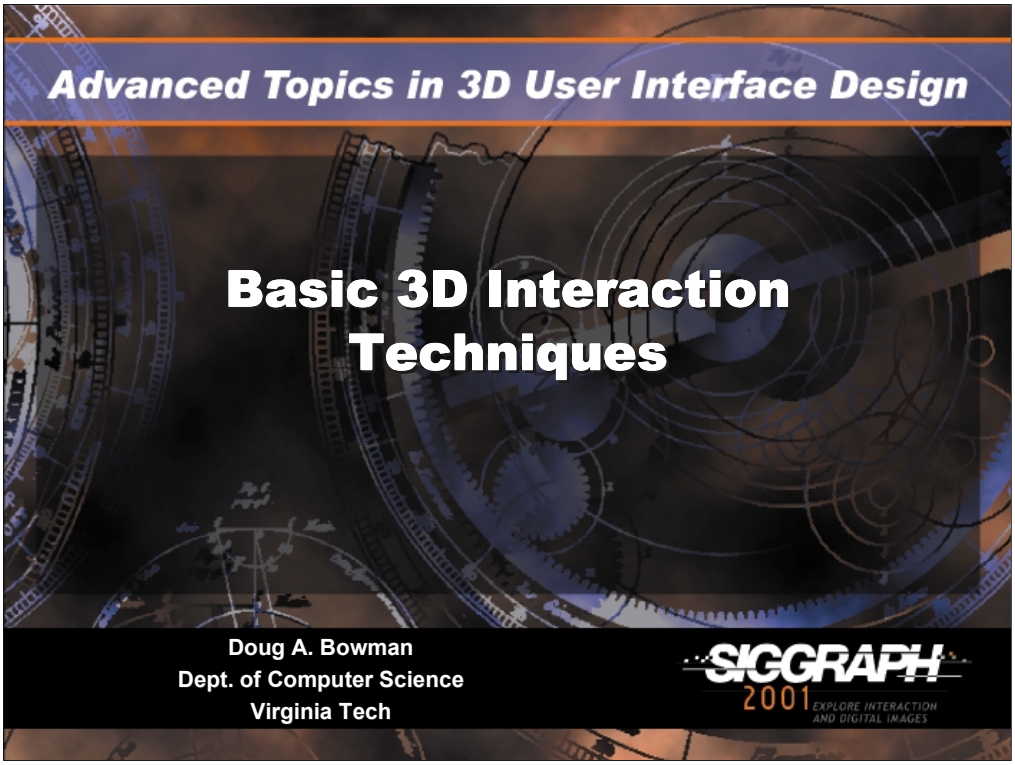


Basic 3D interaction techniques

Doug A. Bowman

Dept. of Computer Science (0106)

