with respect to the manifold. This makes wall hangings follow the changes in wall orientation; if a wall hanging is moved around a corner, the rotation causes it to turn its back toward the new wall as it moves. The on-surface rotation mode simply rotates the object about the normal of the manifold.

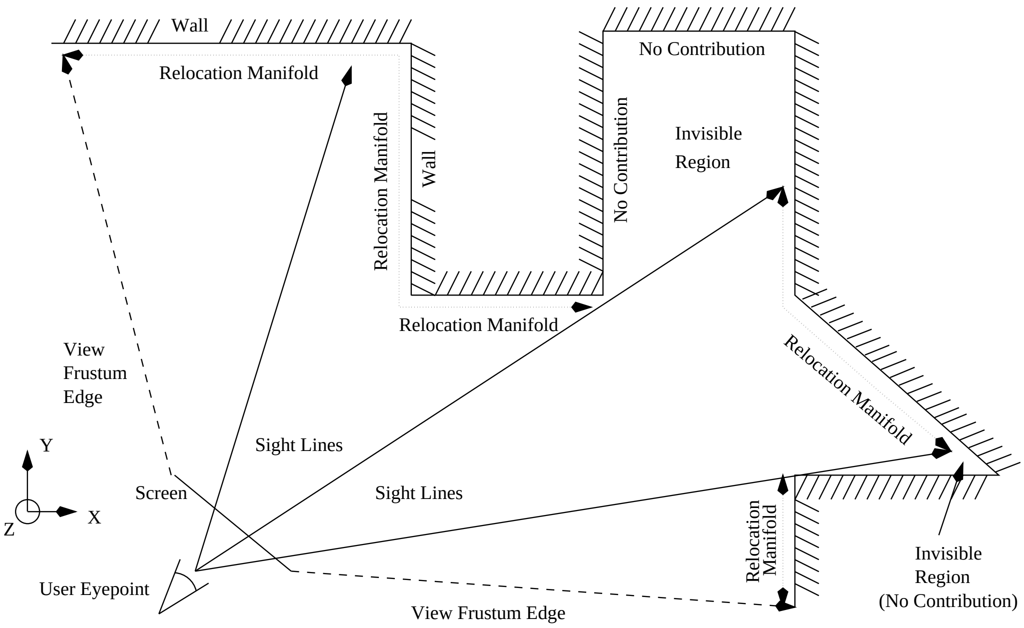

Figure 8: A 2D example of the relocation manifold for the *on-surface* relocation procedure. The dashed line shows the edges of the view frustum. The dotted segments show the piecewise planar segments that the visible portions of the walls contribute to the manifold. The solid rays show sight lines; note that the wall surfaces that contribute are exactly those that are visible to the user. Note also that, in a 3D version, the visible portions of the floor and ceiling would also contribute to this manifold.

After sliding the object along the alignment manifold, the relocation procedure returns a 3D *offset vector* in space, representing the difference between the original pose of the object when it was selected and the new pose indicated by the mouse motion; this represents the fundamental teleological motion intended by the user (Figure 9). This offset position is what is passed on to the *association procedures* for the object.

3.2.3 Association Procedures

At the offset position, the association procedure needs to find the closest valid rest position and orientation for the moving object, given that the object is supposed to obey some particular behavior. The first step is to find the possible candidates for alignment. All association procedures currently rely on

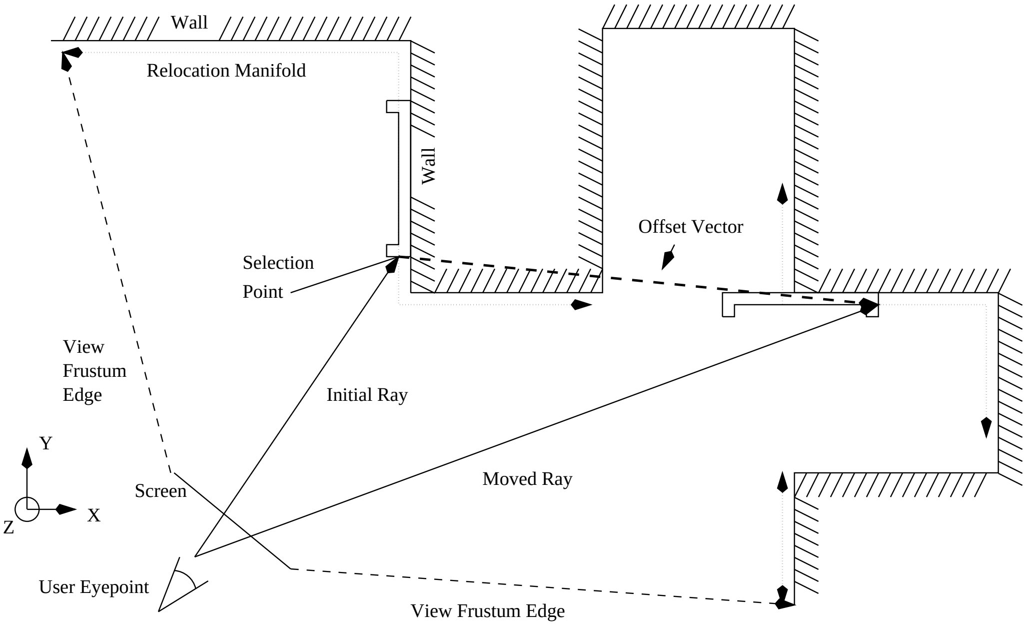

Figure 9: Moving a picture under the on-surface relocation procedure. A user has selected a picture on the wall; the "Initial Ray" shows the mouse cursor ray when the object is selected, and the selection point is indicated at the intersection with the relocation manifold. The user then moves the mouse; the new "Moved Ray" is intersected with the relocation manifold, producing an offset translation (shown as a heavily dashed line) and rotation which is returned by the relocation procedure. The new location of the picture is also shown; note that the rotation has been computed to maintain the picture's orientation relative to the manifold's surface normal.

ray projections. Pseudo-gravity and anti-gravity cast rays vertically downward and upward from the selection point, respectively; the objects hit by these rays are the objects with respect to which the selected object's position is adjusted, causing the selected object to fall down or up respectively. The on-wall association casts rays in the major horizontal axis directions of the original definition of the object; the closest object in those four directions is the one used for alignment, as the object "falls" sideways against the closest vertical surface.

Here is the pseudo-gravity procedure in pseudo code:

1. While the object O has changed height in the last iteration, do:
   (a) Project a ray from the selection point S on object O downward to hit some face F of some object A;
   (b) Determine if S is within the bounding box of some object B (the smallest bounding box if there is more than one);
   (c) if (B is NULL) or (B==A) or (S is visible), drop the bottom of O's bounding box to the height of F; else, lift the bottom of O's bounding box to the height of the top of B's bounding box;
2. Return the total motion of O and associate O with A;

In general, this procedure will place the selected object on top of another one that the user points at by using a combination of visibility cues and interference tests (see section 3.2.6 for discussion of visibility issues). Figure 10 shows some examples of pseudo-gravity in operation. The anti-gravity procedure, used for objects that stick to ceiling surfaces, is identical to pseudo-gravity with the vertical directions reversed ("upward" instead of "downward" and "bottom" for "top"). The on-wall procedure makes some additional assumptions. For an object to attach itself to a wall, it needs to have some notion of a "back-side" which is moved to be coincident with the closest vertical support surface. Since the Soda Hall object descriptions do not carry such a notion explicitly, we assume that the object is defined with its back's surface normal in one of the major horizontal axis directions of the object's local coordinate system (Figure 11). These four directions are then checked for the closest vertical surface, and the pseudo-gravity algorithm is then run along the corresponding axis. Thus when the user first brings such an object into the Soda Hall environment, it needs to be placed close to some wall with its one side that is supposed to act as its back-side. For similar reasons, the pseudo-gravity and pseudo-antigravity procedures assume that the object is defined with its "bottom" at the extreme of the negative Z local axis, or its "top" at the extreme of the positive Z local axis respectively. These assumptions about how an object is defined allow the procedures to align the object properly by rotating the object such that the appropriate local coordinate system major axis is parallel to the support surface normal, and the object can be made to rest "on" the support surface by moving the object's local-axis-aligned bounding box extrema to touch the surface (Figure 12).

For every move generated from an offset vector along the relocation manifold, the association procedures decide what local fix-up motions must be made at the new position to implement the desired local behavior for the object (e.g., falling to a supporting surface, in the case of gravity). Each association procedure computes local components of the overall motion, commensurate with the desired object behavior. The motion generated by the association procedures may also cause the object to change from one supporting manifold to another, such as when the motion generated by the relocation procedure would move the object beyond the edge of the current support or into another solid object.

Once the association has determined what local objects and forces affect the motion of the selected object, the offset vector from the relocation procedure is modified to reflect the local motion, and the

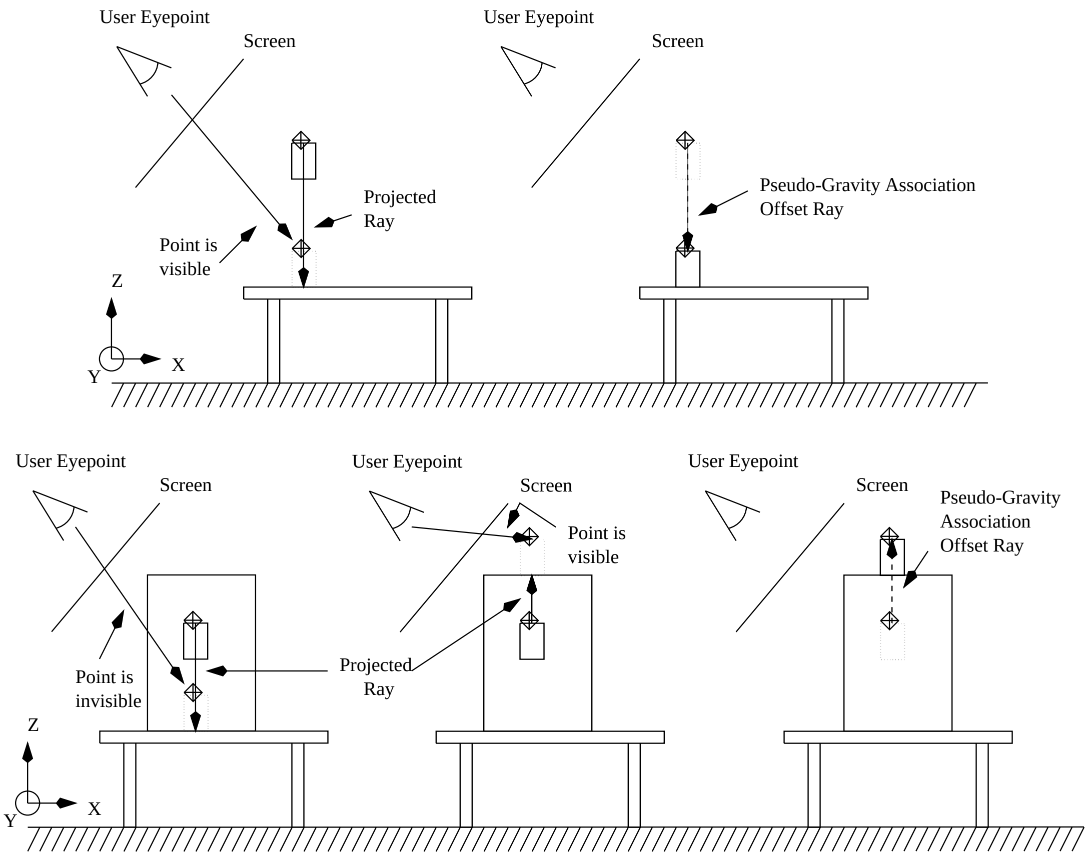

Figure 10: Some situations in which pseudo-gravity is used. In the upper sequence, the cup begins a few inches above the table and is in view; pseudo-gravity casts a ray down, finds the table surface, and moves the object down to it. In the lower sequence, the cup is embedded in a block and is not visible. In the second frame, the association finds the tabletop below the cup, but if the cup is placed on the tabletop the selection point becomes invisible. Checking upward, the association finds the top of the block as the next closest valid support. The cup is moved to the top of the block's bounding box, where it is visible and its bottom is resting on the top of the block.

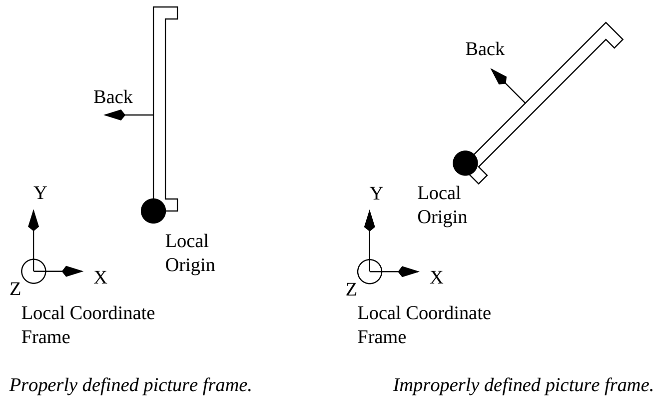

Figure 11: When defining objects for use with the on-wall association, the object must have its "back" normal to one of the major axis directions. On the left, the back is defined normal to -X, which is valid. On the right, the back is at a 45 degree angle to the coordinate axes; on-wall will not position this object properly. Similar restrictions apply to pseudo-gravity, in which the objects' "bottom" must be normal to -Z, and pseudo-antigravity, in which the objects' "top" must be normal to +Z.

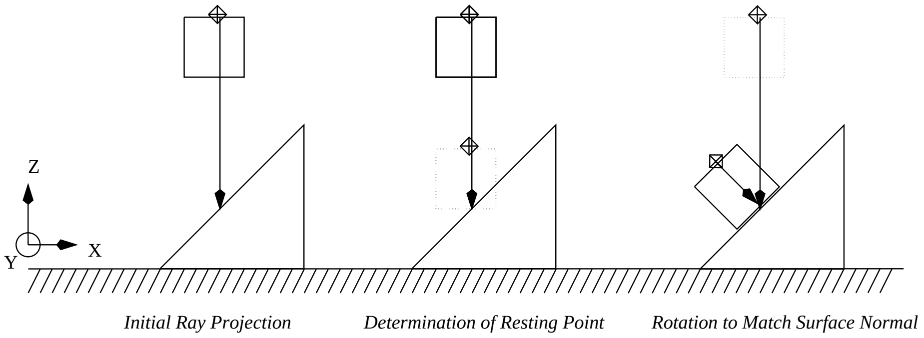

Figure 12: When the association procedures align an object with a support surface, the object is rotated to bring an appropriate major axis parallel to the support surface normal. Here, a box comes to rest on a sloped ramp under pseudo-gravity. The association rotates the box about the intersection point of a ray straight down from the selection point with the ramp's top face. The rotation is calculated to bring the bottom of the box into parallel contact with the support surface.

