Interactive Techniques in Three-dimensional Modeling

Michael Wimmer, Robert F. Tobler
Institute of Computergraphics
Vienna University of Technology

Abstract

This paper describes the methods used in writing a 3D modeler for creating furniture that can be arranged in a room viewer. The main emphasis lies on an intuitive user interface, and various ideas have been exploited to provide that, including direct manipulation, collision detection, automatic measuring techniques, virtual design tools, hierarchical grouping and material management combined with focussing and mixed scaling functions, to name just a few.

Keywords: 3D modeling, direct manipulation, interaction, measuring, collision detection, virtual design tools

1 Introduction

Most of today's 3D modeling systems are not very easy to cope with from a user's point of view. Many of them employ command-line interfaces, others have simple iconic interfaces that are not very intuitive. One of the primary design goals for a modeling system should be the ease of use of the program. No tool, however intricate or powerful, is of much value to the user if he is not able to exploit all of its capabilities because of a badly designed user interface.

As 3D hardware is becoming ever more powerful, new interface metaphors for 3D manipulation can be explored that have not been available before and therefore cannot be found in many existing modeling applications.

To evaluate various such methods, we have implemented a system which we believe could benefit many users: an application environment for interactively designing furniture and placing it in a virtual room. This paper is about a program for the interactive modeling of furniture. The program to actually place the created furniture in a room is thoroughly described in [Eckstein96].

In the rest of this chapter, we will give a short introduction about the principles of interaction in 3D modeling; the second chapter describes the requirements that have to be met and the environment in which the application has to work, and the fourth chapter will describe how these requirements can be met and what ideas went into the design of the modeling application that has been created.

1.1 3D Modeling - a Definition

Every method of mimicking a real world object or some of its aspects with some artificial means or tool can be called modeling. Doing this with a computer gives the advantage of flexibility of representation methods. Of particular interest are geometric modelers that concentrate on the geometric aspects of a model, and ever more importantly, 3D-modelers, which allow visualization of three-dimensional geometric data, which most of the times represents solid objects. In a modeling system, normally the program which allows the user to specify the 3D geometry of the model is called the modeler. For a complete treatise on the methods of data representation and rendering for 3D solid modeling, see [Requicha92].

1.2 Direct Manipulation 2D/3D

Direct Manipulation is a technique of making user interfaces more intuitive by representing the objects of interest visually and letting the user manipulate them directly with an input device like a mouse ([Shneiderman83]).

But beforehand, a few words on current interfaces: Older systems use pure text interfaces, where the user has to enter a programmatic description (for example in a script language), and the computer renders the scene afterwards. This can be combined with more feedback so that the result of each command is displayed immediately. At the moment, menu-style applications with dialogue boxes and simple picking of objects are very popular. Every object can be selected by clicking on it (called picking), then this object can be manipulated by selecting a command from a menu and/or entering appropriate values in a dialog box.

There is also some experimentation with real 3D input devices like bats or gloves. Yet these are not widely accepted because of the cost of the equipment and the additional difficulties that have to be overcome: for example, such devices tend to preclude the use of traditional input devices. Besides, it is not quite clear how to map the additional degrees of freedom gained by such a device to actual movement. Attempts to overcome such problems are to be found in [Yoshimura95], where a Polhemus device has been used as a 3D pointing stick combined with feedback on a normal workstation screen, and in [Liang95], who present various user interface metaphors, like menus arranged in a circle, to deal with the mapping problem.

But back to Direct Manipulation: originally, this method was used to simplify interaction with operating systems and 2D drawing and painting programs. The idea can also be applied to 3D modelers, although with some considerations.

In 2D, creating primitives can be simplified by using rubberbanding [Foley90], i.e., dragging an object to its desired size upon creation. This cannot be extended easily to 3D, as a mouse can only designate 2 dimensions. The alternative is letting the user create objects in default sizes, which can then be modified afterwards.

Selecting and grouping can be much more readily used in 3D: casting a ray through the pixel where the user clicks with the mouse and selecting the first intersected object is a very efficient and intuitive method of selecting objects (also called picking). Alternatives, like a three-dimensional cursor that has to be positioned exactly on the desired object [Databecker95], have proven impracticable. Grouping objects together to treat them as a unit for future operations is also a concept that lends itself very well to 3D modelers.

Much thought has been given to interactively specifying directions for positioning tasks like translation, scaling or rotation in 3D (especially rotation has received thorough treatment in [Chen88], showing a virtual trackball-like approach to be quite advantageous). [Emmerik90] promotes local coordinate systems with handles that can be dragged to achieve a particular movement [Nielson86] projects the coordinate axes into 2D and calculates areas of influence for each axis to define in which direction a particular mouse movement should go. [Bier86] shows an extended version of a 3D cursor (skitter) which can be positioned with a 2D device (jack).