660 McBryde Hall

Virginia Tech

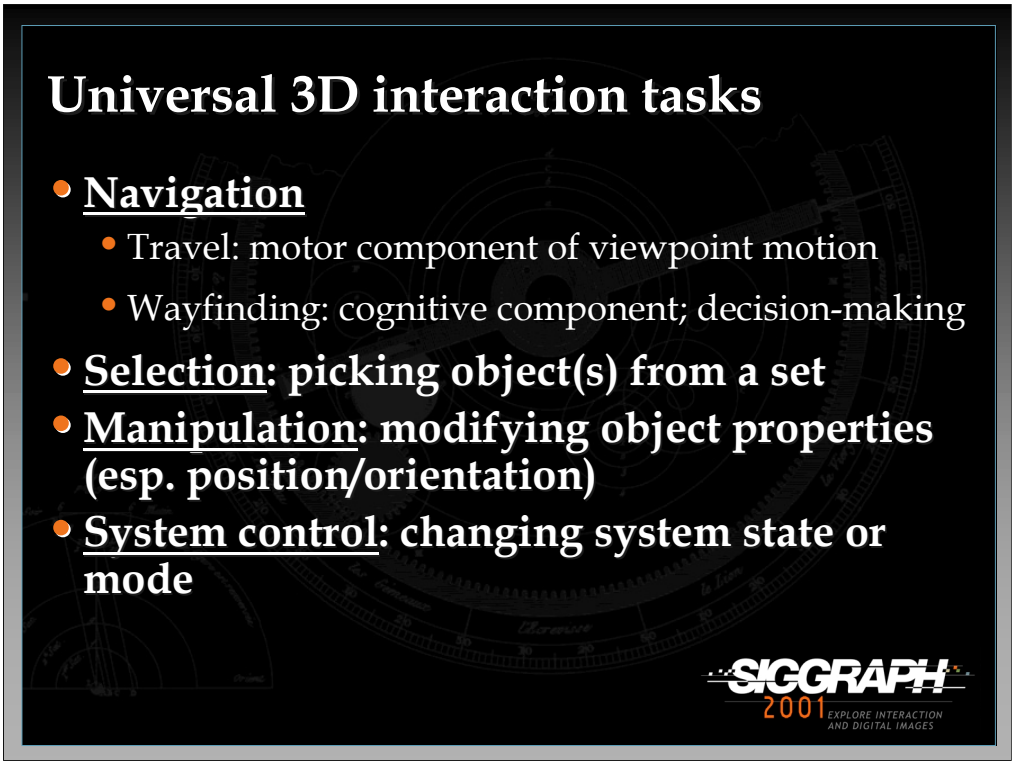
Blacksburg, VA 24061 USA

Email: bowman@vt.edu

Web: http://www.cs.vt.edu/~bowman/

In this lecture, we'll describe several common techniques for basic 3D tasks. All of these techniques were described in last year's course notes, with an emphasis on their advantages and disadvantages, and on their usability characteristics. This year, for the advanced course, we take a different approach. We describe details of the *implementation* of these techniques, including mathematics needed to properly implement them. This information has not been compiled in one place before, to our knowledge, so we hope it will be useful as you implement your own 3D interfaces.

The techniques we describe have mostly been developed in the context of VEs, but many of them are useful in other types of 3D systems as well. We also have a lecture on non-VE interaction later in the course.



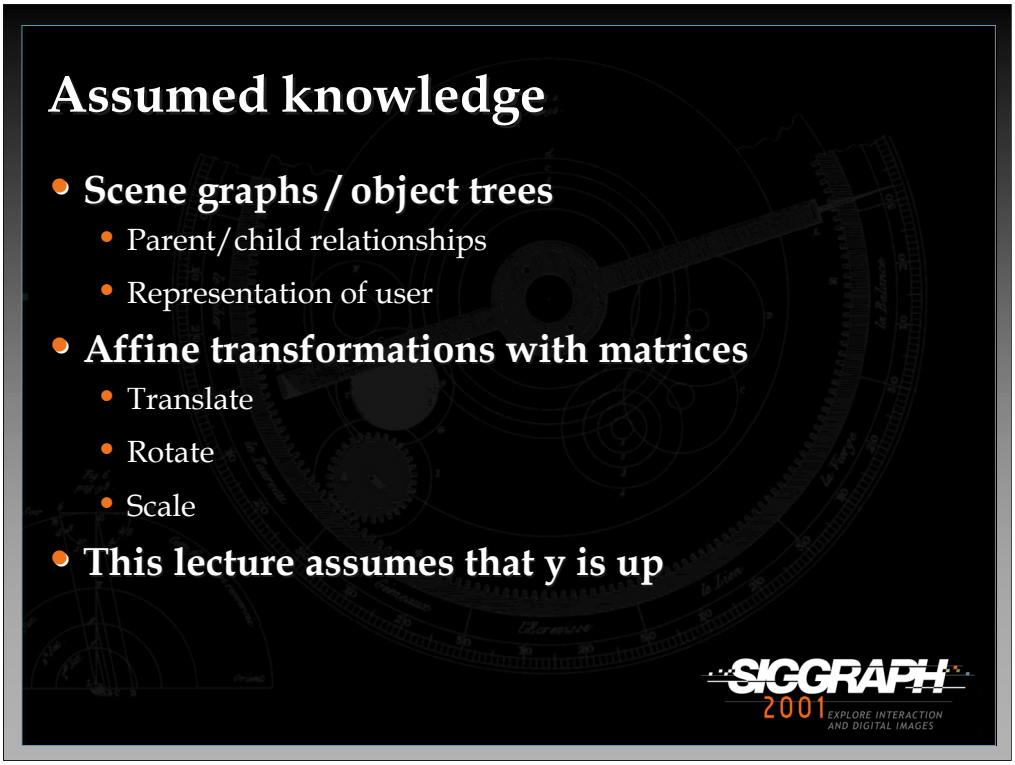
We'll be discussing techniques for four basic 3D interaction tasks that are found in most complex 3D applications Obviously, there are other tasks which are specific to an application domain, but these are some basic building blocks that can often be combined to create a more complex task.

