# Moving Objects with 2D Input Devices in CAD Systems and Desktop Virtual Environments

Ji-Young Oh, Wolfgang Stuerzlinger

Computer Science, York University
Toronto, Ontario, Canada
http://www.cs.yorku.ca/~{jyoh,wolfgang}

*Abstract*

Part assembly and scene layout are basic tasks in 3D design in Desktop Virtual Environment (DVE) systems as well as Computer Aided Design (CAD) systems. 2D input devices such as a mouse or a stylus are still the most common input devices for such systems. With such devices, a notably difficult problem is to provide an efficient and predictable object motion in 3D based on their 2D motion. This paper presents a new technique to move objects in CAD/DVE using 2D input devices.

The technique presented in this paper utilizes the fact that people easily recognize the depth-order of shapes based on occlusions. In the presented technique, the object position follows the mouse cursor position, while the object slides on various surfaces in the scene. In contrast to existing techniques, the movement surface and the relative object position is determined using the whole *area* of overlap of the moving object with the static scene. The resulting object movement is visually smooth and predictable, while avoiding undesirable collisions. The proposed technique makes use of the framebuffer for efficiency and runs in real-time. Finally, the evaluation of the new technique with a user study shows that it compares very favorably to conventional techniques.

*Keywords: 3D object manipulation, Desktop Environment, Computer-Aided Design .*

## 1 Introduction

Moving objects is one of the most basic tasks of scene construction. When people design a scene with multiple objects, they repeatedly realign or adjust different parts, to explore the design space. Our goal is to provide an efficient and smooth object motion technique aimed at facilitating this explorative process in Computer Aided Design (CAD) and Desktop Virtual Environment (DVE) systems.

In the desktop environment, the mouse has proven to be an excellent input device for 2D user interface. It affords relatively precise input on a stable supporting surface. In today's CAD/DVE systems, the mouse or stylus is also commonly used to move objects in 3D environments. However, this brings up the problem of how to map the 2D input to 3D motion.

Providing a natural mapping from 2D input to 3D position is a difficult problem, usually faced by CAD/DVE interface developers. The simplest solution is to provide handles or widgets for explicit 3-axis manipulation. This solution has been adopted by many conventional CAD systems. While this solution allows no room for failure or unexpected results, the task of moving an object becomes tedious, as the user has to mentally separate the desired 3D movement into 1D or 2D components. Moreover, if objects are touching, these objects may make it difficult or even impossible to manipulate an object, as the handles can be occluded.

As documented by research into visual perception, people judge 3D position by many cues. Besides perspective, one of the most important cues for 3D position is occlusion [9], which helps humans to perceive the position of an object in relation to other objects. It is important to note here that (almost) all objects in the real world are attached or connected to other objects, which explains why the human visual system has adopted this strategy. Furthermore, humans frequently use the contact of an object against the surfaces of other objects to position it accurately. Another cue for 3D perception is stereo. However, from an end-users point of view, most stereo technologies are not very mature and are tiresome and/or problematic to use on a daily basis (e.g. [12, 13]).

Most 3D systems rely only on the mouse pointer position (i.e. a point) to map 2D input to 3D movement. However, research into vision in primates has shown that the perceptive field for an object that is being held in the hand covers the whole object [5]. In other words, there is strong evidence that the whole visual area of an object is used to judge position. Another explanation is that humans perceive a manipulated object as an extension of their body. Hence, there is potentially a big gap between how people perceive object position and the way current CAD/DVE systems handle this problem.

Based on these observations, we came up with a novel technique to move an object in a 3D virtual world. As the user moves an object, he/she utilizes his/her knowledge of the area of the surface(s) hidden by the moving object. We utilize this fact by always moving the object on one of the surfaces it occludes. In most current 3D systems, this is not guaranteed, as objects may float in the air and do not attach to other surfaces by default. In our informal observations on people manipulating objects in 3D, people seem to be fairly surprised when they find the objects are floating.

Hence, we designed our new technique so that default is that objects always stay attached to other objects. More precisely, we look for the closest visible surface behind the moving object and move the manipulated object onto it. Finding visible surfaces can be done very efficiently with graphics hardware. We can even exploit the capabilities of modern graphics hardware to avoid object interpenetration.