The most promising approach seems to be so-called draggers (described in [Strauss92], and used for various information visualization purposes in [Grimm95]) respectively manipulators (figure 1), as used in Open Inventor, a high level 3D programming API ([Wernecke94]). Draggers are simple geometric objects that surround the object to be manipulated. By their look, they tell the user how to click on it to manipulate the object. A virtual trackball for example lets the user know that he can rotate the object surrounded by it.

1.3 Constraints

To further enhance a direct manipulation interface, constraints can be used. A constraint defines a relationship between objects or parts of objects. Especially, user-definable constraints that exist for one period of interaction are very easy to implement.

Constraints can be specified by textual input, graphically, or automatically, which requires substantial computing power and is not easy to implement because of ambiguities. A scripting language can be used, as demonstrated in [Rossignac86], another possibility is using direct manipulation techniques [Sistare91]. Resolving constraints can be done incrementally in some specified order (for example in the order in which they are defined, [Emmerik91]) or by satisfying all constraints at the same time, which requires solving a very big equation system and can only be achieved with numerical methods. A probabilistic approach to the problem, evaluating the most likely constraints, can be found in [Fa93].

1.4 Snap Dragging

Many applications can profit from snap dragging ([Bier90]), originally a two-dimensional technique employed to make it easier to align objects relative to each other. For snap dragging to work, there have to be areas exhibiting a certain kind of gravity, and areas that can be attracted by this gravity. Moving the two areas near each other makes them snap together. If applied correctly, snap dragging can greatly enhance the modeling task.

1.5 Case Studies

It is interesting to compare some of the more widely used modeling applications, giving particular emphasis on the quality of their user interface.

AutoCad, for example, is one of the most widely used tools to create precise models of almost anything. It is a very flexible tool and provides menu selection combined with simple picking as the main interface type, often coupled with entering textual commands (depending on the version). Direct Manipulation is not really used, just as rendering is based on lines and parallel projection. To get a shaded representation, an additional program has to be called. AutoCad provides many features and is quite well suited to the professional needs, also because it is extensible for many application areas, but the user interface needs getting used to it and cannot be called Direct Manipulation.

The program *3D Studio MAX* is in many respects similar to AutoCad, although it does allow for some direct manipulation effects. Objects are viewed simultaneously from front, top and side, additionally there is a perspective view. Depending on which of the four windows is used for manipulation, objects or parts of objects can be dragged in particular directions. Even snap dragging can be used upon object creation, as each of the windows is in itself a two-dimensional view of the three-dimensional object.

As the program we wanted to write was directed at designing furniture, one particular program only available in Germany caught my attention: the *Data Becker Wohndesigner* [Databecker95]. It is an application that lets the user arrange predefined furniture in an apartment of adjustable measures. Although the program boasts ease of use and we found the rendering speed and quality to be excellent, we were astonished at the approach the programmers had taken to manipulating (i.e., moving) furniture in a room. The user has to control a virtual 3D mouse, which can be moved in two directions normally with the mouse, and the third direction can be changed by pressing one mouse button in addition to moving the mouse. With this device it was virtually impossible to select an object, let alone manipulate it.

2 Problem Evaluation

In this section, we will briefly outline the requirements for the project and what led to its existence.

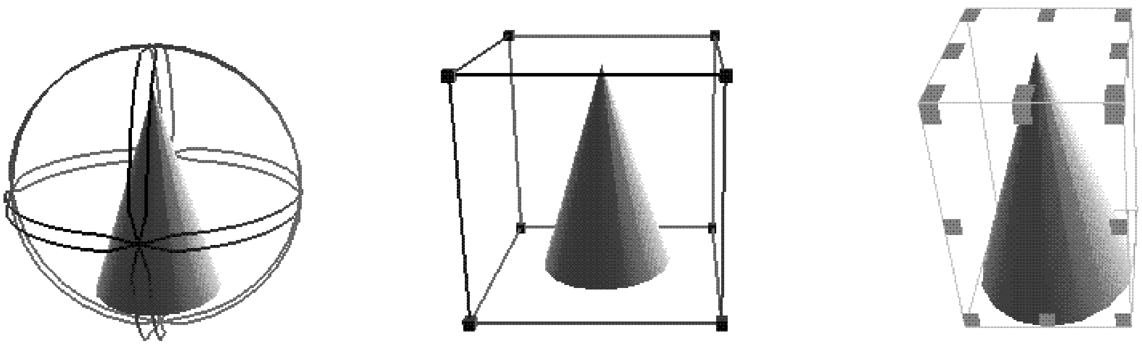

Figure 1 Three manipulators:
Left: a trackball manipulator for object rotation
Center: a transformbox manipulator for translation, rotation and uniform scaling
Right: a handlebox manipulator for translation and non-uniform scaling

2.1 Basic Requirements

The most basic and at the same time most important requirement for the project was ease of use of the program. No one would bother with a program that was difficult to use, as there exist many of this kind, most of them probably much more powerful.

So much effort was directed at making the user interface intuitive and easy to use. This means that the system should have a *steep learning curve*, should be *efficient* to use, should prevent committing errors in its use, *easy to remember* and have an *appealing look*.