Navigation is the most common VE task, and is actually composed of two tasks. Travel is the motor component of navigation, and just refers to the physical movement from place to place. Wayfinding is the cognitive or decision-making component of navigation, and it asks the questions, "where am I?", "where do I want to go?", "how do I get there?", and so on.

Selection is simply the specification of an object or a set of objects for some purpose. Manipulation refers to the specification of object properties (most often position and orientation, but also other attributes). Selection and manipulation are often used together, but selection may be a stand-alone task. For example, the user may select an object in order to apply a command such as "delete" to that object.

-continued on the next page

System control is the task of changing the system state or the mode of interaction. This is usually done with some type of command to the system (either explicit or implicit). Examples in 2D systems include menus and command-line interfaces. It is often the case that a system control technique is composed of the other three tasks (e.g. a menu command involves selection), but it's also useful to consider it separately since special techniques have been developed for it and it is quite common.

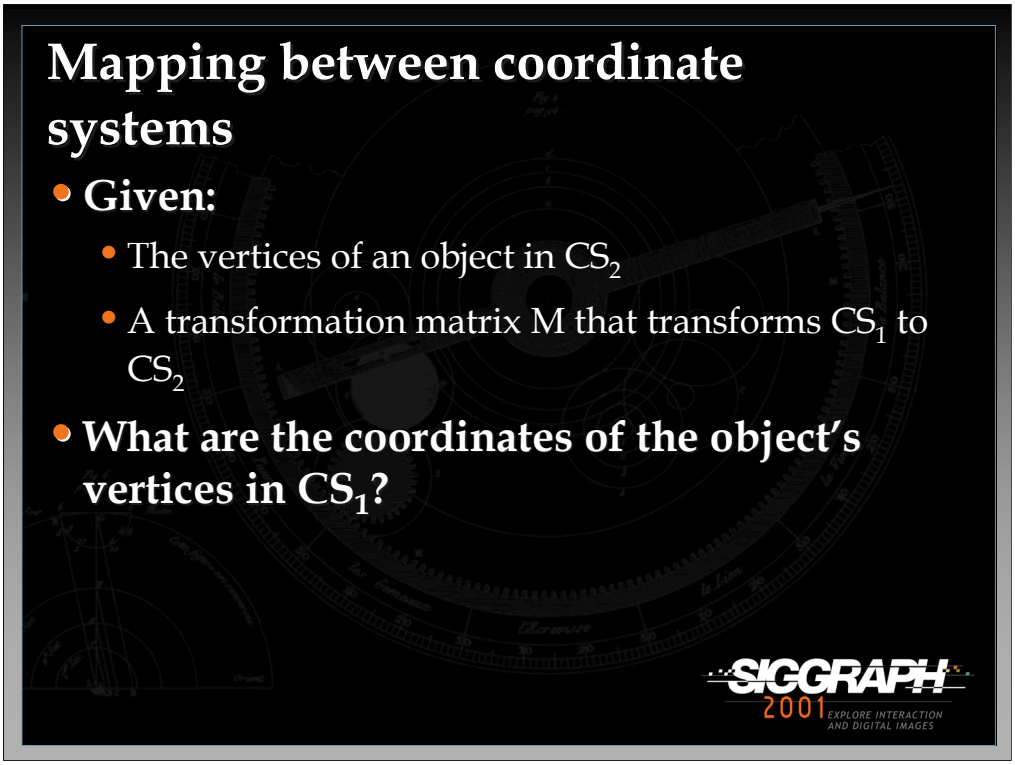


We are assuming a basic level of knowledge in this lecture, so that we do not have to cover the lowest-level implementation details for these interaction techniques. These concepts should be familiar to anyone who has done graphics programming.

First, we assume you are familiar with hierarchical scene graphs or object trees in 3D graphics systems, including the concept of parent/child relationships between objects, inherited transformations, and the representation of the user in a scene graph.