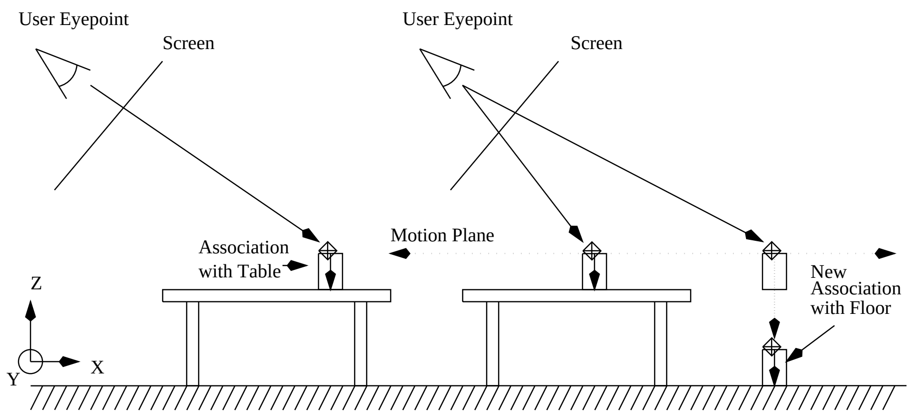

Figure 13: An example of an object changing supports. In this case, a cup moves off the edge of a table. The old support surface is the table, determined by the ray projection straight down from the selection point on the cup; we say that the cup is *associated* with the table. When the cup is moved off the table, the ray projection indicates the floor. Pseudo-gravity adds the necessary downward component to the motion to drop the cup to the floor; the cup is now associated with the floor.

new vector is returned from the association procedure. The procedure may also optionally return a set of one or more new local *associations* of the selected object with the other nearby objects that help determine its final position, such as the table that supports a cup. These associations are used by the implicit grouping mechanism (see section 3.2.5) to determine which objects should be grouped together when they are manipulated. When the user finalizes the motion by "releasing" the selected object, the new associations replace the original associations that were in effect when the object was selected.

3.2.4 Stacking Multiple Associations

Multiple association procedures may come into play for single objects. For example, objects like book cases are supposed to obey *pseudo gravity* and simultaneously fit snugly against walls. This may reduce the DOFs of an object to just one or even zero. In the latter case, the object may jump from one desirable location to the next one as the user moves the mouse pointer and the association procedure selects the closest location that fits the desired behavior.

Multiple associations attached to an object type are explicitly ordered. The corresponding procedures are called in a chain, each one receiving the cumulative associations and offsets generated by the one before. A systems programmer assigning combinations of associations to certain types of objects must consider their possible interactions. The interactions can potentially be very complicated since associations are described functionally rather than mathematically; an association procedure can conceivably do anything. Because of this, it is difficult, if not impossible, for the object association framework to generically resolve conflicts between all combinations of procedures. The associations implemented in WALKEDIT are simple and orthogonal and are particularly tailored to the rectilinear, axial environment of Soda Hall; thus, their interactions are easy to predict and not very problematic. The individual adjustments of all associations are gathered into a single cumulative transformation which is then uniformly applied to the selected object and all its dependent associated objects in the dynamically found group. Figure 14 shows two associations, pseudo-gravity and on-wall, at work on a file cabinet.

Figure 15 shows the flow of control, from the inputs to the object association mechanism to its output

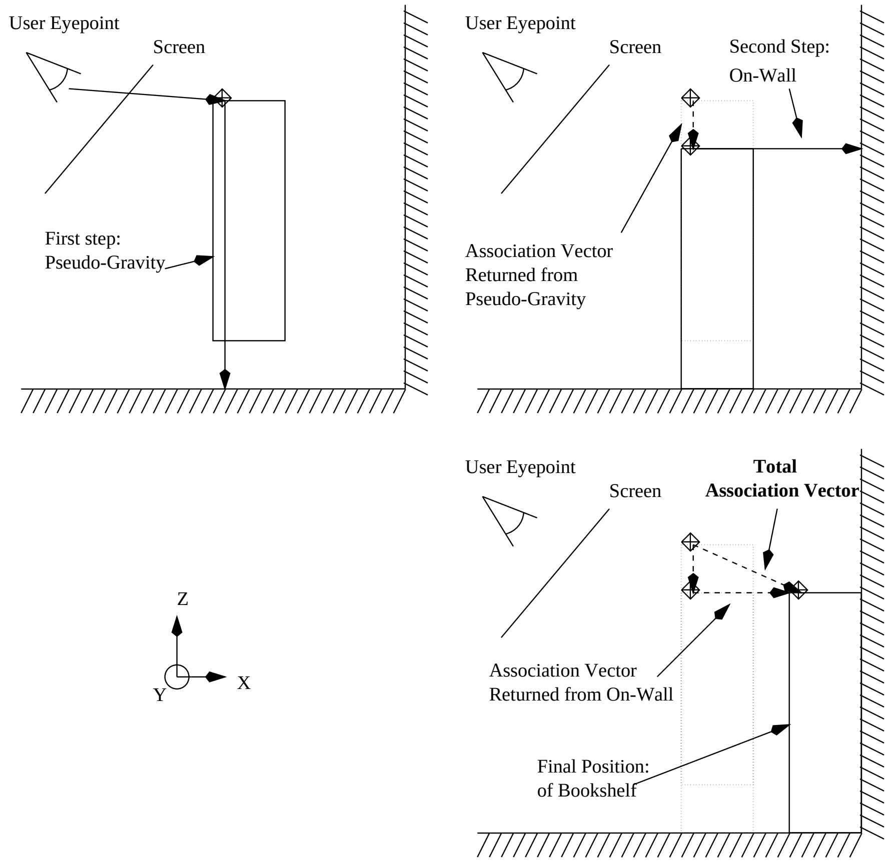

Figure 14: A file cabinet under pseudo-gravity and on-wall. When the cabinet is moved, it first drops to the floor at the selected location (upper left), then positions its back against the indicated wall (upper right). The result is a cumulative transformation that should satisfy both associations (lower right).

for an object with a relocation procedure and two association procedures. On the input side, the user selects the object (upper box) and then moves it with the mouse pointer (lower box). Selecting the object launches the implicit grouping search (see section 3.2.5), which proceeds simultaneously with the other operations. The original position of the object and the motion of the mouse are sent into the relocation procedure, which uses the initial position and the mouse motion to determine an offset which is sent through the chain of association procedures. Each association procedure modifies the offset and sends it to the next procedure, while outputting associations. The last procedure also outputs the final motion of the object in 3D space, which is applied to the list of objects output by the implicit grouping search.

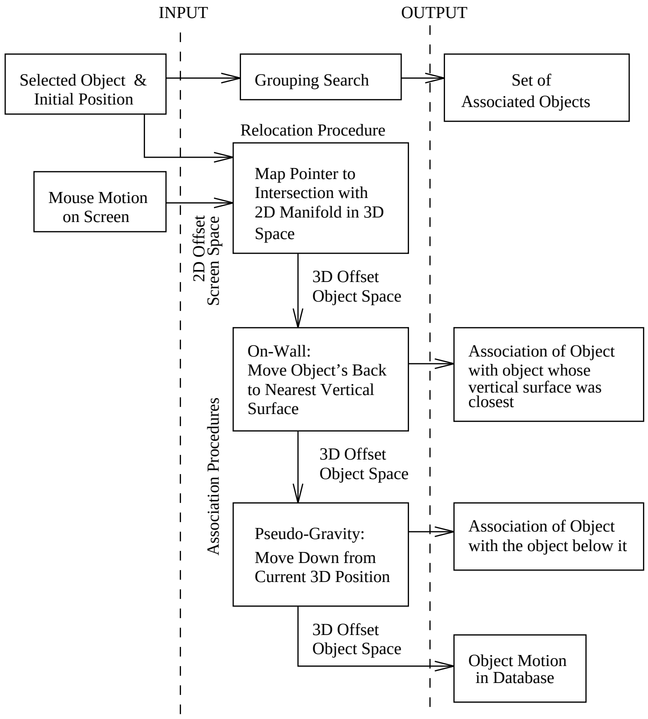

Figure 15: A flowchart showing the various procedures at work for an object that obeys on-wall and pseudo-gravity (such as a bookcase).

An interesting algorithmic question is raised by cyclic constraints arising from the mutual associations of several objects. Imagine placing two "on-wall" objects back-to-back in the middle of a room. Each object will associate with the other, thus forming a cycle. If object A is selected, object B will dynamically group with it, and will want to rigidly follow the motion of object A; however, object A will want to move along the surface of object B, because its association sees B as the closest vertical surface. Thus the two objects can never again be moved away from their joint back-to-back alignment plane. A similar situation may arise if an "on-ceiling" light fixture is attached to the underside of a table obeying pseudo-gravity. Our current solution involves breaking loops - once they have been detected - at the point where a large object associates itself with a smaller object. This approach provides the right feel in a building environment, but it is not be a general answer to the problem.

3.2.5 Interactive Automated Grouping

Basic Principles

In WALKEDIT, selection is performed by shift-clicking the object. There may be other objects that have been previously associated with the selected object; these other objects were positioned with respect to the selected object when they were last moved. For example, the reference object identified by the pseudo-gravity association is the surface on which the selected object came to rest. Since the position of the reference object influenced the position of the selected object, it makes sense to implicitly group the latter with the former and maintain that relative positioning when the reference object is moved. This means that all of these associated objects must be found and grouped with every new object selected; this grouping is maintained for the duration of the motion. An object can have multiple associations; it will then move when any of its reference objects moves.

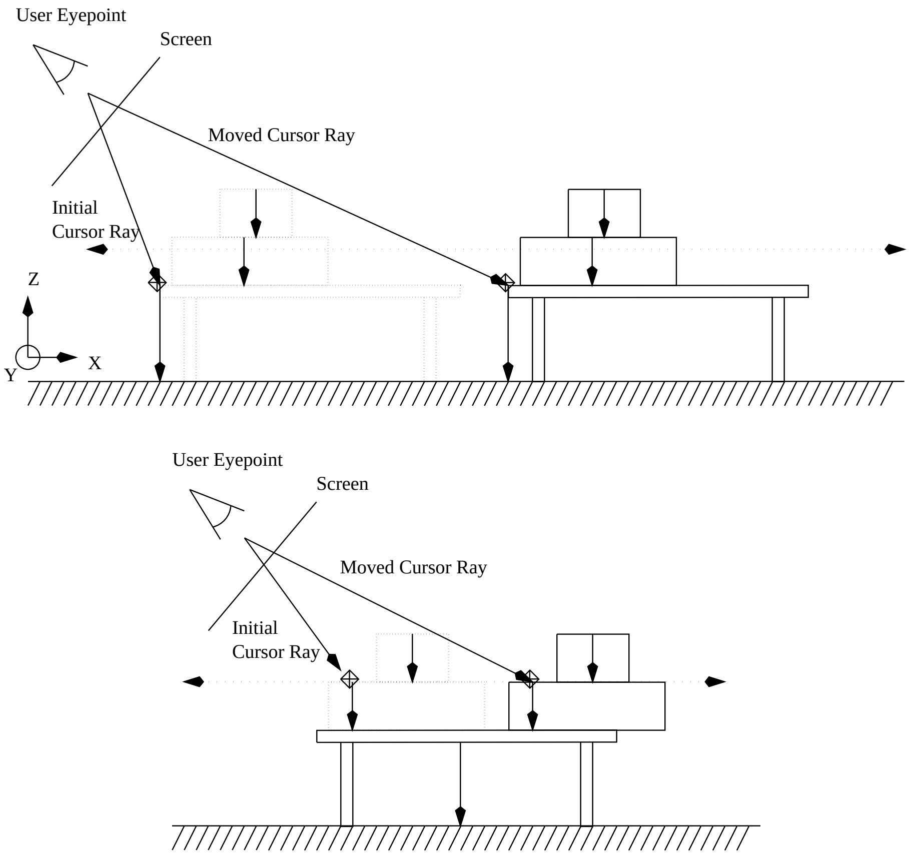

Figure 16: Implicit grouping. The blocks and the table are assigned the pseudo-gravity association. When the table is moved (upper picture), the blocks move with it, because they are all associated with the table or each other. When the bottom block is moved (lower picture), the top blocks (which are associated with it) move, but the table (which is associated with the floor) remains where it is.

Associations are *not* permanently maintained constraints; they are applied to the object that is currently being moved. Because associations are determined from a selected object towards potential reference objects, but are used in the opposite direction, valid associations between two objects may change by the motion of a third, unrelated object. For example, an alignment association between two concave objects may leave space between the two into which a third object can be inserted, thereby breaking the previous association. To allow for such changes and to ensure robust behavior of the object association framework, every time an object is selected we perform a local search for associated objects dynamically in real time and store them in a separate data structure. For efficiency, likely candidates (that is, those objects that were known to be associated with the selected object previously) are checked first. Then, a general search is started in the vicinity of the selected object, relying on our cell-based spatial subdivision structure used for visibility precomputation and observer tracking [43]. The association procedures (see below) are called for all objects incident to the subdivision cells occupied by the selected source object to see if they are associated with it; each object returns a set of association links, and all of these links together form a graph on the objects in that region. The search efficiently calculates a local closure on this graph to obtain the group of objects linked, directly or indirectly, to the selected object.

To keep the virtual environment interactive and the response to any mouse-directed motions instantaneous, we do not delay the interactive manipulation of the original selected object; we carry out the association search in the background. As soon as an associated object is found, it is subjected to the cumulative set of manipulation transformations applied so far to the source object. This approach has the somewhat startling effect, that when the user grabs and moves a fully loaded desk, some of the objects on the desk may at first remain behind, suspended in mid-air, and will then catch up with the new desk position within a few seconds as they are found to be associated with the desk. We found that most users quickly accept this behavior. To minimize this effect, the association closure graphs, once constructed, are cached in memory, so that any further moves of such a group of objects can be truly instantaneous. The closure process may be safely interrupted before closure is complete if the user decides not to move the chosen object but instead selects a different one. The cache holds whatever portion of the graph was completed, and this potentially useful work is saved; the next time an object in the area is selected, the system will simply pick up the search where it was left off.

This implicit grouping mechanism replaces both the explicit grouping mechanism found in many 2D editors and the functional grouping resulting from setting constraints between objects. Our mechanism keeps the user focused on the actual positioning of the desired object, while automatically making many of the grouping connections the user would have to make by hand with either of the classical methods. Furthermore, breaking a connection between objects that have been implicitly associated is as simple as grasping the dependent (associated) object and moving it to a new location, at which point the association with the old reference object is broken and a new one is established. Of course, we also give the user the power to override the automatic grouping mechanism by turning it off, or to perform grouping manually by *alt*-clicking objects to explicitly add or subtract them from the current group. The two grouping mechanisms can be active simultaneously; adding an object to a group by *alt*-clicking will then also add any associated objects to that group.

Dynamic Gathering

The closure of the graph of associations on objects is computed using a dynamic gathering algorithm. The basic dynamic gathering step takes as input a set of selected objects and a cell to be explored for other objects which are associated with those selected objects. The association procedures for each object in the cell are called in turn to determine which other objects (which may or may not be in that cell) the given object is associated with. If any of the objects that the given object is associated with are in the list of selected objects, the given object is added to the selected list. After all objects in the cell are explored in this fashion, the newly selected objects are checked to see if they lie within or partially within any cells other than the given cell; if they are, and those cells have not been searched yet in this run of the dynamic gathering algorithm, the process recurses on the other cells with the new list of selected objects. Upon termination, this algorithm results in a complete list of the objects which are associated, through any sequence of association links, with the original set of selected objects (Figure 17).