One of the important design goals to consider is *consistency*. If a system is not consistent, i.e., exhibits similar behavior in similar situations, it will be confusing for the user.

Another is *feedback*: if the user can watch the effects of his actions directly on screen, he will be able to get a better understanding of the user interface and its possibilities. Feedback can be used to visualize modes, show additional information and inform the user of the outcome of an action.

Most users are prone to committing errors while using a program, so the program should try to *prevent* as many *errors* as possible (like selecting a menu item that makes no sense in the current context). In addition, the user has to be given the possibility to *recover* from *errors* to be able to experiment a little. Otherwise, a simple keystroke could ruin a day's work.

2.2 Requirements specific to 3D modeling

Whereas the previous requirements can be applied to most interactive applications, the following is specific to three-dimensional modelers and to the task at hand in particular.

The program should allow the *instantiation* of primitives such as cubes and spheres, as well as their easy *selection* through picking. The objects should also be *groupable*, and one should be able to order such groups hierarchically. Furthermore, there should be a means to edit objects nested in that hierarchy without breaking up the hierarchy, i.e., a kind of *focussing*.

One aim of the application is to make the objects behave as much as possible as they would in the real world. Specifically, they should not be allowed to overlap each other. This can be achieved with some kind of *collision detection*, which can also be used as a more complex design aid, as it facilitates aligning objects along each other.

A feature that must not miss in a furniture design program is of course the ability to *measure lengths*. This is not easy to accomplish in an intuitive manner; an automatic method would of course be desirable.

Some requirements stem from the fact that the furniture designed with this program should be used and manipulated in the viewing program. One such requirement is the ability to create objects that can *scale parts* of it differently. For example, if you scale a table, you would expect the top of the table to scale accordingly, but not the feet of the table. They should only change their position. Such objects have to be taken into account.

Another is the ability to define *constraints* for the viewer. An example would be to designate the backside of a cupboard as such, so that the viewer knows that the cupboard should only be allowed to move along walls.

Some companies have furniture that can be freely configured from many parts, for example shelves. The user should be able to define pieces of furniture as belonging to the same system of shelves.

Finally, there are some properties of a piece of furniture that are not directly related to computer graphics, but of great interest to both the company and its customers. Examples for these are accurate measurements, price, availability and a bitmap of the real object, which should be included in the 'electronic catalogue'.

3 The Object Designer - Principles and Implementation

As an introductory note to this chapter, the development platform should be described: the program is based on the Open Inventor API, a high level graphics programming API that can be used under Windows 95 or Windows NT in conjunction with Visual C++ to write interactive graphic applications. Inventor already provides a basic selection mechanism and also manipulators and draggers, which had to be substantially modified to be suitable for the application, though. Open Inventor itself is based on OpenGL and thus widely supported.

3.1 User Interface

We decided to use the Microsoft Foundation Classes for dialogue boxes and menus, because experience shows that users either are already accustomed to this interface from various other programs, or they are quick to learn it. To further facilitate the necessary 2D interaction, menus have been appropriately named, and menu depth has been kept low (maximum two sub-menus). Only those commands that cannot be interactively specified with the mouse are available from the menu. Choices that are not applicable at the moment are grayed out to prevent user errors.

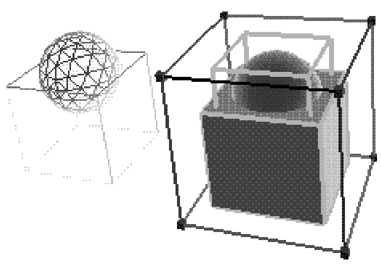

Figure 2 Two kinds of feedback:
1. The left model does not belong to the currently selected (focussed) group, thus it is drawn in wireframe.
2. The right object consists of two parts: a sphere and a box, which are both currently selected. The selection is shown by a highlighting box drawn around each of the two parts.

All typical three-dimensional interaction tasks can be carried out directly with the mouse, the direct manipulation approach that was selected will be described separately. Selection of objects is done by raycasting at the clicked position as described before. The selected object will be surrounded by a manipulator which shows where to click on it to obtain a particular movement. At the moment, different manipulators can be selected for different interaction tasks, but it is planned to merge all of them into one or two manipulators that give easy access to the most important tasks.

3.2 Model Display

We decided to implement only one perspective view of the current modeling environment instead of three ortographic projections (ie side views), as is done in modeling packages like 3D Studio MAX. As the single view can be displayed full screen, it gives a better representation of the scene (which is further improved by the fact that it always shows a shaded representation). This also makes 3D feedback easier to discern and is not as confusing for the user: with three separate views, the user always has to remember which degrees of freedom he can manipulate in each particular view. With our approach, all degrees of freedom are mapped to manipulators, which provide instant visual feedback about all allowed degrees of freedom (figure 1) and can be controlled in a direct manipulation manner.