## 2 Previous Work

Strauss categorized possible solutions to the problem of mapping 2D input to 3D movement in the SIGGRAPH 2002 course notes [8] as follows:

1) Let a user select a movement axis by providing handles for all three axes. This necessitates (potentially tiresome) extra steps for non-trivial movements, but is always guaranteed to work.
2) Move an object on a plane parallel to the viewing plane. Although this is simple to implement, in general this technique does not work for users. The resulting movement is not intuitive and frequently misleading.
3) Use obvious structures in the scene to determine the plane of motion. This fails if there are no convenient structures or if they don't align with the user's intentions.
4) Use heuristics to decide the movement direction based on the initial cursor movement. As with any heuristic, this can fail.

The authors also say that there is no "perfect" solution, as there is no approach that is both easy-to-use and robust at the same time. Consequently, users need to frequently check if the object is the desired position, which can become tedious.

One of the approaches that follows category 3) is to use the ray from the eye point through the pixel currently selected by the mouse pointer to find the first intersection point with the scene. This ray is often called mouse-ray. For instance, Bier [1] used this approach in his snap-dragging technique. His approach searches for the visual feature (a vertex or an edge of an object or a grid-line) closest to the mouse-ray, and snaps the 3D cursor to that feature. The user can choose to accept the snapped position or ignore it by moving away from the feature. The scene is presented in wire-frame to avoid the occlusion of visual features. One limitation of this approach is that wire-frame display is not very user-friendly. Another limitation is that as the complexity of the scene increases, snap-dragging will snap to many features and usability will suffer.

Some systems (e.g. [2, 7]) utilize pre-defined object behaviors to limit object motion. As an example, consider a simple behavior, which constrains an object to move on a horizontal (or vertical) surface. With this behavior, the object will snap to the intersection point on any horizontal surface that the mouse-ray intersects. The concept of behaviors can be used to enable users to quickly populate a room with objects with predefined behavior, such as furniture, books, etc. One major drawback is that the behaviors have to be predefined, and that the definition of adequate behaviors requires a good understanding of the underlying geometrical concepts.

The Virtual Lego system [6] introduced a solution to this problem. This system uses "smart" Lego blocks, which snap automatically to any horizontal or vertical surface. Thus, constraints are implicitly defined within the system. In contrast to previous work, the virtual Lego system does not use the mouse-ray to find the snapping surface. Instead, it looks for the foremost surface behind the moving object. User evaluations showed that novice users could quickly grasp how the system worked and were even able to complete challenging tasks with minimal training. However, the Virtual Lego system can only deal with convex rectangular blocks and the techniques do not extend to more general types of objects.

Clearly, it is advantageous to employ collision detection to prevent objects from interpenetrating. Solving this problem in real-time is non-trivial, but some solutions have been presented (e.g. [3]). Interestingly enough, some of the most recent approaches use the framebuffer to speed computations (e.g. [4, 10, 11]).

## 3 Moving Objects in 3D with a 2D Device

For this paper, the main goal is to provide a visually smooth and predictable object motion, without limiting the user to axis-aligned motions or predefined object behaviors. The fundamental idea is to find a movement surface, and map the mouse movement onto movement on that surface. As a result, the selected object will appear to slide on that surface, while still following the mouse cursor.

At first, we re-implemented a technique used in other CAD/DVE systems and used the surface that the mouse-ray hits as the movement surface. However, this approach leads to unpredictable results, since the user will generally select an object by clicking on an arbitrary point on the object to be moved. However, selecting an arbitrary point on the object results in different motions, depending on which part of the object is selected. More precisely, the object motion depends on the relative position of the mouse cursor on top of the moving object. Figure 1 illustrates this problem. In Figure 1(a), object *P* is “held” on the center of the front face of the object and slides on face *s* of the background object. A pointer movement to the left will then drop the cube onto face *t*. In Figure 1(b), object *P* is “held” on the right face and the same amount of (pointer) movement to the left achieves a different result as the mouse-ray still intersects face *s* in this case.