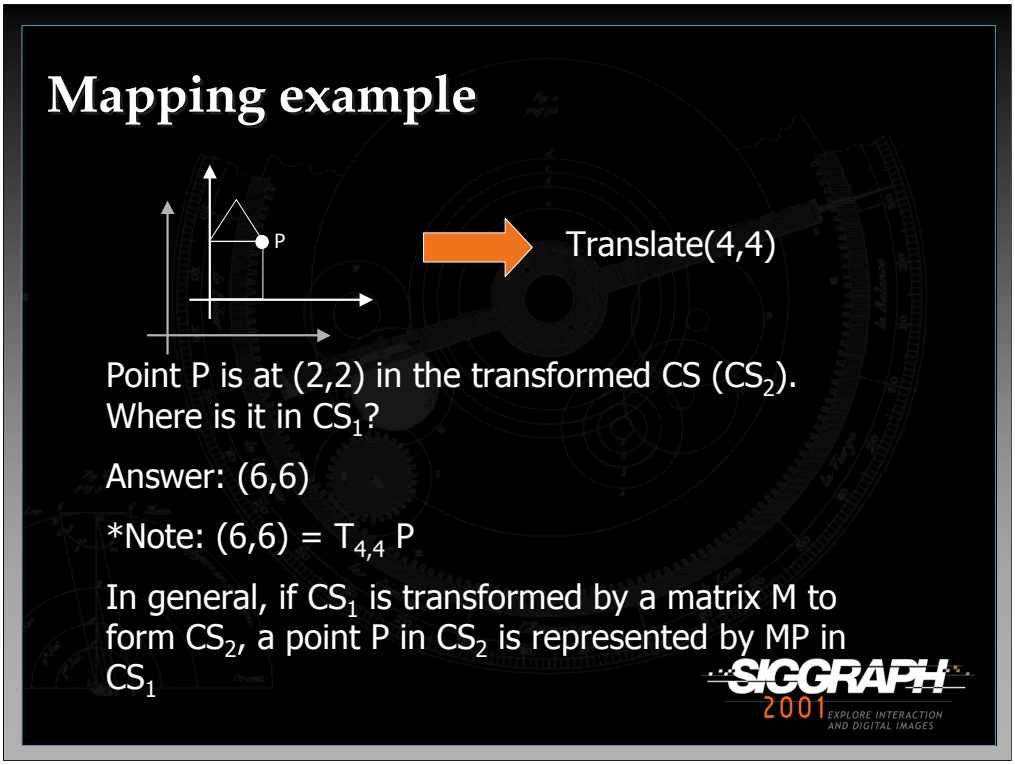
Second, we assume you are familiar with matrix representations of simple transformations such as translate, rotate, and scale, and with the concept of composite matrices for multiple transformations.

Finally, a note on notation: we assume (for the techniques where it makes a difference) that ‘y’ is the vertical axis in the world coordinate system. Some systems will use ‘z’ as the vertical axis.



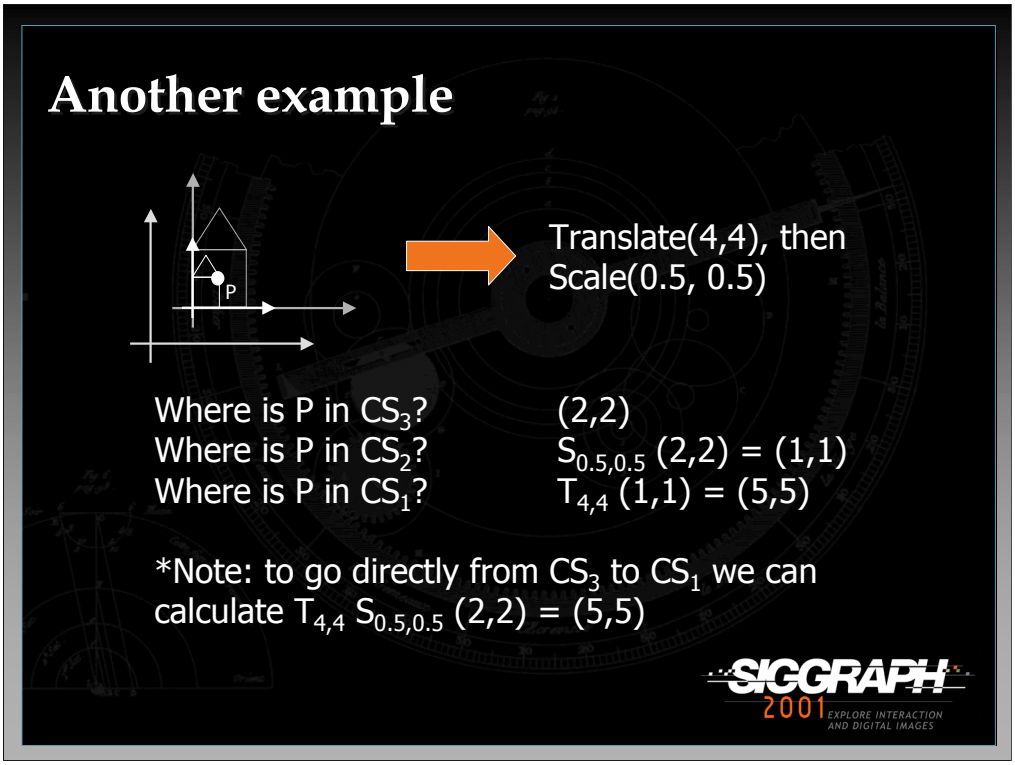
Before we begin with the techniques, we need to review an important concept. In most graphics systems, objects and their vertices can be described in two ways: their location (position, orientation, and scale) in a fixed world coordinate system, or in a local coordinate system attached to the object. For many techniques, we will need to distinguish between world and local coordinate systems, and in some cases move from one representation to another. Therefore, we discuss here the concept of mapping between coordinate systems.