3.3 Feedback

The program makes extensive use of feedback: clicking on one side of a manipulator that allows movement in a the plane where this side lies causes two arrows to appear immediately which show the user in which direction his movement will go. If you select more than one object, all individual objects are surrounded by red boxes. If you group them, only one red box is drawn to indicate that these objects now belong together (figure 2).

Changes in material, which can be effected in a material editor, are immediately displayed in all connected objects. There are even more examples of feedback in the program that will be discussed under the appropriate headers. Together, those measures make sure the user is confident that nothing is happening „behind the scenes."

3.4 Undoing

An important feature of the program is the undo-function. This allows the user to experiment without fear of losing a day's work or having to save his work continuously. It can be expressed even more accurately: we found our program only to be usable from the point we implemented an undo-function, which proved to be a major step in the maturity of the program.

Undoing was implemented via a multi-level undo function which allows a number of undo levels only restricted by the amount of available memory. This was important, as one conceptual user-interaction often consists of more than one actual interaction step. The number of levels can be restricted to any number, though, in which case undo slots are reused in a round-robin fashion. As long as no other action has been carried out, each undo itself can be undone, resulting in a redo. This can be repeated until the original state has been restored. Redoing is important especially in multi-level undo-systems, because you can easily „overshoot", undoing one operation too much. Redo provides a convenient way to navigate to the exact desired spot in the undo-history.

The scope of the undo function has been chosen to coincide with an integral mouse movement. Button down starts an action, button up concludes it. Tracking every individual mouse movement would have been much too costly and is not necessary.

To keep the implementation effort for this function as simple as possible, the undo function is implemented as a list of complete scene graphs which are stored after each action. Issuing an undo replaces the current scene graph with one of the graphs in the undo list. Storing the scene graph has been mostly implemented by way of callback functions from the concerned scene graph objects. Testing has revealed the performance of this solution to be adequate for the task.

3.5 Focussing

A grouping mechanism similar to the one found in popular 2D drawing applications has been implemented, but with some useful extensions. To select more than one object, the user simply holds the shift-key while pressing the mouse button on different objects. Even now the objects can be moved together, but to achieve a more permanent alliance, they have to be grouped through a menu command. Once grouped, all commands apply to the whole group: geometric manipulations as well as changes in material, although objects with a material different from the rest of the group retain this material even if the group material is changed. The extension of this 2D-mechanism to 3D was very important, because it is a necessary feature of any modeling system on the one hand, and the actual way it is implemented resembles the 2D-mechanism which users already know so closely that almost no additional learning effort is necessary.

As hierarchies are especially important in 3D, it is also possible to nest groups inside each other. But is important that the user can edit objects individually although they already belong to a group. To overcome the limitation found in most 2D or 3D design packages that forces users to split up groups for editing its mem-

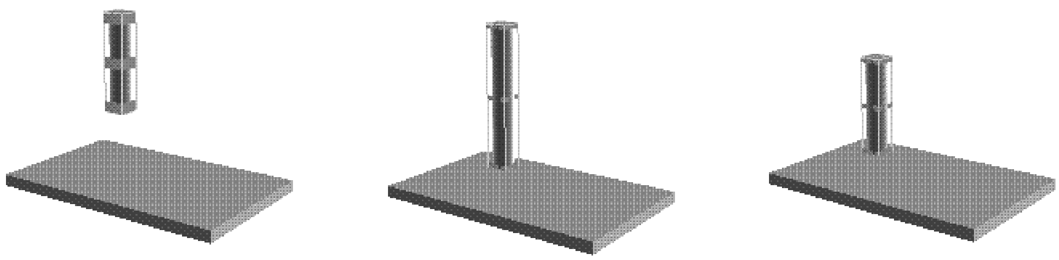

Figure 3 Examples of collision treatment:
Left: a table foot floating over the table board
Center: with only one mouseclick you can tell the table foot to move exactly up to the table board
Right: with only one mouseclick you can tell the table foot to scale up to the table board instead of moving

bers, we have introduced a mechanism best described as 'focussing'. By focussing on a particular group, all further actions only have effect on the members of that group, but not on any object outside of that group. To indicate that, all objects outside are drawn in wireframe style (figure 2) and are not even selectable. With 'zoom in' and 'zoom out', the user can descend and ascend in the group hierarchy and edit individual objects belonging to any level of the group hierarchy.

Yet focussing poses some problems when dealing with materials and transformations in the scene graph. In the following paragraphs, we will describe the method we used to ensure consistency among group properties and individual object properties.

It is important to note that a group itself has the same properties as a simple object in that it has its own local transformation matrix and material. For transformations (scaling, rotation, translation), this has the following impact: Transforming the group changes the local transformation of the group. Transforming its members (by focussing on the group) changes the local transformations of the members. Joining objects to a group assigns an identity transformation to the newly built group; all objects retain their original matrices. Splitting a group causes the local transformation of the group to be prepended to the local transformation of each group member, so that the actual positions of all objects remain the same.