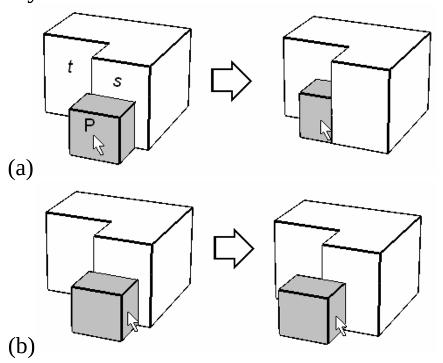

*Figure 1. Techniques that use the surface directly behind the mouse pointer suffer from non-predictability, as the same mouse-motion may generate different results.*

The results of this naïve technique suffer from the fact that a single point determines the object motion. This leads to ambiguous and unpredictable mapping between user input and 3D motion. According to our observations, most users believe that both actions should result in the same result.

Another problem is that the object appears to jump when it snaps to a different movement surface. In Figure 1(a) a relatively small mouse movement results in a significant motion in depth. This is especially noticeable when the background object is relatively smaller than the moving object since the surface behind the object will be occluded.

Lastly, this method does not automatically avoid collisions. E.g. in Figure 1(a), the moving cube actually collides with the geometry of the background object in the final position. This can only be avoided by adding a collision detection method to the implementation. However, it is unclear how the object should move when a collision is detected.

## 4 A New Technique for 3D Movement

As discussed in the introduction, the new technique uses the whole *area* behind the moving object to determine object motion. More precisely, we use the foremost hidden surfaces to determine object motion.

Figure 2 shows an overview of our algorithm. In Figure 2(a) the object slides on surface *s*. As the mouse cursor moves further to the left, the algorithm moves the object on the same plane towards the position in Figure 2(b). In this position, *s* is still the foremost surface behind the cube and consequently the cube continues to move in this plane and is positioned as depicted in Figure 2(c). In Figure 2(d), the cursor has moved even further to the left and the image of the cube does not overlap with surface *s* anymore. Hence, the algorithm snaps the cube onto surface *t* and continues to slide it on this plane.

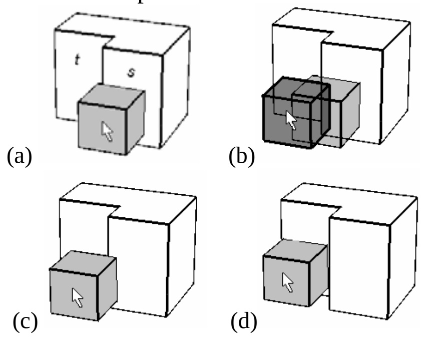

*Figure 2. Objects slide on the surface that is both closest to the viewer and occluded by the object.*

There are two alternatives to find the movement surface in this scheme. The first is to take the foremost surface of the scene that overlaps with the image of the moving object. The second is to use the foremost surface that is behind the moving object, regardless whether the entire moving object is visible or not.

The attractiveness of the first alternative is its simplicity, as we only need to determine the foremost surface of the (static) scene, regardless of the current position of the moving object. Furthermore, this method ensures that the moving object is always closer to the viewer than the rest of the scene. This implicitly guarantees that there are no collisions with other objects. This is an interesting feature, as it means that this is a manipulation technique that does not require a separate collision detection scheme, which means that even extremely complex parts can be handled efficiently. The downside of this method is that when a scene is cluttered with many objects, and there are consequently many surfaces, then the moving object will jump frequently in depth, and positioning the object requires more attention from the user. A pilot study of an implementation of this idea showed that this is indeed a problem that users encounter in practice.

The second alternative again identifies the first surface behind the moving object, but ignores any surface closer to the viewer than the moving object. In this method, the moving object does not immediately pop out to the surface in front of the user, unless the moving object becomes the one closest to the viewer. Usually, this conforms better to the intentions of the user. The limitation of this alternative is that when a small object moves forward, it may penetrate another object in front. To address this problem, we employ a collision detection method. Once a collision is found, the object jumps also in front of the colliding object, as with the first alternative.

Figure 3 depicts several movement sequences with the original mouse-ray techniques and the two new techniques mentioned in this section. Here the goal is to slide the chair under the table. Figure 3(a) shows the movement based on the foremost surface behind the mouse position, determined via the mouse-ray. As soon as the mouse pointer overlaps with the surface of the table, the chair moves on top of it. However, when the mouse pointer moves off the table again, the chair drops immediately to the floor and ends up in a position, where it collides with the table, as shown in the fourth image. There are two ways to get the chair under the table with this technique. One is to change the viewpoint, as depicted in Figure 3(c). The other alternative is to "grab" the chair by the top part of the backrest (an area that is visually quite small) and to move it towards the table while avoiding overlap with the table itself. However, this is very non-intuitive, and most users do not realize that this is possible – especially since the position that needs to be "grabbed" is not obvious at all.