This basic procedure was coded such that it could be called in small steps. The dynamic gathering is initiated with a function call, which sets up data structures and returns immediately. Then, the user calls a stepping function until that function returns the value "done," at which point the gathering

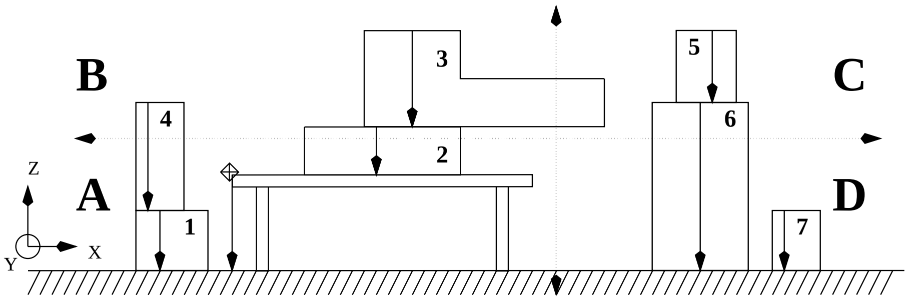

Figure 17: An example of the grouping search in action. The dotted lines indicate subdivision cell boundaries; the solid arrows show the associations between the objects. The table is selected, and a search begins, rooted in cell A. Blocks 1, 2, and 4 are in the same cell as the table, and their associations are checked; block 2 is found to be associated with the table. Since block 2 is also in cell B, that cell is added to the search queue, which causes block 3 to be checked; block 3's association with block 2 is found, adding cell C to the queue. This will cause blocks 5 and 6 to be checked. Since neither block 5 nor block 6 is associated with any object in the group, the recursion terminates without checking cell D.

is complete. Each step computes at most the associations for a single object, and a callback may be registered that is invoked each time a new object is found to be associated with the growing group. This was done to allow the gathering to be "backgrounded;" upon initiating a gathering, the operating system event queue has a "next grouping step" event inserted, and these events are called at a rate determined by how busy the CPU is handling the user's input. If the user is not doing anything, the grouping gets all the CPUs attention; if the user is trying to manipulate the growing group, however, the mouse events are handled as they come in, with one grouping step executed for each mouse or keyboard event handled.

Caching Associations

Early in the design of the dynamic grouping step, it became obvious that the association graph closure performed to determine which objects to group with a particular object discarded a great deal of useful information. The closure found all associations of objects in each cell it touched; thus, it actually computed parts of the grouping closures for each of those objects. Since users will typically perform many consecutive operations on one group, cell, room, or area of the database (for example, setting up their desk or office), it seems inefficient to discard these partial closures only to have to recompute them from scratch for the very next object manipulation. Thus, a mechanism was devised for efficiently storing this partial closure information in the object data.

Each object contains three pointers: a parent pointer, a child pointer, and a sibling pointer. The parent pointer points to an object with which the selected object is associated. The sibling pointer points to a sibling in a circularly linked list of other objects associated with the same parent. The child pointer points to one of the objects associated with the selected object.

When a selected object is moved or the grouping search touches it, its associations are computed. If one or more associations exist, one of them is selected, and the selected object's parent pointer is set to that associated object. The associated object's child pointer is traversed, and the selected object is linked into that sibling list. In this way, with a constant increase in object size (3 pointers, or 12 bytes), we save some (or all) of the association information computed by the grouping search for the object.

When the grouping search is being conducted, each time a new object is added to the selected objects list, its child pointer is traversed, gathering all of the objects pointed to by the child, its siblings, and their children recursively. These objects are simultaneously added to the list, as they are guaranteed to be associated with the selected object. In our environment of WALKEDIT, this caching mechanism nearly always caches the entire set of objects associated with the selected object, causing the desired group to be found instantly upon selection.

Simplifying Assumptions and Heuristics

There is one basic assumption upon which our dynamic gathering algorithm relies for correctness. If two objects are associated, those objects must both be incident to at least one common spatial subdivision cell. This assumption seemed reasonable given that all the associations we constructed bring objects into close contact (e.g., pseudo-gravity will bring an object into contact with a support surface). However, it is easy to posit association procedures for which this assumption is invalid; for example, consider a procedure that pushes objects a specified distance out from a wall. Our algorithm needs to be improved to handle such cases.

Consider two other assumptions we might make about the set of objects we're manipulating: that each object is associated with at most one other object, and no object may be slid between two associated objects to break that association. If we can make these assumptions, and we are using the association caching scheme outlined above, then we provably never need to touch a database cell more than once with the dynamic grouping algorithm. This assumption may be asserted in the options menu of WALKEDIT; when it is active, the cells are marked when any dynamic grouping step searches them, and if that cell is ever encountered by another grouping step, it is ignored. This can greatly accelerate the average completion time of the dynamic grouping step if the user spends a long time manipulating objects in a limited area. Since our most common assignments of association procedures to objects is simple pseudo-gravity or on-wall, we make this assumption by default in WALKEDIT.

3.2.6 User Interface Issues

While we can start from a few desirable paradigms (see Section 2) to define the user interface for object manipulation in a 3D virtual world, there will always be situations that will put some of these principles in conflict with one another and where there seems to be no obvious "right" answer. A few such tricky problems are raised in this section and our current solutions are discussed.

Limitations of the Object Associations Paradigm

We cannot reasonably expect that a few simple relocation and association procedures will take care of *all* of the editing needs in our building environment. However, we also cannot expect the user of the system to deal with dozens of different sets of procedures, any one of which might be appropriate at any given time. Our goal is not to solve all of the problems of 3D direct manipulation with a single approach: object associations should be tailored to make 90% of the typically encountered operations easy and natural. For more special-purpose editing needs we still can access traditional editor functions via pull-down menus. If one needs an exact rotation by 45 degrees, one opens the *rotation operator* menu; if one wants to create a perfect row of 20 chairs, the familiar *replicate* menu is perfectly appropriate. If, on the other hand, one finds that one often has to do a special task that is not well supported by classical editor menu commands, such as pushing furniture into corners, then it pays to write a new association procedure "in-corner." This procedure probes in all 4 directions, finds the *two* closest objects, and then does on-wall alignments in two directions, trying to satisfy them both at the same time.

If this is not good enough, because one frequently wants to crowd furniture together in less regular formations, then it is time to develop a more or less accurate pseudo-physical collision detection mechanism and add it to the collection of association procedures. Depending on the types of objects that need to be manipulated, this may simply be based on bounding boxes (good enough for file cabinets) or may use a more sophisticated algorithm that can handle concave objects (needed for grand pianos). We are currently experimenting with a prototype implementation of such a collision detection routine based on the Lin-Canny algorithm that quickly finds closest features in pairs of convex shapes [32, 4].

Use of Visibility Information

One of the main cues used to disambiguate the depth coordinate during object manipulation is the intersection of the cursor ray with a visible support surface. Thus when moving an object obeying pseudo-gravity, one would typically grab it near its "foot" while looking downwards onto the supporting surface. This establishes a relocation manifold with a reasonable intersection angle with the cursor ray and gives the user good interactive control over the motion. It raises the issue what should happen when the object is dragged beyond the visible range of the support surface or outside the extent of the support altogether. It also raises the issue how one can ever lift an object *off* such a support surface, e.g., to place a book onto a higher shelf.

Figure 18 illustrates a first typical situation. It should be possible to slide a coffee cup underneath a table; thus, we can not simply lift it to the top of the table when the bounding boxes of the cup and of the table start to intersect. Here we use visibility information and our *pointing* paradigm to resolve the issue. As long as the sight ray to the selection point clears the table top, the cup stays on the floor. Since no part of the table is between the cup and the floor, and the cup is not actually intersecting the table, the association procedure has no difficulties settling the cup in a valid position on the floor. However, when the ray intersects any part of the table, *and* the bounding boxes of the cup and the table intersect, the cup gets lifted to the top of the table.

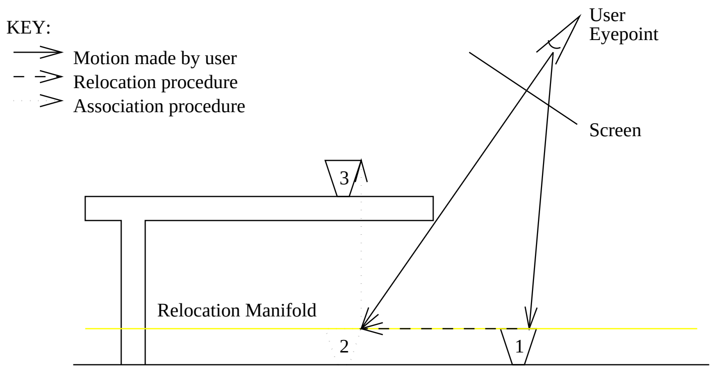

Figure 18: A selected cup (1) is dragged under a table. Visibility information is used to determine when it rises to the tabletop. Solid arrows show user mouse motions; the dashed arrow shows where the relocation procedure moves the object based on the mouse motions (2); the dotted arrow shows the position adjustment made by the association procedure when the selection point becomes obscured by the table edge after relocation (3).

Another critical situation is shown in Figure 19. When the cup is dragged beyond the edge of the table top, a non-physical situation occurs. This could be resolved in two ways. The system could try to place the cup where the cursor ray hits a valid support surface. Since the ray may still hit the table top, or perhaps end in a vertical surface, this will not always lead to a useful answer. Thus we have found that it makes more sense to give priority to the physical view of the world and drop the cup straight down from the spot where it left the table top to the floor, which then acts as its new support surface.

In all these situations we have an interesting interplay between the teleological and the physical view of our virtual world; visibility information and the intersection of the cursor ray with a particular object

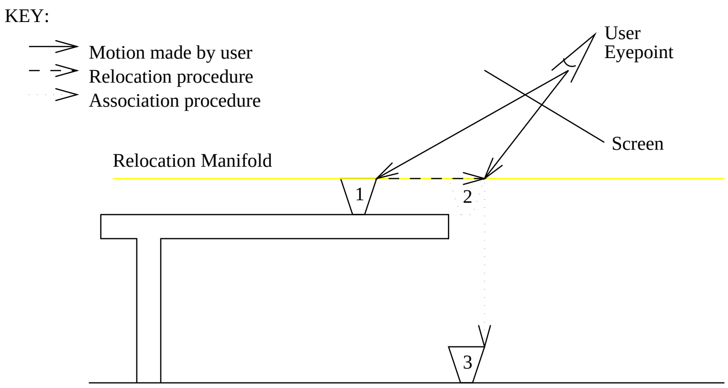

Figure 19: A selected cup (1) is dragged off a table's supporting surface. The cup falls (2) onto the lower surface (3).

are used as additional cues to infer the intent of the user.

Mouse-Cursor Correspondence

Another key paradigm of the desired user interface is that the object should follow the cursor as directly as possible. This principle must sometimes be violated in situations such as the ones above, where establishing a physically valid position may result in a dramatic (vertical) adjustment. As long as the association procedure doesn't add any motion to the object, the relocation procedure usually maintains correspondence. However, the association procedure has no responsibility to maintain the connection between the mouse pointer and the selection point. This then raises the issue whether in such situations the cursor should stay where the user last moved it, or should be "warped" along with the extra motion given to the object by the association procedure. While it is generally preferable to keep the cursor point attached to the handle established at the selection point on the object, this has the consequence that the cursor - and the object itself - may disappear from the screen altogether. Consider the situation in Figure 20 where the cup is moved beyond the *back* end of the table, and where the cursor ray hits no suitable support. The gravity procedure will drop the cup behind the table and possibly out of sight, and the cursor may vanish with it if the floor lies below the lower edge of the viewport. If the fall happens too quickly, the user might not know where the cup (or the cursor) has gone and what should be done to bring it back.

These problems are related to the fact that the association procedure has nothing to do with the mouse cursor position on the screen; it is only interested in placing the object in a valid local position. Since there is no direct relationship, it is counterintuitive in the general case to artificially force the mouse cursor to follow the result of the association procedure's adjustment of the object's position. The user controls the mouse cursor; the cursor should only move in response to the user's direct command. Furthermore, the relative motion of the 2D mouse under the user's control directly reflects motion along the 2D relocation manifold, which is the primary method of controlling the object; if we warp the cursor at the whim of the association procedure, the correspondence of the cursor with the *relocation manifold* is lost, to the detriment of the user's ability to determine where the object will move when a small relative cursor motion is applied. Thus, we do not warp the cursor to follow the results of the relocation or association procedures; there is enough information in the relative motion of the selection point and mouse cursor for a user to be able to determine how the object will move in response to mouse input.

As a result of this decision, the object can become "lost" to the mouse pointer when the association procedure adds large offsets to the object position. We have ensured that there are many redundant escape routes for the user in such a situation. To bring the object back into view, the user can move the cursor back along the relocation manifold so that the (invisible) cup moves back into the bounding box of the table, whereupon it jumps back to the table top (i.e. object motion has no *hysteresis*; see the next section for details). Alternatively, the user may go to a new location from where the cup is visible, and then continue moving it from its current location on the floor behind the table. Finally, if the object seems totally lost, it can readily be brought into the knapsack while it is still selected, and from there it can be placed directly at the current cursor position. A keyboard shortcut permits to "warp" the object directly from any (possibly hidden) position to the cursor position with a single *ctrl*-click. This operation is also a very efficient way to quickly populate a room with furniture. It takes three mouse operations to place an object in a desired spot: one click to select it, a *ctrl*-click to warp it into the neighborhood of the desired spot, and one *shift*-click-and-drag operation to fine-tune the final position.

Hysteresis in Manipulation

Both the mouse-cursor correspondence and visibility problems contain a question of hysteresis during interactive motion; that is, if the mouse pointer, during an interactive motion, is moved from point A to point B on the screen, moving the object along some path in space, and then the cursor is moved back from B to A along the same screen path, does the object follow the exact reverse path in space, or does it follow some different path? Furthermore, does the object even end up in its original position, or is the correspondence of point A to the object's initial position lost as soon as the mouse cursor moved? This question is clearly evoked by moving an object off of the edge of a table; we grab the object on the table (point A) and move the mouse cursor "down" on the screen, moving the object along the table surface until it falls off (point B). The object is now on the floor; do we re-associate the object with the floor at that instant, or do we wait until the entire motion is complete? If we do the former, and move the cursor back "up" the screen, the object will remain on the floor until the cursor ray is obscured by the tabletop, at which point it will return to the surface of the table; this demonstrates a distinct hysteresis, as the object could very well end up on the floor instead of the tabletop when the mouse cursor reaches point A again. If, on the other hand, we do not re-associate, the cup will jump back up onto the table immediately as the cursor begins backtracking, even though if the cup had began the motion in that position the same cursor motion would have resulted in it sliding under the table instead. However, this solution shows no hysteresis; the cup will follow the precise path in reverse upon returning the cursor from B to A as it followed from A to B (Figure 21).

In our tests, hysteresis was found to cause more confusion than it solved. Thus, the current object

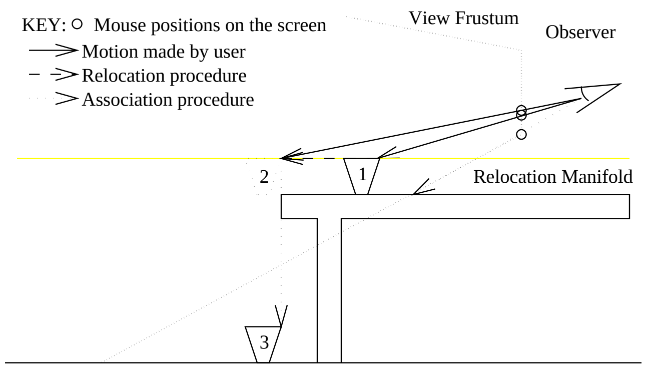

Figure 20: A selected cup (1) is moved off the back of a table (2), falling completely out of the view window (3). The mouse cursor cannot simultaneously track the selection point and remain within the window boundaries.