Materials must be treated a little differently, as they are not cumulative, unlike transformations. Group members have to be divided into those that share the common material of the group and those that have their own material. If the material of the group is changed, all the members that share this material also change, but not the members that have heir own material. As soon as you focus on a group and change the material of a member, this material becomes distinct from the group's material and will not change further with the group. Splitting up a group will assign the material of the group to all members that do not yet have their own material. The most difficult case is the creation of a group: If all objects that are designated to become the group have the same material, this material will be chosen to become the material of the group and all those objects will lose their material. But if even one single object has a material different from the others, all designated members must retain their material and no particular material is chosen for the group.

Admittedly, this kind of material management sounds a little bit complicated, but is required to maintain some flexibility in the matter. Some additional menu commands are required to achieve particular tasks: One very important command allows the user to revert a group member's material to the material of the whole group. This is useful if the user accidentally or for experimentation changes the material of one or more members of a group, but then decides that they should again share the group's material so it can change with the group. Whereas this command operates on single group members, a second command can come in handy if one wants to remove all individual materials in all nested groups of a particular group. This ascertains that changing the group material will change the material of all the contained objects, regardless of how deeply they are nested.

Materials can be changed through a non-modal dialogue box that has sliders for diffuse, ambient and other components of the light. An additional color editor lets the user select the color of each component. A future extension will allow the user to select a material from a predefined list of named materials, which may also contain textures. A preview of the material will of course be given. Another extension will add a checkbox to the material editor that will give feedback whether the currently selected object has an individual material assigned or whether it shares the material of the group. This checkbox will help to better understand the working of the material management and provide additional valuable visual feedback.

3.6 Collision Detection and Sliding

Collision Detection has proven to be one of the most valuable design aids in modeling. This is because it makes it easy to arrange objects so that they just touch each other, which is what is desired most of the time.

Yet it is not sufficient to simply detect collisions. They have to be *treated* to yield acceptable results. Let's have a look at a simple example: if you want to place a table near a wall so that it just touches the wall, you would drag the table right into the wall. Depending on the speed of the mouse movement, the program will probably see two subsequent positions of the table: one a little bit before the table hits the wall and one a little bit after that. Most certainly there will be no position where the table is aligned exactly to the wall. An application only doing simple collision detection would have two choices: either stopping the table in front of the wall, which would not satisfy the need that the table has to touch the wall, or letting the table move right into the wall, which doesn't make very much sense, as solid objects cannot intersect in real life (this kind of collision detection is used in [Databecker95], by the way).

We try to treat collisions correctly. For this to work, we had to distinguish between translation, scaling and rotation (figure 3).

The most common cause for collisions will be translations. The desired result of collision treatment would be that the moving object is translated as much as possible without causing a collision. For that to happen, the program calculates the moving vector

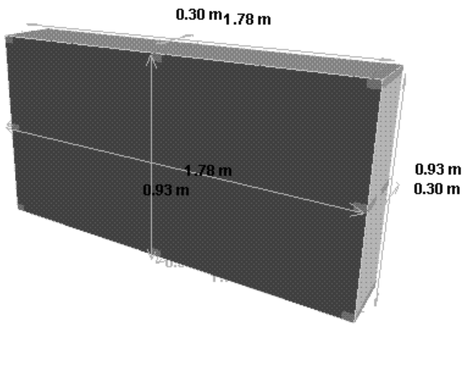

Figure 4 Scaling of a box: all measurements are displayed immediately on clicking and updated frequently.

and intersects it with the collision objects to see how much movement in the desired direction is possible. With this extension, it is possible to align objects against each other. But there is another useful extension. Imagine, for example, a table that should be moved along a wall, always just touching the wall. This would be very tedious with the scheme just described, because every mouse movement that went a little in the direction of the wall would cause a collision, preventing the table from moving at all. A better possibility is to calculate not only the point of collision, but also that part of the motion vector which can still be carried out without causing a collision. This can easily be obtained by intersecting the motion vector with the plane the colliding polygon lies in. This process can also be called *sliding*.

Real collision treatment for rotations is a little bit more difficult and not necessary most of the time, so it has not been implemented yet. Instead, a function that lets objects rotate so that they have the same orientation as some other nearby object is planned. Simple collision detection for rotations works, though.

The third application of collision detection is scaling. Here again it is important to obtain the exact position where collision took place so that an object can be sized to exactly meet another object. This has been implemented with a binary searching algorithm that converges very fast to an appropriately accurate solution. [Eckstein96] describes an efficient method for speeding up all of the above mentioned cases via bounding boxes and k-d trees.

3.7 Measuring Object Dimensions

Real life objects have real life dimensions. To accurately represent those objects in a computer model, the computer has to know these dimensions so he can display them in their correct relative proportions. One possibility to create objects in a desired size would be via a dialog box interface.