The image sequence depicted in Figure 3(b) illustrates the technique that utilizes the foremost surface inside the image of the moving object. As soon as the image of the chair overlaps with the table (second image), the chair starts to slide on the table surface. Only when the image of the chair does not overlap with the table anymore, does the chair drop down to the floor. Note that no collision occurs with this technique, but the only way to drag the chair under the table is again to change the viewpoint as depicted in Figure 3(c).

Finally, Figure 3(d) illustrates the new technique that utilizes the first surface behind the moving object. As the chair slides on the floor it continues to move underneath the table, because the first surface visible behind the chair is the floor. In the third image, the chair is clearly in the desired position and the user is finished. For illustration purposes, we continue this sequence with the fourth image from the left, where the chair collides with the table. When moving the object further the technique moves the chair onto the table (more precisely as soon as the backside of the backrest of the chair collides with the table). By incorporating a simple collision detection algorithm the situation is resolved by moving the chair onto the table as soon as the collision occurs and continuing as in the previous method.

## 5 Implementation

The second alternative presented above provides a more robust solution for object motion compared to the first one, hence we immediately present the implementation of the second alternative. Note that the first technique

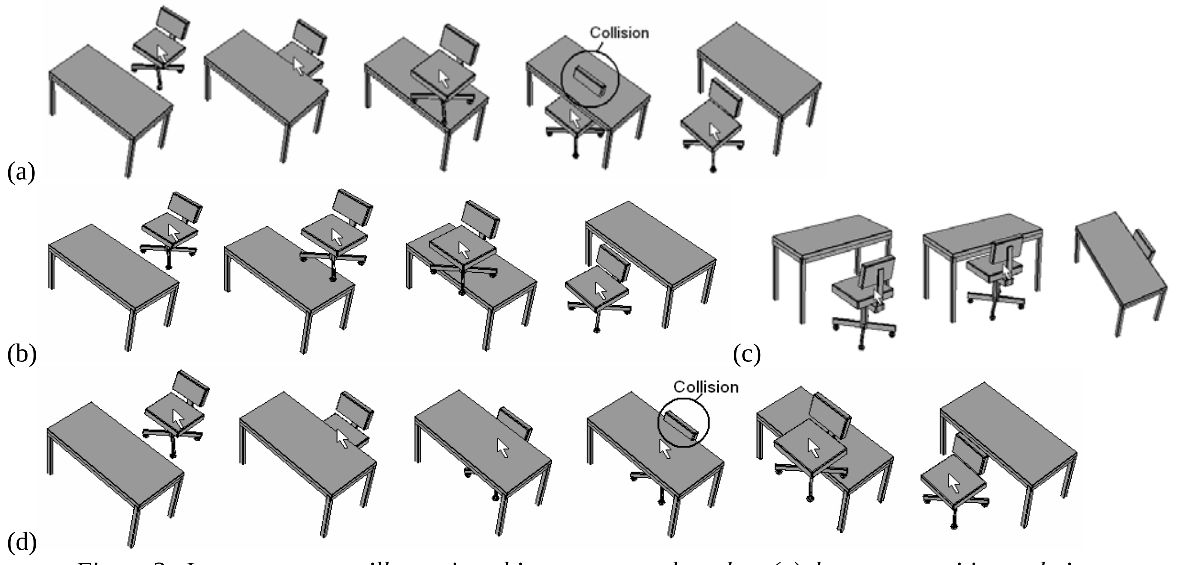

*Figure 3. Image sequences illustrating object movement based on (a) the mouse position technique, (b) & (c) a technique based on the foremost visible surface, and (d) our new technique based on the foremost surface behind the moving object. For a detailed explanation please refer to the text.*

can be implemented by simplifying some of the following steps.

### 5.1 Preprocessing: Store framebuffer for background

When a user starts moving an object, the algorithm first renders the static scene without the selected object into a background framebuffer. Each face of each object is given a unique color to enable easy and fast identification of surfaces. This idea is usually referred to as an *item buffer*. Then the color/item buffer and the depth buffer are read into main memory. With this information the algorithm can quickly identify for each pixel the foremost visible object and foremost visible face by indexing these two buffers with the current pixel position.