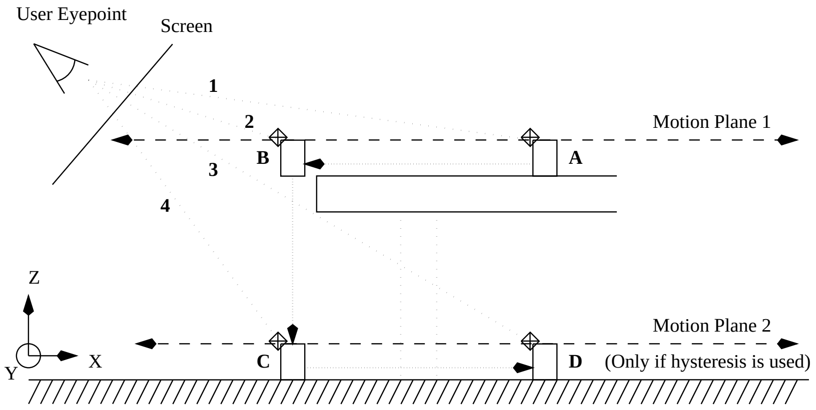

Figure 21: An example of the effect of hysteresis in manipulating a cup on a table. Suppose the cup was selected at point A with mouse cursor ray 1, and the mouse cursor was moved down on the screen until the cup reached point B (mouse cursor ray 2), at which point the cup dropped off the table to point C. If the system dynamically re-associated the cup at point C (requiring the mouse cursor to be warped to position 4), the association procedure would cause the user to experience hysteresis; moving the mouse cursor back up the screen, the cup would stay on the floor and move to point D (mouse cursor ray 3) due to the fact that the selection point would remain visible as the cup slides back toward the table. If the association procedure did not re-associate, the mouse cursor would remain at position 2 even though the cup would move to position C; as the mouse cursor was moved back from position 2 to position 1, the cup would follow the original path backwards from C to B to A.

association system does not re-associate objects until the current manipulation of the object has been completed (i.e. the user has released the mouse button). This means that the option to backtrack the object is always available, which provides a subtle but very convenient trivial undo capability; the user can undo the motion by simply moving the mouse cursor back to where it started.

3.2.7 Software Engineering Issues

Flexibility and Extensibility

Providing desired object behaviors in 3D virtual worlds is in principle not an easy task. Many nitty-gritty problems concerning data structures and efficient representations must be addressed in order to keep the environment truly interactive. Creating a cohesive framework of object associations is our attempt at keeping this overhead concentrated in one place, so that it can be amortized more easily by the systems programmer with each new object behavior introduced, and so that the user can be given the flexibility of easily choosing the types of behaviors for each object that are most appropriate for the manipulation tasks at hand.

The descriptions of the association and relocation procedures used in the Soda Hall walkthrough look very simple in pseudo-code. It is important to note that the pseudo-code is very close to the level of the actual C code used for the implemented procedures. This is because the WALKTHRU program system provides a rich set of libraries including a complete geometric computation package that operates on vectors, rays, points, planes, and other objects. It also provides the mechanisms to easily search the local area of an object for other objects, to find the objects whose bounding boxes contain a given point, and to quickly find the first object intersected by some space ray. Thus, most lines of pseudo code convert to a few lines of actual C code, making implementation rather straightforward. In such an environment, object associations are most naturally implemented with additional C routines; the C language is more flexible and powerful than any higher level geometric scripting language we could design ourselves.

Implementing Relocation and Association Procedures

The object associations system coded into WALKEDIT is designed to easily accept new relocation and association procedures; the only caveat is that adding procedures into the current system requires a recompilation of the WALKEDIT program. New procedures are registered by assigning them a unique identifying number and adding the name, number, and prototype of the procedure to a header file. After this is done and WALKEDIT is recompiled, the new procedure will appear in the run-time lists, may be assigned to objects interactively or off-line in the object properties file, and will be called when necessary by the object association package.

Before the relocation procedure is called, the system converts the mouse X and Y positions on the screen into a 3D vector from the eyepoint, so that the relocation procedures can operate in object space rather than screen space. Furthermore, the system chooses a selection point on the object. If the object is one being directly manipulated by the user, that selection point will be the original intersection point of the cursor ray with the object. If the object is not being directly manipulated (for example, if a dynamic grouping test is being performed on the object, or the editor is being run in batch mode), the selection point defaults to the object's center of mass.

A relocation procedure takes the mouse position on the screen (integer X and Y on the window), the eyepoint and mouse vector in 3D space, the object and the selection point on the object, and the mouse button that was pressed (left, right, or middle). The procedure must return a 6 DOF rigid body transformation.

Once an initial 6 DOF rigid body transformation is determined, the system builds a special structure called the association data packet (ADP). The ADP is an abstract data type that represents the current position and associations of the object. A procedure can query the ADP for the initial positions of the object and its selection point or the current position of the object and its selection point. When the procedure wishes to add a motion to the object, the ADP's current position is changed via two functions, one which translates the object and one which rotates about an arbitrary axis. These changes are propagated to the current positions stored in the ADP. When the association process is over, the system queries the ADP one last time to get the final transformation of the object. The ADP also contains such information as which database and cell the object is in. The ADP allows the procedures to be more modular, as well as making queries to the object's current position easier and making modifications to the position more straightforward.

An association procedure takes a pointer to an ADP. Once it has applied any necessary local position changes to the object via the ADP, the procedure returns the modified ADP along with a list of any new associations of the object with other local objects.

Internally, the object association system provides a piece of "local storage" for relocation or association procedures that is freed between object manipulations; the procedure can use this local storage to store information between calls during an interactive motion. For example, the on-horizontal relocation procedure stores the initial mouse X and Y on the screen when the manipulation began, so it can compute the rotation based on the difference between the current X value and the original X value of the mouse.

3.3 Undo and Redo

A user manipulating a large building database will make many changes in an editing session; inevitably, mistakes will be made, and it is vital that an editor of any kind allow the user to gracefully back out of a bad operation. In the prototype editor, it was only possible to undo the last mouse operation done to the currently selected object. If a new object was selected, even if nothing was done to it, the ability to undo the last operation on the old object was lost. Also, the user could only undo the entire motion; if the object was moved twice without being deselected, it was impossible to undo the second operation alone. These limitations were due to the original construction of the undo procedure, which wrote a script file as transformations were made. Once a transformation on an object was finalized (that is, the mouse button was released), it was written to the script file, and the script was a write-only log. Finally, the prototype undo operated by storing an "old" transformation matrix associated with the current set of selected objects; this matrix was initialized with the original position of the object at selection time, and copied into the current position when the undo button was pressed. Obviously, storing only a matrix doesn't take into account renaming, copying, or deleting objects, so none of these operations could be undone in the prototype.

To allow for a better undo capability, it was necessary to store transformations internally rather than copying them to a file. A linked list stores atomic transformations performed on individual objects. This list is not written out to the file unless the user explicitly selects the "write script" function from the file menu. Each node keeps complete information on how to both perform and undo the operation designated by the node. A pair of general routines, which take a node as input, can either "do" or "undo" the operation of the node. The undo operation simply strips the last node off of this list and undoes the operation of the node (reverses the transformation, deletes the copied object, recreates a deleted object, etc). Once the list was implemented, infinite undo is trivial; each time the undo is pressed, another node is stripped off and undone. This structure also supports a straightforward redo operation; redo simply takes the last undone node, redoes the operation, and places the node back onto the undo list. For efficiency, undo and redo often don't actually perform the specified operations completely. For example, performing a deletion does not actually delete the object, because if it did, it would have to store a complete copy of all of the geometries and data comprising the object in the undo list node. Instead, the object is unlinked from the visibility list; when the operation is finalized, either by writing the database out to file or by the user quitting the editor, the object is actually deallocated and destroyed. This frees undo and redo from allocating or deallocating and copying the data to and from the block-structured binary database; they can simply add or subtract pointers from the visibility lists.

The interaction of groups of objects with the node structure proved to be problematic. For efficiency of representation, each undo node represents a single transformation of a single object. However, most of the time, the user is moving a set of objects with a single mouse motion. When the undo button is pressed, the user wishes to undo a single mouse "gesture," not a single atomic motion. To solve this problem, a special marking mechanism called a "chunk marker" was added to an undo node. When the undo button is pressed, nodes are undone until a node with a chunk marker is found; that node is the last node which is undone. Similarly, when the redo button is pressed, nodes are redone until a marked node is encountered. Chunk marks are inserted into the operation list whenever the user releases the mouse button. Using chunk marks, the system only requires four node classes: transform, copy, delete, and rename, with each node instance operating on a single object.

Given infinite undo and redo, when should the system delete nodes that have been undone? Most often, the user will undo a recent operation or a few recent operations and want those operations to disappear forever. Thus, the default is to delete the nodes on the redo list permanently each time a new operation is performed. This prevents the user from undoing an operation, then, later, pressing the redo button only to find that he has just accidentally redone the move of the desk from long ago. However, it is conceivable that the user may want to "insert" an operation; that is, undo a set of operations, perform a modification or modifications, and then redo the original set. Automated deletion of the redo list must be disabled explicitly for those rare cases where that behavior is desired.

Another useful property of this new format is that scripts can be loaded and saved independently of writing the changes to the database or UG files. The new script format can be loaded into the editor and "redone" onto a binary walkthrough database. Thus, scripts of moves can be saved to be reloaded and propagated back to the UG files at a later time, or can be sent as a compact modification format to be applied to other binary and UG databases. This system could be used as a baseline for parameterized or macro operations on databases.

3.4 The Knapsack

3.4.1 Motivation for Creating the Knapsack

There is no way to move the user's viewpoint while manipulating an object. In the prototype editor, this meant that an object could only be translated from one point to another within a given view. Moving an object from one visible point to another is straightforward; select the object and drag or warp it. Unfortunately, if the user wishes to move the object a long distance (say, from a storeroom in the middle of the building to the terrace on the roof), the paradigm breaks down. It requires many iterations with many intermediate positions in order to get the object to the roof, and it may require an extensive amount of planning for the user to get the constrained object to the final position (it may require navigating stairs or an elevator, while the object is continually constrained to the closest available surface). In fact, it may not even be possible to move the object from one arbitrary position to another by dragging. For example, if an object is constrained to be on a vertical surface and there is no connecting wall between the two points, there may be no way to slide the object along wall surfaces to get to the new position.

The intent of the prototype editor's copy button was that the user would select an object to be copied and press the button on the menu. The next selection of the object would create a copy of the object in the original location, which would be dragged by the mouse instead of the original. Several problems arose with this approach. First, there was the naming problem; what is the copy to be called? The original program did not change the copy's name; however, for the database to be maintained in a consistent manner, each copy must be given a unique name. One approach is to keep a counter of the number of objects of a given type; when a new copy is made, it is given the name (object)(number) and the number is incremented. This guarantees unique names; however, it requires the maintenance of a large number of counters, and reconstruction of the counters upon loading a new database. This counter maintenance problem is also compounded by our script-based update scheme; the user could create a script in one session, then save the script and exit the program, leaving the database itself unmodified. If the user then reloads the database later, makes some new modifications (which re-use the counter numbers generated in the first script), and subsequently re-loads the script, there will be conflicts between the names created in the new session and the names in the script. Such conflicts will exist regardless of whether a single, global counter or multiple, class-based counters are used. Another method of generating unique names is to append a random number instead of a count for each object type. This has the disadvantage of not being guaranteed to provide a unique name; however, it requires no counter maintenance, and given a large enough random number, will be very unlikely to fail. This second approach has been implemented and has caused no problems.

The second problem was with the paradigm itself. The copy performed by this method is necessarily delayed (i.e. the copy is not made until the object is moved). If the copy was made immediately, it would appear inside and exactly coincident with the original object, and it would be impossible to select one or the other reliably. Unfortunately, this also means that the object must store a flag that tells the system whether or not the object needs to be copied the next time it moves. This flag must be checked everywhere the object is moved or altered, making the code expand a great deal. If the user wanted to make many copies of something, a count must be kept rather than a flag. For example, the user might wish to populate a room full of desks with computer workstations. It would be nice under this paradigm to select the object, press the copy command multiple times, and then "strip off" copies to put on each desk.

Finally, long distance transformations of objects, already a problem for the direct manipulation paradigm, also interact badly with this copy paradigm. Often, the user wishes to make a large number of copies of an object and place them in many places. When placing desks on an empty floor, for example, the user will grab a desk and replicate it several times for each room. This is difficult to do when each object must be dragged from the position of another object of the same type.

3.4.2 The Knapsack Mechanism

Our solution to these problems is to provide an inventory or *knapsack* for the user. This feature follows the well known cut and paste paradigm common to many editing interfaces, or the inventory paradigm common to computer adventure games. In the classical inventory, the user places objects into an infinitely large backpack and carries them from place to place without cluttering up the viewport. The walkthrough editor knapsack is a dialog box with a set of buttons and a scrolling list. When an object is selected in the main window, the user can click on the "Cut" or "Copy" buttons in the knapsack window. Upon pressing the cut button, the object disappears from the main window and an entry appears in the scrolling list with the name of the selected object. Any objects that have been grouped with the selected object are also removed from the view; those grouped objects are stored internally, identified with the selected object, and do not appear in the list. The copy button leaves the selected and grouped objects in the main window; copies of these objects are made, and the name of the copy of the selected object appears in the scroll box.

The user can also add new objects defined in external UniGrafix files directly to the knapsack. A button on the Knapsack menu allows the user to load a UniGrafix file containing any number of definitions and instances of objects. One class is created for each definition, and one object is created for each instance and inserted into the knapsack. The selection points are set to the default, which is the center of the bounding box of the object. Once loaded, these objects and classes may be used normally.

Once the user has an object in the knapsack and the name of the selected object appears in the scroll list, that object is effectively removed from the database and resides in the knapsack. The object retains its original position, selection point, and object association data; however, it is tagged "invisible" and "intangible," so it cannot be interacted with in any way. A rename button allows the user to rename objects in the knapsack; the object's name is picked from the list and the new name is typed in. This is often called after the copy up function to rename the newly made copy. Deletion of objects in the database is accomplished in a similar fashion; the object or group is cut into the knapsack, the group is selected from the list, and the "Delete" button of the knapsack menu is selected, which destroys all objects in the group.

The user now moves in the normal walkthrough fashion until the area where the group is to put down is in view. The user can then select a group of objects from the scroll box and press either the "Paste" or "Paste Copy" buttons to remove the group from the knapsack and place it back into the database. Both of these procedures initiate an object association warp operation on the selected object. The paste operation then removes the objects from the knapsack list and re-marks them as visible and tangible. Paste copy produces a set of copies of all of the objects in the group and warps this set of copies to the desired location. The originals are left in the knapsack for further use.

3.5 Event Handling in the Walkthrough

The non-interactive WALKTHRU program used multiple parallel processes to improve its frame rate. Upon initialization, the database manager launches a secondary frame drawing process which was offloaded to a second CPU. The "cull" process generates "cull frames," a data type which includes pointers to all of the objects to be drawn in the frame, plus other necessary information such as the view frustum and eyepoint. These cull frames are placed onto a queue. The "draw" process operates on the other end of the queue, rendering each frame in double-buffer mode by fetching and drawing the polygons of the specified objects.

This process poses a problems if objects can be deleted in the time between when a frame is placed on the queue and the time that the frame is drawn. Unfortunately, modifications performed by the editor are made at the "back" of the queue, during the cull process. This causes problems if the editor deletes an object; it is almost certain that object is referenced in a frame on the queue, since the user was probably looking at the object immediately before deleting it. Furthermore, the editor continually creates and deletes "shadow objects" while performing dynamic grouping searches. This interaction causes segmentation faults, as the draw process attempts to draw nonexistent objects.

One solution is quite simple: disable the separate drawing process. This obviously slows the walkthrough down, but the slowdown is not generally perceptible. A better solution would be to have the editor store objects to be deleted in the cull frame structure; after the draw process finished drawing the frame, it would actually delete the marked objects. Since no object can appear in frames after the frame in which it was deleted, this solves the problem without disabling the separate draw process. WALKEDIT is still using the former solution.

3.6 UniGrafix Writeback

3.6.1 Motivation

An editor for a virtual environment is useless if changes made by that editor cannot be permanently entered into the database files that represent the environment. However, propagation of changes made in the editor to the database is a nontrivial task for the walkthrough. This is a result of the fact that the walkthrough uses two database formats: binary and UniGrafix. These formats have different purposes and are used in different ways.