But the program we wrote strives to do without such mechanisms as much as possible, thus an automatic measuring mechanism for object dimensions is provided, which works like this: as soon as an interaction can possibly cause a scale, the program displays two perpendicular double arrows on each side of the bounding box surrounding the object, with each arrow carrying the measures of its length. When the object is scaled, the measurements are updated continuously so that the object can be easily resized to the desired dimension. Thus, automatic feedback about object dimensions is readily available whenever the user wants to resize objects (figure 4). We found that this function facilitates the task of giving an object particular dimension. For some tasks, it should be accompanied by a traditional dialogue box, though.

3.8 Measuring Distances - Virtual Rulers

The previous section dealt with how to allow measuring object dimensions. Yet another important measurement is the distance between objects. This cannot be automatically measured, as there would be too many possible distances to measure at any one time. One possible solution would be a mechanism allowing you to click at two points to designate a line whose distance should be measured, but it is not very intuitive and also error-prone.

A very new idea is to use the same tools for measuring that we would use in the real world. The concept of a ruler, for example, can readily be extended to a virtual tool. For the application, we are going to create a virtual ruler that will facilitate measuring distances between objects (figure 5).

For such an idea to work, the virtual ruler must be easy to move around and rotate by direct manipulation. Furthermore its length must be easily adjustable by dragging a handle on the ruler itself. Of course, this idea only works because of collision detection, or rather treatment. If you want to place two blocks in a certain distance from each other, you position one of the blocks, then adjust the distance on the ruler and move the ruler so that it just touches the block (via collision detection), and then move the other block so that it just touches the ruler, also via collision detection.

Through a simple extension, the ruler can have block-sized endpieces, which allow it to be used to measure distances between edges or between edges and blocks (see the illustration for better understanding). These endpieces can of course also be switched on and off directly on the ruler by way of direct manipulation.

3.9 Mixed Scaling

Many furniture companies let the customer choose between different dimensions for some objects. A table, for example, can be bought in various lengths and widths, all other parameters staying the same. This could easily be implemented by scaling the object to the desired size. But there is a hidden problem: if a table is scaled to a different size, you do not necessarily want the table feet to scale with the rest of the table. Instead, they should only change

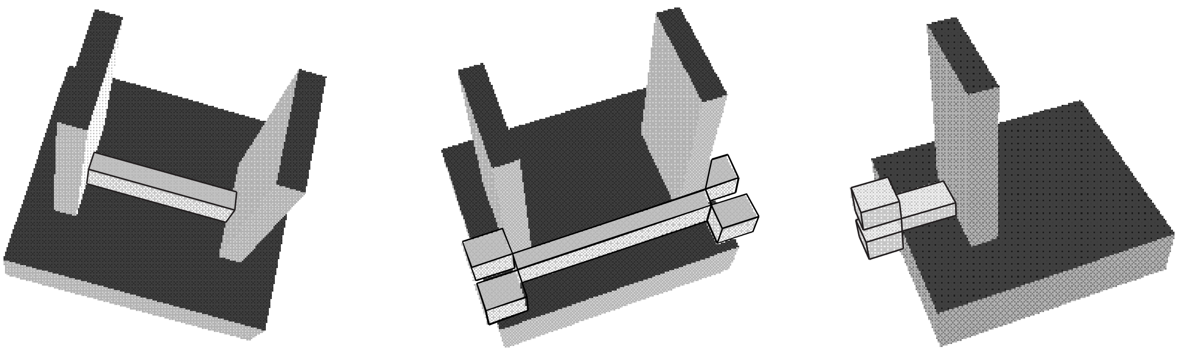

Figure 5 Examples for the application of a virtual ruler (prototype)
Left: without endpieces, the ruler measures distance betweent the two blocks.
Center: with both endpieces, the ruler can measure the distance around the blocks
Right: with one of the endpieces, you can easily measure the distance from the block to the boarder of the board.

their positions relative to each other. Thus, the furniture designer has to specify which parts of an object that should be allowed to scale and which parts should only change their position.

First of all, the objects that should participate in such a mixed scaling function have to be grouped together. The group defines the interface one would see in the object placement application (which will simply be called viewer from now on). Thus, the user of the furniture designing program has to specify in which directions the group should be scaleable. He can also specify upper and lower bounds for the dimensions of the group and also whether the object should be freely scaleable or only to specific positions. The user interface for this function is a combination of direct manipulation and non-modal dialogue boxes. For each direction, an arrow is shown on the object, which can be toggled between an active and an inactive state by clicking on it. In the dialogue box, one can specify upper and lower bounds and values between, or even increments for the direction whose arrow was last clicked on.

Having specified the scaling properties of the whole group, the user now has to specify how each member should react to a scaling of the group. For this to be understandable, one has to imagine each member of the group to be surrounded by a bounding box that is embedded in one big box encompassing all other boxes which represents the group itself. Let's look at one member: Each side of its box is assigned a value that defines the relative influence of a movement of the side of the outer box that has the same orientation. If this value is one, then this side of the box moves exactly the same amount as its surrounding counterpart, if it is zero, it is totally unaffected by the side of the outer box with the same orientation, but moves with the side of the outer box that has exactly the opposite orientation.