### 5.2 Step 1: Generate framebuffer for moving object.

On every frame, the system renders the *back-faces* of the moving object in its *old* 3D position into an empty framebuffer, by setting the depth comparison of the graphics card to render polygons further away and also using front-face culling. We also use an item buffer for quick identification of the back-faces of the moving object.

### 5.3 Step 2: Identify movement surface

The algorithm then identifies the movement surface by comparing the depth values of the *front-faces* of the static scene (computed in the preprocessing phase) and the *back-faces* of the moving object (computed in step 1). However, we shift the depth image generated in step 1 by the relative change of mouse position from the previous frame to account for the current mouse position. This is similar to moving the object parallel to the viewing plane.

```
// f[I]: Depth of front-face of static scene in pixel i
// b[i]: Depth of back-face of moving object in pixel i

mindiff = LARGE_POSITIVE_NUMBER
For all pixels i covered by the moving object
   diff = b[i] – f[i]
    If diff < mindiff And diff >= 0
       mindiff = diff
       location = i
   End If
End For

Return mindiff,location
```

*Figure 4. Pseudo-code for identification of the pixel that determines the movement surface.*

Figure 4 presents the pseudo-code for this step. In essence, we locate the smallest positive depth difference. This biases the object to stay on the current surface, until a major collision occurs. Using the pixel position of the minimum depth difference, we retrieve the corresponding face of the occluded object from the item buffer. Note that this computation is used only to identify the movement surface, but does not determine the actual object movement directly.

### 5.4 Step 3: Compute movement offset

Given that we now know which surface is the closest surface behind the object, we have to move the object on that surface. The depth difference computed from *step 2* is not directly useable to snap the object onto the movement surface due to the non-linear nature of the depth buffer and depth discretization artifacts. We use two phases to slide the object to its new position. In the first phase, we move the object on a surface parallel to the movement surface. In the second phase, we recompute the depth difference as in *step 2* to snap the object onto the movement surface.

In the first phase, we compute the plane that is parallel to the movement surface identified in the step 2 and that passes through the previous 3D position, which is determined by the intersection of the previous mouse ray and the moving object. Then we intersect this plane with the current mouse ray. Finally, we move the object by the vector between the computed intersection point and the previous 3D position. In the second phase, the object moves along the mouse-ray to snap onto the movement surface. To do this, we re-apply the algorithm presented in *step 2* to find the pair of closest faces and points on those faces. Then we move the object to align the two points in the scene.

After the motion, we check for interpenetration. If the object penetrates the rest of the scene, then we simply allow the object to pop to the front by repeating the algorithm used in step 2, but this time we allow the variable *mindiff* (Figure 4) to be negative – i.e. allow the object to pop closer to the viewer. To find potential collisions we use the technique introduced in [11]. In this paper, the system detects intersection of an edge of an object with a face of the other object using the framebuffer. The presented method may fail to detect intersection if the moving object is much larger than the static objects. However, in our technique, collision only occurs when a larger static object is closer to the viewer, occluding whole area of the moving object.

The logic behind the decision to compare depth values for the front surfaces of the static scene and the depth values of the back-face(s) of the moving object is illustrated in Figure 5. In this figure, object *T* is moving toward object *S*. Once object *T* touches object *S*, face $t_1$

in object *T* is closest to face $s_1$. Then, as it moves further towards the face $t_1$ and before $t_2$ collides with object *S*, the object must move upwards, to place $t_3$ on top of $s_1$, so that object *T* can continue to move freely. This can be expressed as the problem of identifying the closest points between the back-faces of the moving object and the front-faces of the static scene. Of course, the test will fail when there is no overlap between the images of the objects involved. In this case, we assume that the object is in free space, a case that is discussed in the next subsection.