The basic format for a walkthrough database is a set of floors. Each floor is represented by a set of UniGrafix (UG) ASCII files: one special file which describes the walls and floors, and one file for each room or major subdivision (such as a hallway) describing the contents and furnishings of that room or subdivision. This organization allows the database construction and preprocessing algorithms to determine which are the proper polygons to do the spatial subdivision on for visibility computations (the file containing the wall and floor polygons). It also serves to localize room and hall contents to simplify manual modification of the database via textual modification of the UG files. Note that UG files store only geometry; they have no organizational or visibility information in them.

The binary format is what is actually used when running the walkthrough: it stores objects in packed C data structures, and contains the visibility and spatial subdivision information needed for real-time interactive rendering. When a database is compiled, a .wk (binary) file is generated from the various .UG files and subdirectories. This single .wk file is the run-time database format used by both the editor and the standard WALKTHRU program; generating it from the UG files can take several hours of CPU time. This binary format file is the only database that the editor sees after it has loaded the building. Run-time modifications, including editor operations, are made to the binary database. These modifications show up in the current editor session, and are propagated back to the .wk file when the editor is shut down. However, they are not automatically propagated back to the UG ASCII database files which are the basic data exchange format used for all of our graphics programs. To address this issue, a set of routines for propagating binary database changes back to the original UG ASCII files was necessary.

3.6.2 Naming Issues

The binary format associates a unique textual name with each object in the database. These names are generated by the instance statements in the UG files describing the floor; during construction of the binary database from the UG database, the "root" UG filename from which the instance statement of the object was encountered is prepended to the name of the object. For example, a sphere called "sphere1" whose instance statement is in a file included in the database file "Zsphere.ug" is called "Zsphere.sphere1" in the binary database.

Nominally, these names should serve to identify the file and exact instance statement from which the manipulated object was derived. However, the UG format allows arbitrary nesting of include files. Thus, the instance statement creating the object could be in any descendant file included in the root file specified in the object's name. The system needs to track the filenames which are included via explicit include statements in the UG file; these filenames may contain the instance statements from which the objects were derived. These filenames may or may not be in different directories than the original UG file, so finding the files can be a problem.

3.6.3 The Writeback Process

The input to WALKEDIT's UG writeback process is a sequence of elementary operations; these include transforming an object, copying an object, renaming an object, and deleting an object. These operations are assumed to be a legal sequence in time; for example, an object must exist when a copy event is handled. This input list is condensed into a set of *modifications*. A modification consists of an "old" object name from the original UG database, the (potentially identical) new name for the object, a single transformation of the object from the position of the "old" object before any transformations to the final resting place of the "new" object after all transformations in the editing session, and a flag indicating whether or not to delete the object from the database. At most one modification node exists for any one object in the final database.

After the modification list is made, the UG ASCII database files are sequentially scanned. The initial queue of UG files to be processed is the union of all filename prefixes of objects in the modification list. When a file is scanned, statements other than instances and includes are ignored. An include file adds the name of the included file to the file queue if it has not been scanned already. An instance statement is checked to see if the instanced object is the "old" object of one or more modification nodes. If it is, the statement is replaced with set of new instance statements, one for each modification node that has the object as its parent and is not flagged as deleted. Each new statement has the "new" name of one of the modification nodes as the instance name, and the body of the statement is replaced with the old statement body plus the modification node transformation. This process generates a modified UG ASCII file. If the modified file has changed from the original in any way, the original is replaced with the modified version; otherwise, the modified file is simply deleted. This process is repeated until the file queue is empty.

A few other features help reduce the amount of work necessary on the part of the user. Paths are not stored with the object names; thus, the system will often not be able to find the file in the current directory when it tries to first open it. Rather than forcing the user to specify the filename with a full path each time it needs to open a file (and most databases have hundreds of files), the system keeps a "working directory list" which initially contains only the current working directory. Each time an instance statement is processed, the path in the instance statement is added to the directory list, and each time the user is forced to enter a path, that path is added to the directory list. When a new file needs to be opened, the system attempts to open it on every path in the directory list before it is forced to ask the user where the file is. Usually, most of the files will be in a small set of directories. In theory, this bookkeeping allows the system to ask the user at most once for each directory in which database files are stored. In practice, behavior is even better; pathnames are often stored in the include statements themselves, so the system normally only asks the user for a single pathname to get the very first root file, and all other necessary paths are deduced by the system.

3.7 Transformation Consolidation

Another feature of the UG writeback system is instance statement compression. When a human user generates UG files by hand, especially without the editor, the tendency is to produce multi-line instance statements with many redundant rotations and transformations on them. This leads to huge and cluttered files. Furthermore, if the UG writeback simply worked as stated above and postpended the transformations to each line, each use of the editor would cause the instance statements to grow. The file sizes for the UG database would expand unnecessarily. To remedy this, we provide a compression switch for the writeback process; compression is active by default. When compression is active, the system reads and processes each instance statement into a single transformation matrix. The transformation in the modification node is added to this matrix to produce a single matrix transformation from the world origin. This transformation can be described by at most 12 parameters in a UniGrafix instance statement, stored as a matrix and output as a set of mirrors, scaling, translation, and rotation parameters about major axes. Thus, each instance statement in the file, no matter how long, is reduced to a single zero to nine parameter statement in the newly generated UG file. This has two beneficial results; the editor in compression mode never results in UG file explosion, and old UG files made by human trial and error can be substantially reduced in size simply by running them through the editor's writeback procedure.

Transformation consolidation takes place during the sweep-and-update pass through the UG ASCII files. If consolidation is not being performed, an instance statement of an object that has not been modified is left as it is, and an instance statement of an object that has been modified has the transformation matrix of the modification postpended to the existing instance statement. If consolidation is being performed, every instance line is rebuilt. Reconstruction is performed by computing the transformation matrix that the instance line gives for the initial object position in the database, then postmultiplying any new transformation to that initial matrix. The complete transformation matrix is then decomposed into a single set of mirrors, scales, rotations, and translations (at most one of each per axis), and the instance line is rewritten with only this single, orthogonal set of operations.

4 Realism in Motion and Manipulation

4.1 Simulated Gravity

In the real world, gravity is a major element of our daily life. Both our bodies and the objects we manipulate obey physical constraints, and it is almost impossible to create a believable interactive environment without providing some Newtonian behavior to both the user and the objects in the world.

In the original walkthrough, there were two modes for user motion. The normal mode constrained the user to a particular height in the universe; this height was set to an initial value upon loading the database, and the only way to change the height was by pressing "up" and "down" keys, which moved the user some fixed amount (usually a few inches) up or down. The mouse allowed walking forward or back while rotating in the plane of the floor. Aside from the obvious deficiency of not allowing the user to look up or down, this mode also prevents simple or realistic motion for standing on top of objects or for moving between floors. The other mode, affectionately called "F16 mode," allows the user to spin vertically as well as horizontally, and move straight ahead along the current view vector. This provides a standard "fly-through" capability; the user may point himself in any direction by rotating into that direction, and may go anywhere by pointing at the desired location and moving forward. This is no more realistic than the standard mode; the user floats through the air like a miniature space ship, with no inertia or gravity, flying up and down stairwells and across three-story drops without flinching. It is very easy to get lost in this mode, as the vertical freedom of motion makes it difficult to simply walk down a hallway; the user ends up drifting into the ceiling or the floor, changing height in an unnatural way.

We want behavior that combines the steady height above the floor while walking down the hallway with the ability to look around from that normal height and to walk up and down stairs, take elevators, and generally move around the building in a manner approximating the way a real person would move around the real building. The latest implementation of WALKEDIT provides a gravity approximation that yields this behavior.

The database model allows the editor to project rays from any point in space and returns the first intersection of that ray with a solid object. Our first attempt at simulating gravity was to insert a process that continually projects a ray from the user's eye point straight "down" (that is, in negative Z). On its first call, the process stores the initial distance to the closest surface below the eye point. This distance becomes the "height" of the user. On subsequent calls, the system projects the ray and subtracts the height of the user to get a current height error.

When the height error is nonzero, the user's height must be adjusted to reduce the error. If the error is negative, the user's eye is too close to the floor. In this case, the user needs to "stand up." The normal human muscular system provides a smooth motion to move the person from a squatting position to a standing position; this motion is initially rapid and slows as the extent of the legs are reached. To simulate this process, the proportional correction method was borrowed from control theory. Every cycle, a large percentage (user selectable, which defaults to 75 percent) of the error in the height is eliminated (that is, the user's height is increased by that percentage of the error). This provides a smooth rise, rapid at first, slowing as the proper height is reached.

For a positive error, the user must "fall" to the appropriate height. This is done with a simple discrete time acceleration simulation; the user's vertical velocity is maintained from cycle to cycle, and a fixed acceleration term (also selectable) is applied to that velocity every cycle. The velocity is applied as a change in height in each frame. Note that the velocity can be positive as well as negative; in this case, the user will fly up and then down in a parabolic arc as if thrown upward. If the user has a large downward velocity but the height error for the frame is negative, the user has fallen and struck the floor. In this case, a damping term is applied, and the velocity is reflected about the floor surface. Thus, a small fall will result in the velocity being cancelled by the damping term (as in, for example, stepping down from a short platform or dias), so there is no bounce; but stepping off of a second story balcony will result in a perceptible pogo stick action.

Numerical instability was observed near the user's set height, due to the relatively long virtual time interval between iterations of the process. To cure this problem, for "small" downward steps (if the user's height error is positive but less than half of the user's height) the proportional adjustment is applied to gently lower the observer to the floor. This adjustment interacts with the damping term to provide a gentle, smooth landing, as well as preventing vertical oscillation near the proper height.

This method provides realistic motion on nondegenerate (i.e. continuous) floor surfaces. The user can walk up and down stairs by simply moving over them; the user steps smoothly up or down onto each stair. Moving rapidly over many small vertical steps (such as walking up a staircase) doesn't allow the system to fully process a step before the next one is encountered; the result is a steplike vertical motion corresponding in period to the spacing of the steps themselves. Taking an elevator or moving lift is also handled by this process; the observer rises or falls with the floor of the elevator.

In fact, it was the elevators that revealed another problem with this approach. The elevator doors had a finite thickness (an inch or so). This meant that between the floor and the interior of the elevator was a small crack the width of the elevator door. Such a tiny crack wouldn't bother a person with a finitely large footprint (indeed, real elevators have such a crack, if it is only a fraction of an inch wide). Unfortunately, the observer projects an infinitely thin ray; thus, when stepping over the crack, the user fell through into the elevator shaft. This same problem can result with any arbitrarily tiny crack in the floor, even a small numerical error in the modeling.

The first reaction to this problem is to create a finitely large "footprint" for the user; if any portion of this footprint can get a hold, the user won't fall through the crack. However, our system only supports ray projections, and the projections are not inexpensive; thus, we must make do with a small number of rays instead of a footprint. Furthermore, such rays may have problems in some cases; for instance, if the rays are projected in front of the user and the user runs face-first into a wall, the front ray will be projected on the other side of the wall, and the resulting height error can be odd.

Since the problem is cracks in the floor, and the user will be crossing those cracks perpendicularly, a ray projected some distance in front of the user should be able to find an available patch of floor if one is available. The system projects a second ray from one fifth of the height of the user in front of the eye point, straight down. This projection yields another error estimate on height. However, since this error estimate is to be used to avoid falling through cracks, we use that estimate only if the error under the user's eyepoint indicates a fall. If the user's eyepoint height error is stable or indicates a rise, that value is used. If the eyepoint height error shows a fall and the forward height error shows a lesser fall or a rise, the forward error is used. This stops the user from falling through cracks, prevents problems with the user stepping up to or close to the edge of a balcony or precipice, and provides the ability to get a "toehold" on surfaces just ahead of the user. In practice, this gravity system seems to produce the desired behavior.

4.2 Physical Object Simulation

4.2.1 Motivation

In the design and construction of WALKEDIT, we often wanted to have the ability to do various degrees of physical simulation. Part of the goal of our object association technique was to be able to make objects behave similarly to the ways they behave in the real world: when objects behave naturally, manipulation of those objects can be far more intuitive. Unfortunately, true collision detection and dynamic simulation are time-consuming algorithmic processes, dominated in most cases by the former. Because the walkthrough had an incompatible "higher goal," that of interactive frame rates, we had discounted dynamics as a viable alternative.

However, in the recent past, algorithms and techniques have appeared in the literature to support real-time collision detection and dynamics. Specifically, here at Berkeley, Lin and Canny have developed an algorithm that can maintain and quickly update the closest feature of two convex polytopes. With this algorithm, doing limited dynamic simulation should be possible while maintaining the frame rates that make the walkthrough program interactive. Once we have this capability, we can reach further with both our level of realism and our interactive techniques.

Research into realistic rendering and visualization techniques is an ongoing effort for the walkthrough. With the ability to do dynamic simulation, we can make objects settle properly; if there is a table with a pair of legs in the air, that table will no longer float above the floor, but will fall to its natural resting position. If a cup is sitting on an incline, it will not stay still, but will slide to the next lower surface. If a book is hanging over the edge of a table, it will fall off and hit the floor. Such events will make the walkthrough a much more realistic experience.

Our research into interactive manipulation techniques will also benefit from dynamic simulation capability. The ability to do true collision detection will enable the pseudo-gravity association to settle objects to the supporting surface realistically. Further user interface research will also be able to make use of dynamics and collision detection: for example, we could implement a "rubberband" dragging interface, where "tugging" an object with the mouse produces a force that drags the object toward the cursor position. Such a force-based approach to movement has advantages for alignment of objects. Aligning two desks, for example, is as easy as pressing one against the other; the torques generated by their collision will force them to rotate and press their backs evenly together. These new approaches will negate the need to for the user to do some of the explicit constraint of objects necessary in the absence of collision detection. Another possible use is in design of buildings for construction. Is it possible to move a piano up a staircase? If we have dynamics and collision detection, we can find out by trying to drag it up the stairwell.

This section of the thesis describes a preliminary effort to integrate the Lin-Canny algorithm and a fast

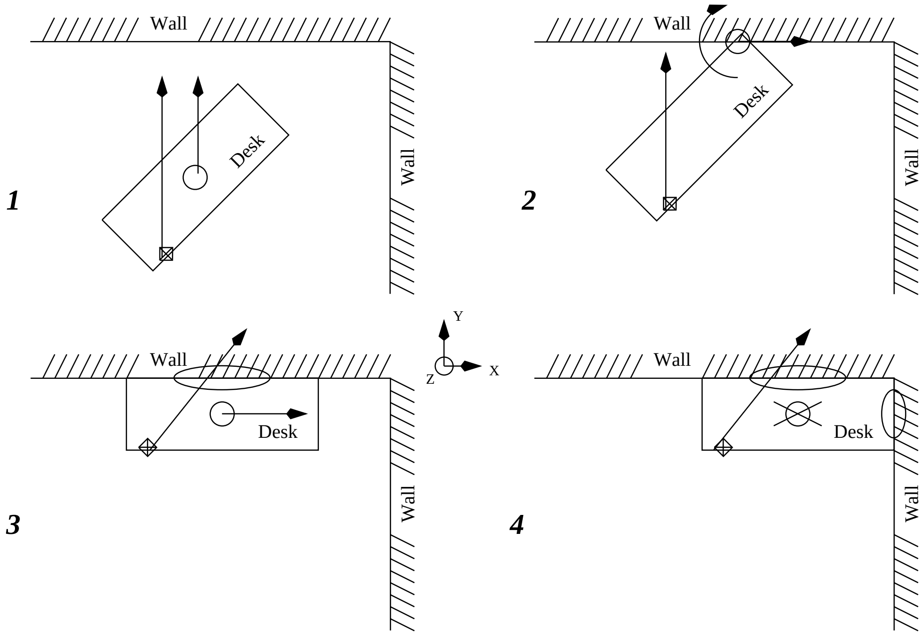

Figure 22: An example of how collision detection and force simulations can benefit object manipulation. The user wants to put a desk in the corner. At first, the user simply pulls the desk toward the upper wall (**1**). The desk moves linearly with the mouse until the collision detector finds an impact with the wall (**2**). The torque produced at the contact point (the circle in frame **2**) forces the table to rotate so that its backside nestles against the back wall (**3**). After that, the sliding contact against that wall (the ellipse in frame **3**) keeps the table aligned with the back wall, but allows the user to slide the desk into the corner by angling the applied force. Finally, a sliding contact is achieved when the collision detector finds an impact with the side wall (the ellipse in **4**). The desk stops moving, with all DOF removed by physical contacts and the input force, properly nestled into the corner. Note that all the user had to do was grab and drag.

contact force computation algorithm into a special, standalone testing version of WALKEDIT, called WKCM (WalKthrough with Collision Modeling). Although the algorithms are not integrated into the object association system yet, our initial results with WKCM indicate that the approach shows promise as a tool for interactive, realistic manipulation of objects in a virtual environment.