Take a table, for example (figure 6): Each side of the table top would be assigned a one, so it can be scaled according to the surrounding box. Also the top side of the table top is assigned a one, in case the table is to be made taller. But the lower side of the table top must be assigned a zero, because it should not move with the lower side of the surrounding box, but with the upper side. Thus, if the table is to be made taller, both the upper and the lower side of the table top move with the upper side of the surrounding box, and the thickness of the table top remains unchanged. It is quite similar for the table feet: the upper and lower side of the table feet are assigned a one because they should scale exactly with the surrounding box. Also the outer sides get a one, but the inner sides should also move with the outer sides of the surrounding box, so they are assigned a zero.

The user interface for this function will probably be similar to the interface for the surrounding box, i.e., the group. There will be a non-modal dialogue box for specifying values, and arrows on the object itself that can be clicked to designate which value to change.

3.10 Constraints as Construction Aids

Many construction tasks can be vastly simplified if constraints can be applied. Due to the complicated nature of constraints, the furniture design program contains only a small number of constraints which are all activated at user request (for example by pressing an additional key) for the duration of a mouse movement. Creating further constraints would be quite difficult and thus be against the original intent of the program, intuitivity, as there are simply very few assumptions you can make when designing a piece of furniture from scratch.

Two major constraints have been implemented: One restricts movement that was originally designed to be possible in a plane to a line and is available by pressing the Shift-key. Another is always active and can be deactivated by a menu command - it restricts rotations to multiples of 90 degrees. The assumption that most pieces of furniture will be rectangular is one of the few assumptions that mostly hold true. At a later stage, an extension will allow restricting orientations of an object to the orientation of nearby objects.

3.11 Defining Constraints for the Viewer

As was shown with the mixed scaling functions, some of the properties defined in the furniture designing program are solely to be used in the viewer. Another example are specific constraints that can be defined in the design program and used in the viewer.

The constraints used in the viewer mostly tell the viewer that a specific object moves with one side against the wall, like a cupboard, or with one side against the floor, like most objects, or with one side against the ceiling, like a lamp for example. To

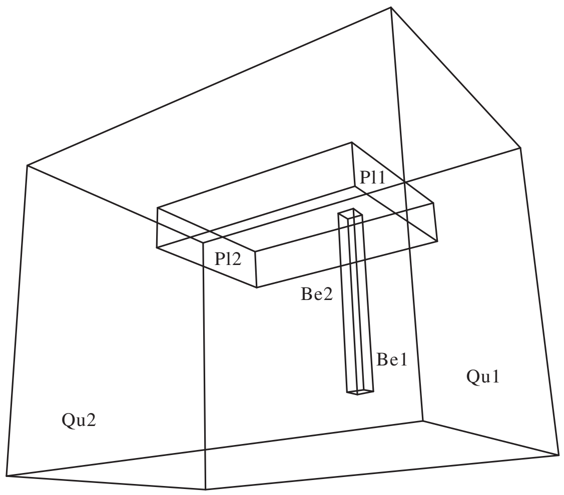

Figure 6 Mixed scaling for a table board and foot: Be1 and Be2 are right and left sides of the table foot, Pl1 and Pl2 are right and left sides of the table board, Qu1 and Qu2 are right and left sides of the enclosing box.
1. Pl1 und Pl2 get a 1, ie Pl1 always moves with Qu1 and Pl2 always moves with Qu2.
2. It is different for the table foot: Be1 gets a 1, ie moves with Qu1, whereas Be2 gets a 0 so as to move with Qu1, too, not with the opposite sode Qu2!

define such constraints, an additional input mode has to be entered in the program in which all objects are shown with a bounding box around them. A non-modal dialogue box shows the constraints of current (i.e., last clicked on) side of the current bounding box. In the dialogue box, the user can choose whether this side should move against a wall, the floor, the ceiling or should be freely placeable.

Of course, such properties can be specified for arbitrary nested groups, so that not necessarily the bounding box for the whole object has to be used for constraint creation, but most times this will suffice.

Another similar constraint that can be defined in the furniture design program is collision detection for the viewer. If objects become very complex, computing time for collision detection can become very time consuming if the automatic collision detection mechanism is used which creates collision polygons for each and every primitive. For most objects, it would suffice to use its bounding box, or at least the bounding boxes of nested groups, for collision detection. To achieve that, the user has the possibility to designate each group as a collision bounding box for all embedded objects. In the viewer, the embedded objects will then not be considered when doing collision detection, only the bounding box. Of course, bounding boxes could be generated automatically, but they could only be used for simple rejection tests. Bounding boxes specified interactively actually replace the object with its bounding box for the collision treatment algorithm, so intersections with its faces will never have to be done.

3.12 Complex Furniture Designs

Some furniture companies offer designs in which the customer can freely configure the parts he wants and arrange them to his liking. Shelf systems are an example for this: the customer can add shelves in different sizes, side parts and corner parts.