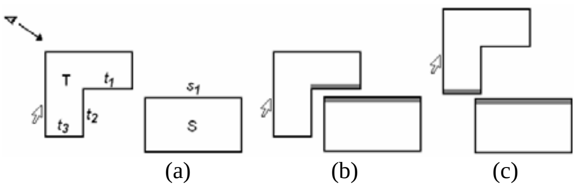

*Figure 5. . Object movement scenario (a) Object T approaches object S. (b) The face $t_1$ slides on face $s_1$ of object S. (c) As the object T move further, the face $t_3$ must pop up on top of face $s_1$.*

## 5.5 Movement in free space

In our technique, an object slides on the foremost surfaces of the scene. However, sometimes there is no surface behind the moving object. In this case the algorithm detects no minimum depth difference in *step 2*. For simplicity, we move the object in free space on an axis-aligned plane in this case. The choice of this plane is dictated by viewing direction. That is, the system chooses the axis-aligned plane that is most orthogonal to the view-direction, as this provides the most predictable mapping between 2D input and object motion for this case according to our observations.

## 5.6 Discussion

The presented technique works well for general shapes, even for objects that have large concavities or curved surfaces. For curved surfaces, the API's of current graphics cards necessitates an approximation of the curved surface into many small planar surfaces, which allows our algorithm to work without a problem.

To demonstrate the speed of our algorithm, we assembled a lounge chair consisting of 13,196 polygons (Figure 6) with our system. On a 1.2GHz computer system with an NVIDIA GeForce2 graphics card, we recorded an average frame rate of 32fps during manipulation of the parts.

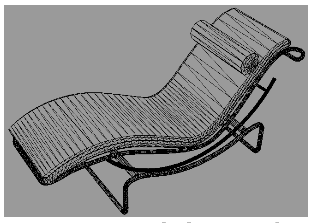
*Figure 6. Example lounge chair*

However, we found several limitations to our technique mainly because we use visible surfaces in the background scene to find the movement surface. The first limitation is due to the view dependency. The technique identifies a movement surface among the currently visible surfaces. When the surface where the user wants to place the object is not visible due to the viewing direction, then the user cannot place the object in the desired position. Therefore, our technique requires that the user navigate to an appropriate viewpoint in the scene to place an object.

The second limitation is that, if the object visually occupies a large part of the screen (e.g. more than half), object movement may become difficult. This is because the object itself occludes a large part of the scene, making it hard for the user to judge where the object is currently relative to the rest of scene. The easiest way to address this problem is to move the camera further away, so that the desired position is more clearly visible in screen space.

Another situation that is problematic is a bolt that slides into a tightly matched hole. The algorithm can deal with this situation *only* if the user looks more or less straight down into the hole. The reason behind this is that current graphics hardware renders polygons slightly larger to avoid cracks between adjacent surfaces. In other words, all pixels that are “on the edge” are set. For our techniques, this makes it impossible to put the bolt into the hole, as the polygons overlap in screen space by one pixel. One (simplistic) solution is to render the moving object slightly smaller than it really is. A better solution is to configure the graphics card to render only pixels that are inside of the polygons, which ameliorates the situation. However, due to potential rasterization artifacts it is currently not always possible to *guarantee* that this will work perfectly in every situation.

## 6 Evaluation with User Study

We conducted a 2 (techniques) x 2 (display) factorial test, which compares the presented technique and the conventional axis-handle technique, each of them in two display conditions. The first display condition uses four views (3 orthogonal views along the 3 major axis and one perspective view, as shown in Figure 7a), and the second condition is a single perspective view.

The axis-handle scheme (Figure 7b, c) and the four-view window layout resemble the interface of Maya$^{TM}$, a widely used CAD/animation package. In the axis-handle scheme the user can drag the object by clicking on various parts of the handles. If a user selects the cube-part of the handle, the object moves on the (2-dimensional) plane that is parallel to the corresponding axis. If the user selects the cone part of the handle, the object moves along the corresponding (1-dimensional) axis. Collision detection is not provided in this condition, since users may need to move objects into arbitrary positions and should not be blocked by other objects on the way.

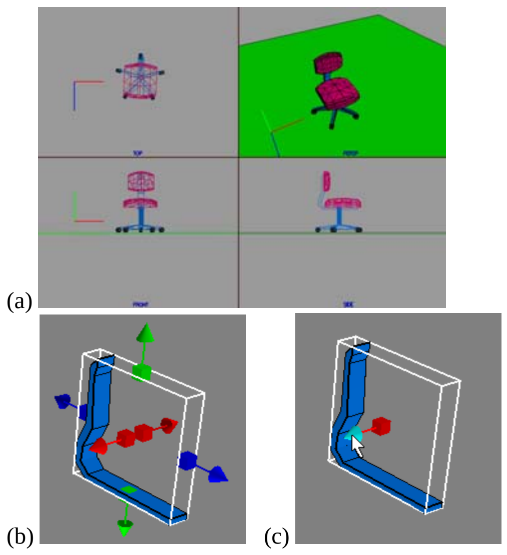

*Figure 7. (a) Four-view display: top, perspective, front, and side view, in clockwise, (b) Axis handles (c) Once a handle is selected, the other irrelevant handles are eliminated from the display.*