4.2.2 Integrating the Lin-Canny Algorithm into WKCM

Object Representation

Representing objects for simulation is a problem in the walkthrough. The spatially partitioned, preprocessed run-time binary WALKTHRU database (the WK database) is generated from UG source code, but it does not maintain the UG winged-edge data structure during the actual walkthrough. In the WK database, an object is represented as a list of unrelated 2D facets, each of which has its own separate list of vertices. This representation is not compatible with the Lin-Canny algorithm, which requires a convex decomposition of each database object with a winged-edge structure for each convex subpolytope. Furthermore, simulation requires the storage of a center of mass, mass, and inertia tensor with each object.

Due to the fact that the walkthrough and Lin-Canny algorithms have very different object data needs, WKCM maintains a second runtime database, called the physical database. For each object class in the WK database, the physical database contains a convex decomposition of the object into subparts with a winged-edge format, the relative poses of the subparts, and the physical parameters of the object such as the center of mass, mass, and inertia tensor of the object in its local coordinate frame. This database is initialized from two auxiliary files that must be precomputed for each object and must be present in the same directory as the WK database at load time. One of these auxiliary files holds the winged-edge convex pieces; the other contains the simulation information (inertia tensor, mass, and center of mass) and convex decomposition information for each class. At load time, all of this physical information is gathered into a set of data structures called *polyobject* structures. There is one polyobject structure per WK object class containing all physical data for the class. When the simulation algorithms need to operate on a WK database object, they request a polyobject representation of that object from the physical database.

A separate utility program was written to assist in the creation of these files. The *WKCM object converter* takes as input the original UG file for the WK database and creates the winged-edge structure, center of mass, and mass for each object class in the file. This data is then written to the auxiliary files in the format required by WKCM. Center of mass of a polytope is computed by finding a point inside the polytope and taking the mass-weighted average of the centers of mass of all pyramids formed with the given point and each facet of the polytope. Currently, the converter does not perform the convex decomposition or compute the inertia tensor; further efforts in simulation will require coding these algorithms.

Object Linking and Closest Feature Maintenance

The Lin-Canny algorithm dynamically maintains the closest pair of *features* of two convex polytopes. Features are defined as points, edges, or facets of the polytope. The algorithm is incremental, with two main procedures. The first procedure is an initializer; when simulation is begun, the initializer is called to determine the closest features, which must then be stored by the caller. The initializer runs in linear time in the number of facets of the polytopes. When the objects are moved, the incremental procedure is called with the previous set of closest features as a starting point. The procedure "walks" the surface of the polytope from the old closest features to the new closest features and returns them. Because the

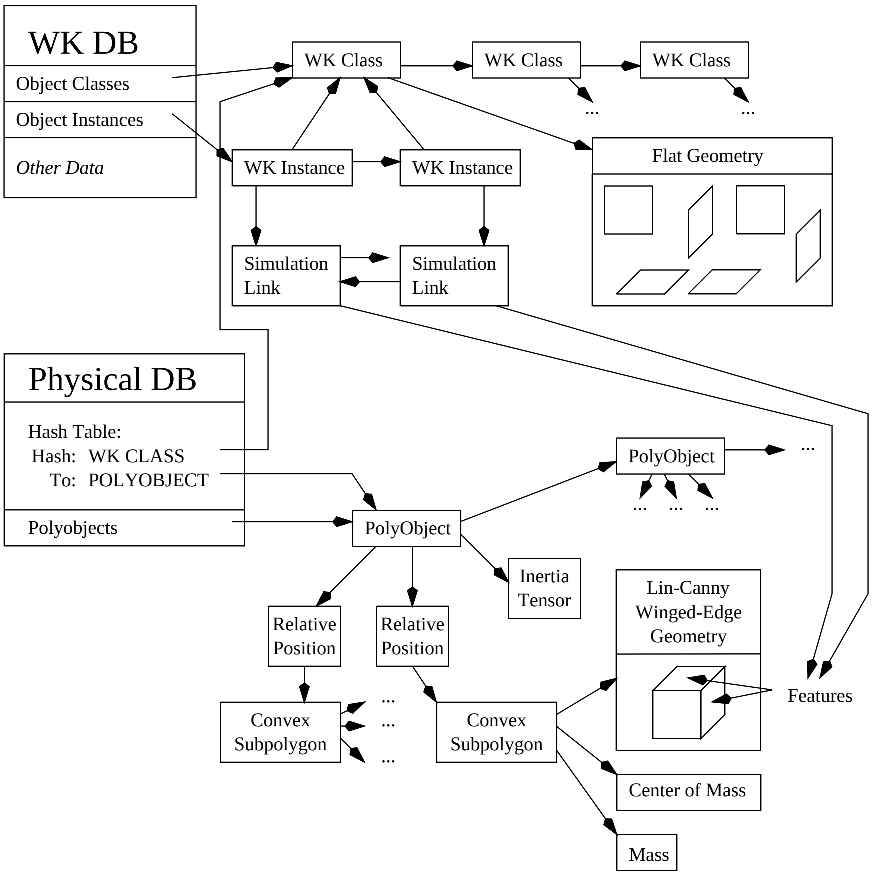

Figure 23: Organization of run-time databases for simulation. The WK database is the original walkthrough database, with "unstructured" object definitions and walkthrough object and class data. The physical database contains a more structured winged-edge object definition, plus simulation-oriented object data such as center of mass and moment of inertia.

old features are used as a starting point, finding the new features is very fast if the objects only moved a small amount; given the "small motion" assumption, run time of the incremental update is expected constant [33].

Brian Mirtich's implementation of the algorithm [34] was ported to the Walkthrough environment and integrated into WKCM. The polyobject representation uses Mirtich's structures for the convex subpieces, so there is no translation layer involved; the library can directly call Mirtich's functions on the relevant subpart of two polyobjects to find their closest features. However, in order to provide maximum performance for closest-feature determination between a pair of objects, the system must maintain the closest features of the convex subparts of the objects between calls to the closest feature finder.

Closest features are maintained via the linking mechanism implemented in the WKCM simulation library. When the user wishes two database objects to interact during simulation, a library function is called that allocates a link between the objects. Each WK database object has a single additional pointer added to its external data field, which points to a singly linked list of *simulation links*. Each simulation link is a part of two such linked lists, one for each object being linked. The link contains pointers to the objects, pointers to the next elements in the respective lists, and a matrix of the closest features of the subpolytopes as of the last call to the Lin-Canny algorithm. This matrix contains $2nm$ entries, where n and m are the number of subpolytopes of the two objects. Each entry is a single void pointer that points to a feature in the object class representation of the subpolytope, making the link as compact as possible.

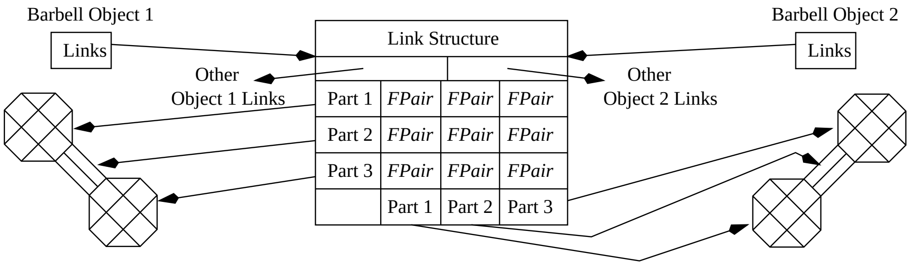

Figure 24: An example of a link structure between two barbells, which are represented by two convex weights and a convex cylinder connecting them. The link is a member of both barbells' link lists, and contains a 3 by 3 matrix of closest features (labeled *FPair*) of the subpolygons.

The Lin-Canny algorithm itself is accessed by the rest of the system via a function call that takes two database objects and a transformation matrix for each. If there is a link between the objects, the function retrieves the previous closest features from the link structure and uses those as the starting point to find nm new closest features, pairwise for each subpolytope, on the polyobjects transformed by both the object's current pose and the given matrices. The smallest distance between any two subpolytopes is the smallest distance between the polyobjects. This smallest distance is returned, and the link is updated with the new closest features. The linkage system interface provides transparent closest-feature determination at the granularity of (potentially concave) database objects to the rest of the WALKTHRU system.

Linking Strategy

When the collision detection algorithm is invoked with an object and a planned path for the object, the object links are traversed to find which other objects in the world can possibly interact with the selected object. Thus, the programmer can control the extent of the interactions between objects by making and deleting links as the objects move around; if an object moves to a different room, for example, it can be unlinked from the objects in the old room, and those objects will not be considered for collision with the moved object. This allows the system to take advantage of the cell subdivision structure to greatly reduce the number of objects that have to be included in collision tests when moving a particular object interactively. In WKCM, a callback is registered with the object associations' grouping function to link every object touched by the grouping search to the selected object. This has provided satisfactory local linking, but may be inadequate for longer-distance movements of objects. In the current version of WKCM, unlinking has not been implemented.

4.2.3 Collision Detection with Moving Objects

Once links are established, the user may use the collision detection routine on any database object. The current collision detector takes a database object, a motion vector for that object, and a maximum time value. The motion vector is a complete 6-element 3D velocity vector, with a linear velocity component and an angular velocity component, where the angular velocity is specified about the object's center of mass. The function computes the following: Starting with the object at its current position, for what maximum value of t, $0 \leq t \leq t_{max}$, can the object move by vt, where v is the given velocity vector, before the object collides with an object to which it is linked?

This function uses a numeric solver on $d(t)$, the minimum distance between the moving object and any other linked object as a function of time, to find t_c, $0 \leq t_c \leq t_{max}$, for which $d(t) \geq \delta$ for all $0 \leq t \leq t_c$. For numerical stability, no objects may get closer than some preset small δ; values used in our tests are about one thousandth of an inch.

In creating this function, the properties of the Lin-Canny algorithm were very important. The algorithm cannot handle interpenetration or even contact between objects; if such a condition occurs, an infinite loop is created. Thus, it was imperative that the collision detector be very conservative, never even computing $d(t)$ for any value of t that might result in interpenetration or contact. To achieve this behavior, we use a "stepping" algorithm. We start at $t = 0$ and attempt to step forward in time as much as possible with each iteration while remaining absolutely safe. Since our velocity vector is rigid and linear, if we know $d(t_1)$ for some t_1, we can compute a time step bound t_{step} for which $d(t) \geq 0$ for all $t_1 \leq t \leq t_1 + t_{step}$. t_{step} is the maximum delta time such that any point on the object can move at most $d(t_1)$ in that time. If v is the translational velocity, a is the greater of the angular velocity or π, and r is the radius of the bounding sphere of the object, we can write

$$vt_{step} + art_{step} \leq d(t_1)$$

or

$$t_{step} \leq \frac{d(t_1)}{v + ar}$$

The collision detection algorithm, in pseudocode, works roughly as follows:

1. Set $t = 0$;
2. While $t < t_{max}$ and $d(t) \geq \delta$:
    (a) Compute t_{step} for distance $d(t)$;

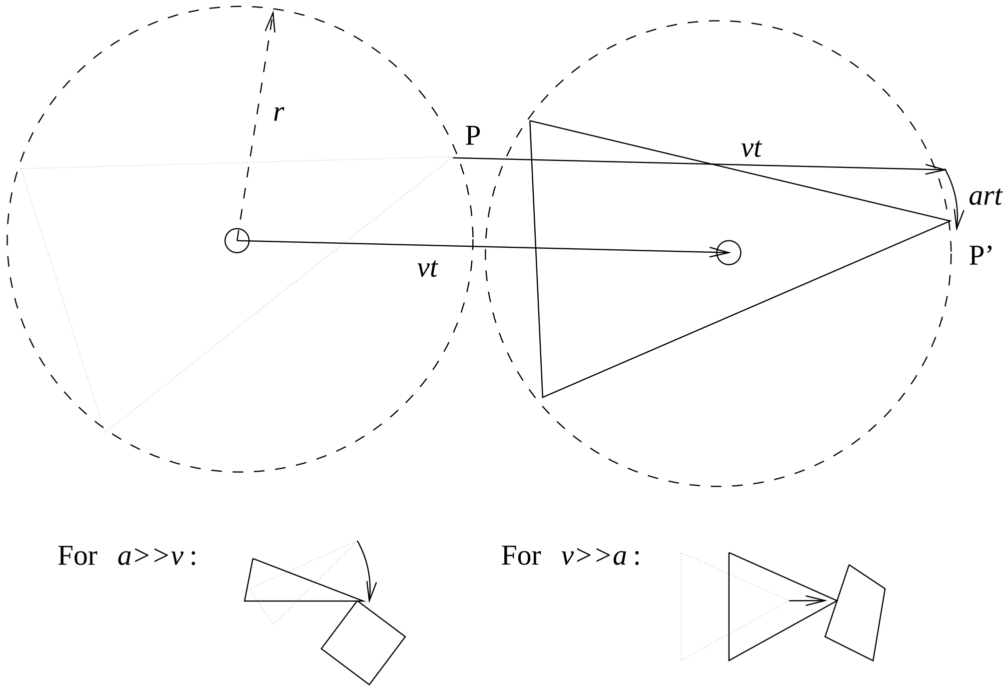

Figure 25: An illustration of computing a conservative maximal timestep that avoids collision between the moving convex polyhedron and a stationary polyhedron. The large figure demonstrates that for a polyhedron with linear velocity v and angular velocity a, the largest distance that any point P on the polyhedron can move in some time t is $vt + art$. The lower figures show qualitative examples of how the two terms relate to collision distance and time for the $v >> a$ and $a >> v$ cases for a pair of hypothetical polyhedra. For $a >> v$, the closest distance $d(t_1)$ between the objects at the beginning of the interval is less than or equal to $art_{collide}$ where $t_{collide}$ is the time step from t_1 to a collision. In fact, the approximation becomes arbitrarily good (i.e. "less than or equal to" becomes "equal to") as $t_{collide}$ approaches 0. Similarly, for $v >> a$, $vt_{collide} \geq d(t_1)$. Combining the equations gives $vt_{collide} + art_{collide} \geq d(t_1)$, so $t_{collide} \geq \frac{d(t_1)}{v+ar}$. Thus, if we pick a time step less than or equal to $\frac{d(t_1)}{v+ar}$, that time step is guaranteed to be smaller than the time to collision.

(b) $t = t + t_{step}$;

(c) if $t > t_{max}$ then $t = t_{max}$;

3. if $t = t_{max}$ then return t;
4. Set t to the last valid time when $d(t) \geq \delta$;
5. While $d(t) \geq \delta + \delta^2$:

   (a) Compute t_{step} for distance $d(t) - \delta$;

   (b) $t = t + t_{step}$;
6. Return t;

The first loop attempts to advance t to t_{max}. If this loop fails, then a collision occurs; the second loop backtracks and attempts to step as close as possible to δ distance between the objects before returning a value.