The viewer should provide a mode in which such systems can be arranged; but more importantly, the furniture design program should provide a mode in which the user can define how parts of such a system may be connected, so that the customer can only arrange the parts in valid configurations in the viewer. This makes it very easy for the user, as parts snap together as soon as they are in a position that is near a valid configuration.

Snapping polygons have to be created for this to work, and an easy way of implementing this would be to let the user 'show' the program how the parts fit together. Whenever the user issues a command that specifies a valid configuration, snapping polygons are created on the intersecting polygons.

4 Summary

We have described how various methods can be used to create a more intuitive user interface for three-dimensional modelers. This has been shown with the example of a furniture design program that should interact with a room viewer.

Persistent use of direct manipulation techniques in 3D has allowed a very intuitive user interface. Other very important features include a fully functional collision detection which makes object movement more realistic, a hierarchical grouping and focussing mechanism which makes it easy to make changes in the hierarchy without breaking it up, automatic measuring which allows easily building objects with correct dimensions, virtual design tools like rulers that allow measuring distances; and mixed scaling function for more realistic behavior of objects.

The current prototype is capable of most of the functions described in this paper, but there is still much work to do to make this a commercially useable product.

References

[Bier86] Eric A. Bier: „Skitters and Jacks: Interactive 3D Positioning Tools“. Proceedings of 1986 Workshop on Interactive 3D Graphics, S. 183-196, 1986

[Bier90] Eric A. Bier: „Snap-Dragging in Three Dimensions“. SIGGRAPH '90 Computer Graphics, S. 193-204, 1990

[Chen88] M. Chen, J. Mountford, A. Sellen: „A Study in Interactive 3-D Rotation Using 2-D Control Devices“. SIGGRAPH '88 Computer Graphics, Vol. 22, No. 4, S. 121-129, 1988

[Databecker95] „3D Wohndesigner“, DataBecker Software, 1995

[Eckstein96] Erik Eckstein: „Interactive Constraint-Based Room-Designer“. Diplomarbeit, Technische Universität Wien, Institut für Computergrafik, 1996

[Emmerik90] Maarten J.G.M. van Emmerik: „A Direct Manipulation Technique for 3D Object Transformations with a 2D Input Device“. Computer Graphics Forum, Vol. 9, S. 355-361, 1990

[Emmerik91] Maarten J.G.M. van Emmerik: „Interactive design of 3D models with geometric constraints“. The Visual Computer, Vol. 7, S. 309-325, 1991

[Fa93] Mingxian Fa, Terrence Fernando, Peter M. Dew: „Direct 3D Manipulation Techniques for Interactive Constraint-based Solid Modeling“. Eurographics '93, Vol. 12, No. 3, S. 237-248, 1993

[Foley90] James D. Foley, Andries von Dam, Steven K. Feiner, John F. Hughes: „Computer Graphics: Principles and Practice“. Second Edition, Addison-Wesley Publishing Company, 1990

[Grimm95] Cindy Grimm, David Pugmire, Mark Bloomenthal, John Hughes, Elaine Cohen: „Visual Interfaces for Solids Modeling“. User Interface (UIST) 1995, S. 51-60, 1995

[Liang95] Jiandong Liang, Mark Green: „JDCAD: A Highly Interactive 3D Modelingsystem“. Computer & Graphics, Vol. 18, No. 4, S. 499-506, 1994

[Nielson86] Gregory M. Nielson, Dan R. Olsen Jr.: „Direct Manipulation Techniques for 3D Objects Using 2D Locator Devices“. Proceedings of 1986 Workshop on Interactive 3D Graphics, S. 175-182, 1986

[Requicha92] Aristides A.G. Requicha, Jarek R. Rossignac: „Solid Modeling and Beyond“. IEEE Computer Graphics & Applications, September 1992, S. 31-44

[Rossignac86] Jaroslaw R. Rossignac: „Constraints in Constructive Solid Geometry“. Proceedings of 1986 Workshop on Interactive 3D Graphics, S. 93-110, 1986

[Shneiderman83] Ben Shneiderman: „Direct Manipulation: A Step beyond Programming Languages“. IEEE Computer, Vol. 16, S. 57-69, 1983

[Sistare91] Steve Sistare: „Graphical Interaction Techniques in Constraint-Based Geometric Modeling“. Proceedings of Graphics interface '91, S. 85-92, 1991

[Strauss92] Paul S. Strauss, Rikk Carey: „An Object-Oriented 3D Graphics Toolkit“. Computer Graphics, Vol. 26, No. 2, S. 341-347, 1992

[Wernecke94] Josie Wernecke, OpenInventor Architecture Group: „The Inventor Mentor“. Addison-Wesley Publishing Company, 1994

[Yoshimura95] Tetsuya Yoshimura, Yasuhiro Nakamura, Masataka Sugiura: „3D Direct Manipulation Interface: Development of the Zashiki-Warashi System“. Computer & Graphics, Vol. 18, No. 2, S. 201-207, 1994