In the beginning, we planned to compare three techniques: the handles technique presented above, the mouse-ray technique and our technique. However, in a pilot test, we saw that naïve users could not readily distinguish between the mouse-ray technique and our technique after the short learning period (about 20 min to learn three techniques), as their a-priori appearance does not differ significantly. This led to significant confusion and frustration. To eliminate this confounding factor, we remove one condition from our experiment and chose to compare only the handle technique with our new technique.

Ten paid participants (Age: 20-35, 6 male, 4 female) were recruited from the local university. All of them had little or no previous experience with CAD systems. They were given the task of assembling a chair as shown in Figure 8. The initial configuration is depicted in Figure 8(a) and the assembled chair is shown in Figure 8(b).

To eliminate another potentially confounding factor, namely varying "3D construction" skills, we informed the participants about the correct order of the parts to be moved. To simplify the task and to eliminate another potential problem, the five rollers of the chair moved as one (i.e. as if they were one object). Participants were explicitly instructed not to move the rollers and start moving *shape #1* in Figure 8(a) onto the rollers.

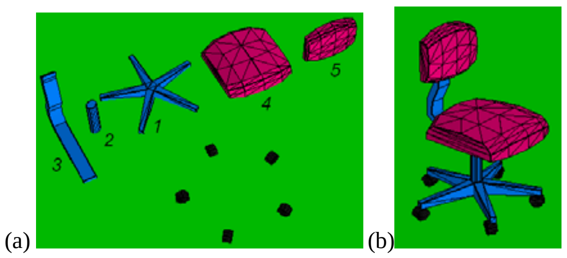

*Figure 8. The experimental task, assembling a chair, (a) Start scene, (b) Target scene*

After a short introduction period, where users could practice movements on a scene with a few boxes, the participants conducted the chair construction task twice for each of the conditions of the 2x2 experiment. To eliminate potential learning effects we counterbalanced the order of the techniques. Consequently, users assembled the chair eight times. In the analysis, we considered the first iteration of each condition as practice, and we present only the results from the second iteration of each trial. We identify our movement technique as *OFS* (Overlap with Foremost-Surface), and the conventional axis-handle technique as *handle*. The condition with four views is called *four-view*, and the single view condition *single-view*. At the end of the experiment, users were given a questionnaire to rate preferences among the techniques.

## 6.1 Results

An ANOVA analysis of the results reveals that the difference between the *handles* and *OFS* conditions was significant ($F_{9,1}$=47, $p<0.001$). In fact, participants took about twice as long to complete the task with *handles*.

The difference between the *four-view* and *single-view* conditions was also significant ($F_{9,1}$= 18.74, $p<0.05$), and *four-view* was the faster alternative. We attribute this to the fact that users did not have to change the viewpoint as often in this condition.

Table 1 and Figure 9 illustrate the overall results.

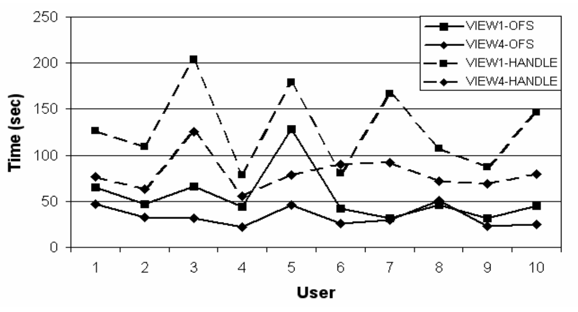

*Figure 9. Task completion time by user*

*Table 1. Average task completion time*

| | OFS | Handle | Average | SD[*] |
|---|---|---|---|---|
| Single-view | 54.64 | 128.46 | 91.55 | $p<0.001$ |
| Four-view | 33.34 | 80.11 | 56.72 | $p<0.001$ |
| Average | 43.99 | 104.29 | - | - |
| SD | $p<0.05$ | $p<0.05$ | - | - |