Note that this program loop is not called on all objects to which the selected object is linked. Before any motion is attempted, a culling step is performed where the bounding sphere for the selected object is swept along the velocity vector and compared with the bounding spheres of all of the linked objects. Only linked objects whose bounding spheres intersect with the selected object's bounding sphere somewhere along the path are tested in the main loop. This provides a large advantage in efficiency, since the bounding sphere computation is very fast, is only done once, and tends to remove all but a few of the linked objects from consideration as blockers.

This relatively simplistic approach has been found to produce interactive collision detection on objects being dragged with the mouse in WKCM. Our approach is similar to other approaches in the collision detection literature that focus on a preliminary bounding shape cull step or spatial subdivision to reduce the number of potential collisions from $O(n^2)$ to a more tractable set. We combine both a spatial cull (in the form of the linking phase) with a bounding sphere cull. Approaches for the spatial cull step in the literature focus on structures like octrees that automatically partition the objects into groups; our algorithm simply uses the precomputed cell structure of the WK database. Our bounding sphere cull is also similar to, but simpler than like approaches in the literature, such as four-dimensional sweep volumes [13, 30] or spatial tiling techniques such as that used in [34]. To a large degree, such complex culling techniques are made unnecessary by the existing WK cell structure, which produces a comparable result to these approaches in practice.

Once potential collisions have been quickly pared down to one or two other objects by the cull step, our numerical solution takes time inversely proportional to the closest distance that passes between the objects in the motion. A more standard numerical method would likely improve performance, but we were concerned about the infinite-loop problem with the Mirtich implementation, so our approach sacrifices efficiency for a guarantee of non-intersection. A newer public domain implementation of the Lin-Canny algorithm, called I-COLLIDE, that avoids this problem has been recently presented by Cohen et. al. [16]; we will be replacing the Mirtich code with the I-COLLIDE system in the near future.

4.2.4 Contact Force Computation

For physical simulation, we need a contact force computation algorithm as well as a collision detector. In WKCM, we have implemented a contact force computation algorithm taken from a paper by David Baraff in SIGGRAPH '94 [4]. The paper points out that you can model contact forces on polyhedral, frictionless rigid objects as point forces of two major types. Vertex to face contacts generate a force at the vertex-face intersection point, in a direction purely normal to the face. Edge to edge contacts generate a force at the intersection of the two edges in the direction of the cross product of the direction vectors of the edges. We can specify the directions of the forces because the frictionless nature of the problem implies that contact forces must be purely normal to the contact surfaces. All other types of contact may be decomposed into these two primitive types. For example, face to face contact may be modeled by considering the region of overlap of the faces: each of the vertices of the polygonal overlap region will be either a vertex-face or edge-edge contact, and this set of contacts completely describes the contact force between the two faces.

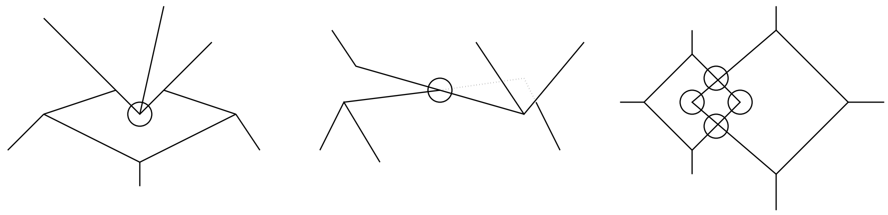

Figure 26: The two major types of contact between objects: vertex-to-face (left) and edge-to-edge (middle). On the right is an example of how another type of contact, face-to-face, is decomposed into the other two types. Two diamond-shaped faces are touching to produce two edge-edge contacts and two vertex-face contacts. The contacts are circled in each picture.

To compute the contact points between colliding objects, WKCM takes advantage of the available convex decomposition. The pairwise closest features of the subpolytopes of the objects are stored in the link structure. The only valid contact points that can exist between a pair of subpolytopes are the immediate neighbor features of the closest features of those subpolytopes. For example, if a vertex v of convex polytope 1 is closest to a face f of convex polytope 2, the only valid contacts can be between the edges and vertices on facets incident to v and the edges and vertices on face f. Thus, for each of the mn pairs of Lin-Canny closest features, we need only consider the neighbor features of each of those closest features to find all possible contacts. Since there are expected $O(mn)$ of these neighbors, we have reduced the number of comparisons between features necessary to compute all of the contact points from a pairwise comparison of all features of each object to a comparison of only $O(mn)$ features, a substantial savings. Furthermore, a trivial reject for a pair of subpolytopes can be performed in the case that the stored distance between the closest features is smaller than 2δ.

Once the full set of contact points and force vectors at each contact point are determined, Baraff's algorithm comes into play. We will define two vector quantities, the relative acceleration a_i and the contact force f_i at each contact point p_i. a_i is the relative acceleration between the two bodies at p_i; a positive value means the objects are moving apart, and a zero value means they are sliding or pushing against each other. f_i is positive if the objects are pushing together at p_i, and zero of the objects are moving apart. From this description, it is clear that $a_i \geq 0$, $f_i \geq 0$, and $f_i a_i = 0$ for all i, since objects can either be moving apart or pushing against each other, but not both. Also, from physics,

$$\mathbf{a} = \mathbf{Af} + \mathbf{b}$$

where $\mathbf{A}$ is a matrix representing masses and inertial forces, and $\mathbf{b}$ is a vector representing external forces such as gravity. These conditions define a *linear complementarity problem* or LCP. Baraff's SIGGRAPH paper [4] gives an algorithm for solving it.

Given the solution for the forces f_i, we multiply the previously determined contact force direction times these values and apply the forces to the center of mass, giving a single 3D force and torque vector that represents the total force on the object at the given instant in time. Given the total translational and rotational force (f and τ) and the mass (m) and inertia tensor (I), we can determine the acceleration of the body (a and α) by Newton's and Euler's equations $f = ma$ and $\tau = I\alpha$. Since WKCM limits the dynamics simulation to "pseudo-static" conditions (described below in section 4.2.5), i.e. no momentum or velocity, we do not need to include the full formulation of Euler's equation $\tau = I\alpha + \omega \times I\omega$, which includes angular velocity ω.

4.2.5 Pseudo-Static Simulation with Contact Forces

The provided functions can now be used to generate simulated dynamic motion of objects. We call the contact force computation routine, which returns a net force on the object. We call another routine which converts this to an acceleration. The acceleration and object mass is used to compute an instantaneous velocity vector and is passed to the collision detector, which determines how far the object can move in the direction of the acceleration. If the object cannot move at all, it is stable, and the simulation is completed. If the object can move, it is moved as far as possible, and the process is repeated at the new location, which will in general have an entirely new set of contact forces.

The interaction of the collision detector and the pseudo-static force computation algorithms in the simulation loop causes some difficulties. Since the math is subject to small numerical fluctuations and the accelerations are not really being applied "properly" (as accelerations on a continuously updated velocity vector), the collision detector can cause objects to get "stuck." This usually happens when a sliding contact is indicated by the algorithms. Sliding contact involves a motion that is very exactly parallel to the two surfaces. Even double-precision arithmetic is not enough to make the velocity vector exactly parallel the surface, so the object will tend to try to move infinitesimally toward the surface. The collision detector then stop the object from performing any motion at all, since the motion would cause the objects to get a tiny bit closer together than δ. Since the object doesn't move, the contact forces don't change, and the object gets jammed in a nonphysical position.

The present solution to this problem is to have the contact forces repel the object slightly. A repulsive force proportional to some user-set constant times the mass of the object times an inverse square of the distance between objects is applied at each contact point. This tends to cause the objects to want to separate slightly at their contacts, giving the collision detector some slack and allowing the object to slide along the surface. This solution still gets stuck sometimes, but it can be "un-stuck" by increasing the repulsion value. This suggests an adaptive algorithm which checks to see if the collision detector is forcing the object to remain in an unbalanced position, and if it is, increases the value of the repulsion slowly until the collision detector "lets go" of the object. Such an algorithm is a part of intended future work.

4.2.6 Testing Setup

WKCM, a specially modified version of WALKEDIT, integrates the simulation library into the normal WALKEDIT user interface. There is a new control panel in the options menu with three controls. The first is a checkbox that turns on collision detection for normal interactive motion. While this option is active, any motion the user applies to an object with the standard editor operations is collision checked, and if a collision is detected, the object is only moved far enough to collide. This demonstrates the speed and interactivity of the collision detection and linking mechanisms.

The second control activates simulation. When this is on, the user may click and hold on an object, and that object will be subjected to the simulation loop until the user releases the mouse button. The user can cause blocks to settle, octahedra to fall over, or bricks to tip off of surfaces. The third control is a numerical entry box that allows the user to control the repulsion force parameter, increasing it if the objects get stuck or decreasing it if the objects are "bouncing" too violently (which happens when the repulsion gets too large).

4.2.7 Results

Unfortunately, due to the lack of a good convex decomposition routine, we have not been able to test the collision detection or pseudo-static physics algorithms within the "real" Soda Hall environment. WKCM has been run with a simple "blockworld" environment containing geometric objects and constructs whose inertia tensors and decompositions could be determined by simple mathematics. In that environment, two things became clear: the collision detection algorithms were definitely quick enough for interactive applications, and the pseudo-static force and simulation computations were clearly *not* quick enough (cycle times for the force computation algorithm were several seconds). Further research must be done before gravity and contact force computation can be used successfully in the object associations system. However, the collision detection algorithms could be used interactively as an association that prevents the user from dragging an object through holes or gaps through which the object could not realistically fit. The algorithms are sufficiently robust to handle a wide variety of contact situations, and with a bit of help do a good job of simulating frictionless interactions between objects.

5 Extensions and Future Work

5.1 Supporting Modification of Structure

WALKEDIT allows you to manipulate building contents and furniture. However, creating the building's *structure* (i.e. walls, floors, and ceilings) from scratch or even modifying existing structural elements is still an open problem for virtual environments. There are no truly interactive, virtual-environment-based design tools for the structure of buildings. Furthermore, the tools that do exist (primarily 2D CAD tools) often fail to "force" the user to make geometrically valid models: in AutoCAD, for example, you can easily create a model that looks reasonable in plan views or elevations but is totally useless as input to a computational geometry or Walkthrough environment. In the Berkeley WALKTHRU, the user must still lay out the walls of the building with a text editor in a UniGrafix file. Obviously, the next logical extension to WALKEDIT is to allow it to interactively add, move, and delete structural elements such as walls, floors, doorways, and windows. With this capability, WALKEDIT would become the first complete interactive virtual environment for creating architectural walkthrough models.

There are a number of problems associated with achieving this capability. They fall into two classes: user interface problems, which are the primary focus of WALKEDIT thus far, and algorithmic problems involving real-time visibility processing. The set of user interface problems is quite large; it is also fundamentally different from the problem of moving building contents, in that people often move furniture in real life, but they do not normally push walls and ceilings into new configurations. There is no "natural" way for a single person to quickly change a room's configuration from rectangular to octagonal, or to carve new doors and windows into the walls. This makes the interactive structure manipulation task much more of an open problem.

On top of the user interface problem, there is another fundamental difference between manipulating structure and manipulating contents in our WALKTHRU environment. The contents of the building generally do not affect the visibility processing at all in the WALKTHRU; they are not considered "major occluding surfaces." Thus, moving them consists of simply removing a pointer from the "contains" list of one cell and adding them to the "contains" list of another cell. However, when a wall moves, the very cell structure itself is altered. Moving a major occluding surface can propagate changes in cell boundaries all the way up to the root of the K-D tree, rendering almost all of the visibility preprocessing instantly useless, and changing reams of cell subdivision data. We need ways to dynamically change the cell structure and perform the new visibility processing on the fly. These are also open problems, and both our group and Seth Teller's group at MIT are working toward solutions.

5.2 Hierarchical Models

There are only two levels of objects in the WALKTHRU system: objects and occluders. Objects are rigid entities that may be manipulated as a single body by WALKEDIT, and occluders are wall surfaces that cannot be moved. In reality, many objects are best defined by multiple pieces that move in constrained ways with respect to each other. For example, desks or file cabinets have drawers that move in a one-dimensional space anchored to the base unit. Doors move in one degree of freedom with respect to their frames. We would like to be able to package up hierarchies of objects along with some form of "relative" object associations to define their motions. However, adding hierarchical models would involve large scale changes to the way the WALKTHRU represents objects; thus, the problem is beyond the scope of this masters work. One very promising route is to change the walkthrough from a UniGrafix basis to a more advanced object representation system such as Silicon Graphics' Open Inventor format. Such a shift will raise questions in both the basic WK cell partitioning and object instancing structure and in object associations' interaction methods, which will have to take into account ambiguity in selection and manipulation of subparts of hierarchical groups.

5.3 Virtual Environments for Simulation and Training

Up until a few years ago, the usefulness of virtual environments was primarily limited to visualization tasks. Recently, these technologies have begun to propagate into design tasks such as AutoCAD's 3D visualization of a proposed construction, or the many "home improvement" programs on the shelves of software stores (which do "still-frame" walkthroughs of small, 1 or 2 room additions to houses, remodeled kitchens, or the like). As interactive walkthrough technology improves and computers speed up, virtual environment systems such as the Berkeley WALKTHRU will be useful as simulation environments as well as design environments. In the future, we may see virtual environments being used to train firefighters in how to combat raging building fires without ever setting foot near a flame, or to determine how changes in lighting or airflow will affect the comfort levels of occupants of a building. These are but a few examples of ways a combination of simulation and virtual environment technology can be used, but many challenges in both visualization and interface techniques stand before us.

Providing suitable virtual environments for single-user or distributed multi-user simulations in a single 3D world will require research on a variety of frontiers. In addition to moving furniture around, there are other, more constrained motions that a user needs to carry out in order to set up the exact state of the simulation. Doors and windows need to be opened and closed, blinds may have to be lowered, curtains drawn, and drawers on file cabinets or dressers may have to be pulled open partially to set up initial conditions for the simulation. We need to find simple and efficient ways to create models that permit such constrained movements as well as interface paradigms to make use of these models in interactive virtual environments. Furthermore, the WALKTHRU provides potential for improving visualization of the results of simulations. Outputs of modern simulators are often devoid of graphics, focusing on tables of numbers or two-dimensional graphs. It is difficult to visualize conditions in a multi-room environment, quantitatively or qualitatively, from a set of a few graphs. It would be more natural and more instructive to set up and view simulations from within the virtual 3D environment itself. We need to develop tools to help build such virtual simulation environments for buildings. Once we have established a suitable framework for interactive simulations in 3D virtual building models, the technology can readily be extended to many environments such as mine shafts, factories, submarines, or air craft carriers. Any reasonably compartmentalized environment with relatively confined visibility from most observer positions can make use of our visibility preprocessing techniques and of our caching schemes to give real-time rendering performance on mid-range graphics workstations.

Since many of these virtual environment applications can naturally involve more than one person in a single environment, we also need to look into providing a distributed environment where multiple users located at their individual workstations can visualize and interact in real time with one and the same building and simulation environment. This involves extending the virtual simulation environment into a distributed and concurrent program in which the scenarios displayed on the various workstations are properly synchronized. Such an environment can be used in many contexts: to allow teams to develop and rehearse collaborative techniques, to help building designers to work collaboratively on new designs, to allow personnel training for the control and maintenance of a complicated environment such as submarines and aircraft, or for practicing firefighting techniques under a wide variety of simulated conditions, to name but a few examples. NPSNET, an interactive multi-user virtual environment which is already in place, provides a real-world demonstration of the feasibility of such environments [52].

6 Conclusion

The goal of the WALKEDIT project was to construct a placement editor for real-time interactive walk-through of large building databases. We wanted to design WALKEDIT to work with off-the-shelf input and display hardware, a goal requiring the use of a software framework to allow the user to perform unambiguous 3D manipulation with 2D devices.

WALKEDIT's manipulation techniques are based primarily on *object associations*, a framework that provides the flexibility to combine pseudo-physical properties with convenient teleological behavior in a mixture tailor-made for a particular application domain or a special set of tasks. We have found that such a mixture of the "magical" capabilities of geometric editing systems with some partial simulations of real, physical behavior makes a very attractive and easy-to-use editing system for 3D virtual environments. The combination of goal-oriented alignments, such as snap-dragging, with application specific physical behavior, such as gravity and solidity, reduce the degrees of freedom the user has to deal with explicitly while maintaining most of the convenience of a good geometrical drafting program.

We found it practical to separate into two types of procedures the mapping of 2D pointing to 3D motion and the enforcement of the desired object placement behavior. These procedures are clearly defined and easy to implement as small add-on functions in C. Geometric and database toolkits allow high-level coding and ease of modification. Our object associations normally cause little computational overhead to the WALKTHRU system. This is an important concern, since keeping the response time of the system fast and interactive is a crucial aspect of its usability and user-friendliness [22].

The result is a technique that makes object placement quick and accurate, works with "drag-and-drop" as well as "cut and paste" interaction techniques, can provide desirable local object behavior and an automated grouping facility, and greatly reduces the need for multiple editing modes in the user interface. The resulting environment is devoid of fancy widgets, sophisticated measuring bars, or multiple view windows. To the novice user it seem that not much is happening – objects simply follow the mouse to reasonable, realistic locations. Ideally, that is how it should be: any additional gimmick is an indication that the paradigm has not yet been pushed to its full potential. Some issues remain to be fully resolved, such as dealing with association loops, but our prototype demonstrates that this approach provides a simple, flexible, and practical approach to constructing easy-to-use 3D manipulation interfaces.

The object associations mechanism was combined with a set of more traditional tools: a knapsack mechanism with cut, copy, rename, and paste capabilities on objects and groups of objects, infinite undo and redo capability, the ability to import new object models from UniGrafix files, and a set of more standard 3D manipulator widgets. Together, this package of tools defines a complete suite of editing functions that makes populating an architectural model with detail objects a quick, easy, and intuitive task even for computer "novices."

In addition to the manipulation mechanisms, some physical simulation capabilities were explored. The first of these, a gravity simulation which simulates a "height" for the user above the ground, has been found to greatly enhance the WALKTHRU experience by not only allowing the user to look up and down as they walk, but also allowing them to take elevators and staircases in a natural way. Secondly, interactive collision detection was attempted on objects, and was found to be helpful in some cases when placing objects against each other. However, physics simulation needs to be explored and refined further before it becomes useful as a tool for interactive manipulation in our WALKEDIT environment.

The result of our efforts has resulted in a building contents editor that provides simple and intuitive tools for populating a building model. This is a major step in the development of interactive virtual-environment-based design tools. With some additional work on manipulation of structural elements, we believe this work can be extended to be the first true start-to-finish building model generation environment, as well as the basis for a new generation of simulation-based design and virtual environment CAD tools.

Figure 27: An example of a scene constructed in about five minutes using WALKEDIT. The scene contains over 40 objects, all properly aligned with each other and artfully arranged by hand.

Acknowledgements

This work was done under the guidance of Carlo Séquin, who has been an advisor, mentor, and friend for the last 3 years. His unique combination of enthusiasm for tackling new challenges and down-to-earth sensibility about what is important in life and research have been inspirational, and I consider myself extremely fortunate to be one of his students. I am grateful to the whole Walkthrough research group: alumni Seth Teller and Tom Funkhouser, and ongoing colleagues Maryann Simmons, Ajay Sreekanth, Randy Keller, Rick Lewis, and Laura Downs, none of whom have never hesitated to give me assistance, insight, and support. Special thanks to Laura, of course, for reasons entirely unrelated to academics, although she continues to be the primary mathematical resource for both me and the whole lab. Thanks to Mark Brunkhart, with whom I have commiserated on many topics, including masters theses and the whole Ben and Jerry thing.

More generally, thanks to the whole Berkeley Computer Science department, which is a great bunch of people. Special thanks to Bob Miller, Terry Lessard-Smith, and Kathryn Crabtree for helping me take care of those administrative details that nobody in their right mind would want to deal with.

No academic research gets very far without support. Thanks to the National Science Foundation, whose excellent three-year fellowship program gave me the freedom to pursue this work without the hassles endemic to graduate students who have to take on teaching or GSR positions to pay their room and board. I would also like to thank the National Project Center of NEC Corporation for providing support for both this and other research in computer graphics and virtual reality here at Berkeley.

Finally, ultimately, thanks to my family, especially my parents, Richard and Maria Bukowski, and my grandparents, Guy and Virginia Agostino and Richard and Helen Bukowski; they have been and remain an unending supply of support, inspiration, and, of course, grief, without whom my present level of success would never have been possible.

References

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

[2] Airey, John M., John H. Rohlf, and Frederick P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, 24, 2 (1990), 41-50.

[3] *AutoCAD Reference Manual*, Release 10, Autodesk Inc., 1990.

[4] Baraff, D. Fast Contact Force Computation for Nonpenetrating Rigid Bodies. *Proc. of SIGGRAPH '94* (Orlando, FL, Jul. 1994), pp. 23-34.

[5] Barlow, M. Of Mice and 3D Input Devices. *Computer-Aided Engineering* 12, 4 (Apr. 1993), pp. 54-56.

[6] Bechtel, Inc. *WALKTHRU: 3D Animation and Visualization System.* Promotional literature, 1991.

[7] Bier, E.A. Snap-Dragging in Three Dimensions. *Proc. of the 1990 Symposium on Interactive 3D Graphics* (Snowbird, UT, Mar. 1990), pp. 193-204.

[8] Borning, A. The Programming Aspects of Thinglab, a Constraint-Oriented Simulation Laboratory. *ACM Trans. on Programming Languages and Systems* 3, 4, pp. 353-387.

[9] Borning, A. Constraint Hierarchies and their Applications. 1991 IEEE CompCon Spring 1991 Digest of Papers. IEEE Computer Society Press: Los Alamos, 1991, pp. 376-381.

[10] Brooks, Jr., Frederick P. Walkthrough - A Dynamic Graphics System for Simulating Virtual Buildings. *Proceedings of the 1986 Workshop on Interactive 3D Graphics.*

[11] Brown, Thurman A. *Interactive Object Displacement in Building Walkthrough Models.* Master's Thesis, Computer Science Division (EECS), University of California, Berkeley 1992.

[12] Bukowski, Richard W. and Carlo H. Séquin. Object Associations: A Simple and Practical Approach to Virtual 3D Manipulation. *Proc. of the 1995 Symposium on Interactive 3D Graphics* (Monterey, CA, April 1995), pp. 131-138.

[13] Cameron, S. Collision Detection by Four-Dimensional Interference Testing. *Proc. of International Conference on Robotics and Automation* (1990) pp. 291-302.

[14] Chen, M., Mountford, S., and Sellen, A. A Study in Interactive 3D Rotation using 2D Control Devices. *Computer Graphics* 22, 4 (August 1988), pp. 91-97.

[15] Codella, C, Jalili, R., Koved, L., Lewis, J., Ling, D., Lipscomb, J., Rabenhorst, D., Wang, C., Norton, A., Sweeney, P., and Turk, G. Interactive Simulation in a Multi-Person Virtual World. *Proc. of the ACM Conference on Human Factors in Operating Systems – CHI '92* (Monterey, California, May 1992), pp. 329-334.

[16] Cohen, J., Lin, M., Manocha, D., and Ponamgi, M. I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments. *Proc. of the 1995 Symposium on Interactive 3D Graphics* (Monterey, CA, April 1995), pp. 189-196.

[17] Dingeldein, D. and Lux, G. Theseus++: A High Level User Interface Toolkit for Graphical Applications. *Computers and Graphics* 17, 2 (March/April 1993), pp. 147-154.

[18] Khorramabadi, Delnaz. *A Walk Through the Planned CS Building.* Master's Thesis, Computer Science Division (EECS), University of California, Berkeley 1991.

[19] Faigle, C., Fox, G., Furmanski, W., Niemieo, J., and Simoni, D. Integrating Virtual Environments with High Performance Computing. *Proc. of the 1993 IEEE Annual Virtual Reality International Symposium* (Seattle, Washington, 1993), pp. 62-68.

[20] Fiala, J. Generation of Smooth Trajectories Without Planning. NIST Internal Report IR4622, National Institute of Standards and Technology, Gaithersburg, MD, September 1988.

[21] Figueiredo, M., Bohm, K., and Teixeira, J. Advanced Interaction Techniques in Virtual Environments. *Computers and Graphics* 17, 6 (November/December 1993), pp. 655-661.

[22] Funkhouser, T.A. and Séquin, C.H. Adaptive Display Algorithm for Interactive Frame Rates during Visualization of Complex Virtual Environments. *Proc. of SIGGRAPH '93* (Anaheim, CA, Aug. 1993), pp. 247-254.

[23] Funkhouser, Thomas A. *Database and Display Algorithms for Interactive Visualization of Architectural Models.* Ph.D. Thesis, Computer Science Division(EECS), University of California, Berkeley 1993.

[24] Gleicher, M. Briar: A Constraint-Based Drawing Program. *Proc. of the ACM Conference on Human Factors in Computing Systems – CHI '92* (Monterey, CA, May 1992), pp. 661-662.

[25] Gray, M., de Bear, D., Foley, J., and Mullet, K. Coupling Application Design and User Interface Design. *Proc. of the ACM Conference on Human Factors in Operating Systems – CHI '92* (Monterey, California, May 1992), pp. 658-658.

[26] Greenberg, D. More Accurate Simulations at Faster Rates. *IEEE Computer Graphics and Applications* 11, 1 (January 1991), pp. 23-29.

[27] Hahn, J.K. Realistic Animation of Rigid Bodies. *Computer Graphics* 22, 4 (Aug. 1988), pp. 299-208.

[28] Helm, R., Huynh, T., Lassez, C., and Marriott, K. Linear Constraint Technology for Interactive Graphic Systems. *Proc. of Graphics Interface '92* (Vancouver, BC, Canada, May 1992), pp. 301-309.

[29] Houde, S. Iterative Design of an Interface for Easy 3D Direct Manipulation. *Proc. of the ACM Conference on Human Factors in Operating Systems – CHI '92* (Monterey, California, May 1992), pp. 135-141.

[30] Hubbard, P.M. Interactive Collision Detection. *Proc. of IEEE Symposium on Research Frontiers in Virtual Reality* (October 1993).

[31] Kurlander, D. and Feiner, S. Inferring Constraints from Multiple Snapshots. *ACM Transactions on Graphics* 12, 4 (October 1993), pp. 277-304.

[32] Lin, M.C. and Canny, J.F. A fast algorithm for incremental distance calculation. *International Conference on Robotics and Automation*, IEEE (May 1991), pp. 1008-1014.

[33] Lin, M.C. *Efficient Collision Detection for Animation and Robotics.* Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, December 1993.

[34] Mirtich, B. and Canny, J. Impulse-based Simulation of Rigid Bodies. *Proc. of the 1995 Symposium on Interactive 3D Graphics* (Monterey, CA, April 1995), pp. 181-188.

[35] Myers, B.A. Creating User Interfaces using Programming by Example, Visual Programming, and Constraints. *ACM Trans. on Programming Languages and Systems*, 12, 2 (Apr. 1990), pp. 143-177.

[36] Nelson, G. Juno, a Constraint-Based Graphics System. *Proc. of SIGGRAPH '85* (San Fransisco, CA, Jul. 22-26, 1985). In *Computer Graphics* 19, 3 (Jul. 1985), pp. 235-243.

[37] Nielson, G. and Olsen, D. Direct Manipulation Techniques for 3D Objects Using 2D Locator Devices. *Proc. of the 1986 Workshop on Interactive 3-D Graphics* (Chapel Hill, NC, Oct. 1986), pp. 175-182.

[38] Shaw, C., Liang, J., Green, M., and Sun, Y. The Decoupled Simulation Model for Virtual Reality Systems. *Proc. of the ACM Conference on Human Factors in Operating Systems – CHI '92* (Monterey, California, May 1992), pp. 321-328.

[39] Shoemake, K. ARCBALL: A User Interface for Specifying Three-Dimensional Orientation Using a Mouse. *Proc. of Graphics Interface '92* (Vancouver, BC, Canada, May 1992), pp. 151-156.

[40] Smith, R. Experiences with Alternate Reality Kit: An Example of the Tension Between Literalism and Magic. *IEEE Computer Graphics and Applications* 7, 9 (September 1987), pp. 42-50.

[41] Snibbe, S., Herndon, K., Robbins, D., Conner, D., and Van Dam, A. Using Deformations to Explore 3D Widget Design. *Proc. of SIGGRAPH '92* (Chicago, Illinois, July 26-31, 1992). In *Computer Graphics* 26, 2 (July 1992), ACM SIGGRAPH, New York, 1992, pp. 351-352.

[42] Strauss, P., and Carey, R. An Object-Oriented 3D Graphics Toolkit. *Proc. of SIGGRAPH '92* (Chicago, Illinois, July 26-31, 1992). In *Computer Graphics* 26, 2 (July 1992), ACM SIGGRAPH, New York, 1992, pp. 341-349.

[43] Teller, S.J., and Séquin, C.H. Visibility Preprocessing for Interactive Walkthroughs. *Proc. of SIGGRAPH '91* (Las Vegas, Nevada, Jul. 28-Aug. 2, 1991). In *Computer Graphics*, 25, 4 (Jul. 1991), pp. 61-69.

[44] Teller, Seth J. *Visibility Computations in Densely Occluded Polyhedral Environments.* Ph.D. Thesis, Computer Science Division (EECS), University of California, Berkeley, 1992.

[45] Van Emmerik, M. Direct Manipulation Technique for Specifying 3D Object Transformations with a 2D Input Device. *Computer Graphics Forum* 9, 4 (December 1990), pp. 355-361.

[46] Van Emmerik, M. Interactive Design of 3D Models with Geometric Constraints. *Visual Computer* 7, 5 (September 1991), pp. 309-325.

[47] Venolia, D. Facile 3D Direct Manipulation. *Proc. of the ACM Conference on Human Factors in Computing Systems – CHI 93* (Amsterdam, Netherlands, Apr. 1993), pp. 31-36.

[48] *Virtus Walkthrough.* Promotional literature, 1991.

[49] Ware, C. and Balakrishnan, R. Reaching for Objects in VR Displays: Lag and Frame Rate. *ACM Transactions on Computer-Human Interaction* 1, 4 (December 1994), pp. 331-356.

[50] Yoshimura, T., Nakamura, Y., and Sugiura, M. 3D Direct Manipulation Interface: Development of the Zashiki-Warashi System. *Computers and Graphics* 18, 2 (March/April 1994), pp. 201-207.

[51] Zeleznik, R., Herndon, K., Robbins, D., Huang, N., Meyer, T., Parker, N., and Hughes, J. An Interactive 3D Toolkit for Constructing 3D Widgets. *Proc. of SIGGRAPH '93* (Anaheim, CA, Aug. 1993), pp. 81-84.

[52] Zyda, Michael J., David R. Pratt, James G. Monahan, and Kalin P. Wilson. NPSNET: Constructing a 3D virtual world. *ACM SIGGRAPH Special Issue on 1992 Symposium on Interactive 3D Graphics*, March, 1992.