* SD: Significance of difference obtained from repeated ANOVA test

Analyzing the data further, the average navigation time using *handles* in the *single-view* condition is 25.46s (seconds) and the average navigation time using *OFS* in the *single-view* condition is 13.60s. These two data points are significantly different ($F_{9,1}=78.89$, $p<0.001$). This is an interesting result in that even though *OFS* requires navigation in some cases (as mentioned in section 5.6), users spent less time on navigation using *OFS*. As an object moves freely in space with the *handles* technique, the users seemed to check the 3D position of the object after almost every object motion, while this was not necessary in *OFS*. Even more interesting is the difference between *handles* in the *four-view* condition and *OFS* in the *single-view* condition. Even though *OFS* is relatively speaking handicapped by the *single-view* display, the completion time was significantly less than that of the *handles* with the *four-view* display ($F_{9,1}=6.26$, $p<0.05$)!

We believe that these results demonstrate that users understood the 3D position of the object relative to the scene much more easily using *OFS* compared to *handles*, as *OFS* does not allow an object to move freely in space *by default*. Moreover, our results suggest that our technique can work also efficiently in Virtual Environments where a *four-view* like display is not suitable. We plan to explore this in future work.

Analysis of the results from the questionnaire yielded that six out of ten participants preferred *OFS*, two preferred *handles*, and two were neutral. The users who preferred *OFS* stated that the technique is efficient and intelligent in a sense that the system knows where an object should be moved based on the user's action. On the other hand, the users who preferred *handles* stated that they could adjust the position of an object more precisely. Users who were neutral between the techniques mentioned a mix of all the reasons above.

## 7 Conclusion

This paper presents a new technique to move objects in 3D using a 2D input device. Our technique does not utilize axis-handles or widgets, as generally provided by conventional CAD/DVE systems. The new technique enables natural 3D motion of objects while avoiding collisions. The technique is works for arbitrary types of shapes and runs in real-time on current graphics hardware

The user evaluation showed that users could understand the 3D position of an object more easily with our method compared to the most frequently used conventional method. Finally, the evaluation suggests that the presented technique may significantly improve the efficiency of object manipulation in CAD/DVE systems.

## References

[1] E. Bier, Snap-dragging in three dimensions. ACM Computer Graphics, 24(2): 193-204, 1990.

[2] R. Bukowski and C. Sequin, Object associations: a simple and practical approach to virtual 3D manipulation. SI3D'95, 131-138, 1995.

[3] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: a hierarchical structure for rapid interference detection. SIGGRAPH '96, 171-80, 1996.

[4] N. K. Govindaraju, S. Redon, M. C. Lin, and Dinesh Manocha, CULLIDE: interactive collision detection between complex models in large environments using graphics hardware. SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, 25-32, 2003.

[5] S. Obayashi, T. Suhara, K. Kawabe, T. Okauchi, and J. Maeda, Functional brain mapping of monkey tool use, NeuroImage 14: 853-861, 2001.

[6] J.-Y. Oh and W. Stuerzlinger, Intelligent Manipulation Techniques for Conceptual 3D Design, IFIP Interact 2003.

[7] G. Smith, and W. Stuerzlinger, Integration of Constraints into a VR Environment, VRIC 2001, 103-110, 2001.

[8] P. S. Strauss, P. Issacs, and J. Shrag, The design and implementation of direct manipulation in 3D. SIGGRAPH 2002 Course Notes, 2002.

[9] C.D. Wickins, and J.G. Hollands,. Chapter 4. Spatial displays, in Engineering psychology and human performance, Prentice-Hall, 3rd Ed. 1999.

[10] B. Heidelberger, M. Teschner, and M. Gross, Detection of collisions and self-collisions using image-space techniques, *WSCG* 2004.

[11] K. Dave, K. P. Dinesh, CInDeR: Collision and Interference Detection in Real-time using graphics hardware, Graphics Interface, 2003.

[12] Z. Wartell, L. F. Hodges, W. Ribarsky, A geometric comparison of algorithms for fusion control in stereoscopic HTDs, IEEE Transactions on Visualization and Computer Graphics, 8, 129-143, 2002.

[13] D. B. Diner, Fender, D.H., Human Engineering in Stereoscopic Viewing Devices. Plenum Press, New York and London, 1993.