The problem we want to solve is this: given the vertices of an object in one coordinate system (say its local coordinate system), and a transformation matrix M that transforms another coordinate system (say the world CS) to the first, what are the coordinates of the vertices in the world coordinate system?



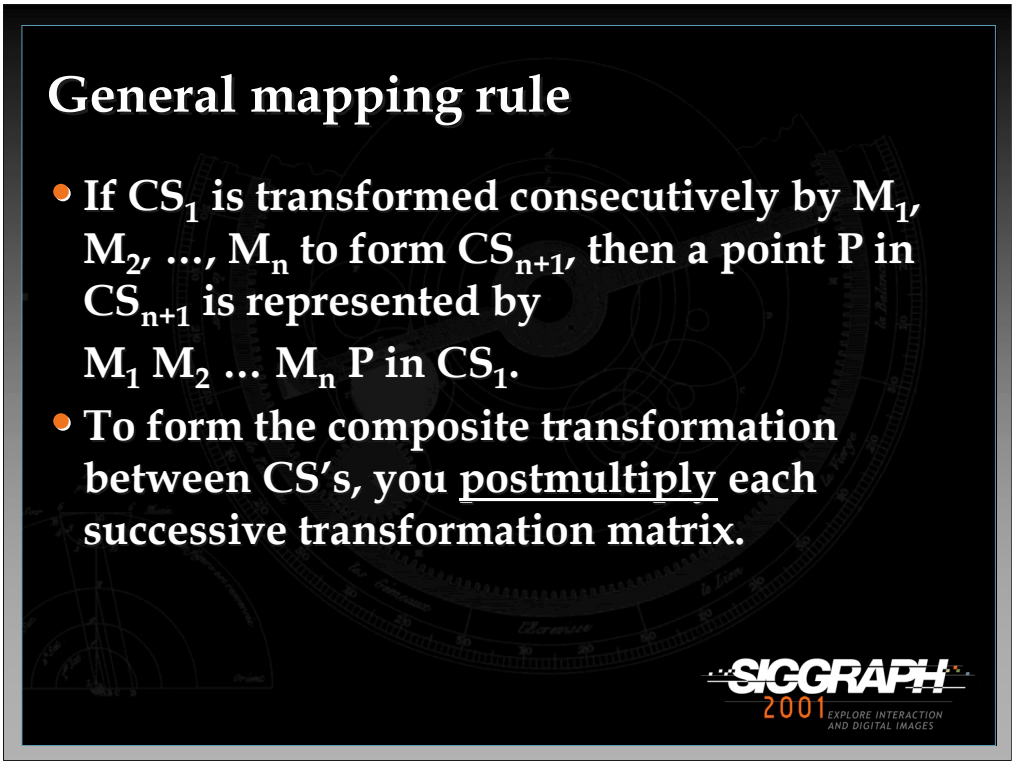
This problem is relatively trivial. If the corner of the house is at (2,2) in CS_2, and the mapping between CS_1 and CS_2 is a translation by (4,4), then the corner of the house is obviously at (6,6) in CS_1.

Therefore, we can see that if M is the mapping between CS_1 and CS_2, then $P(CS_1) = MP(CS_2)$.



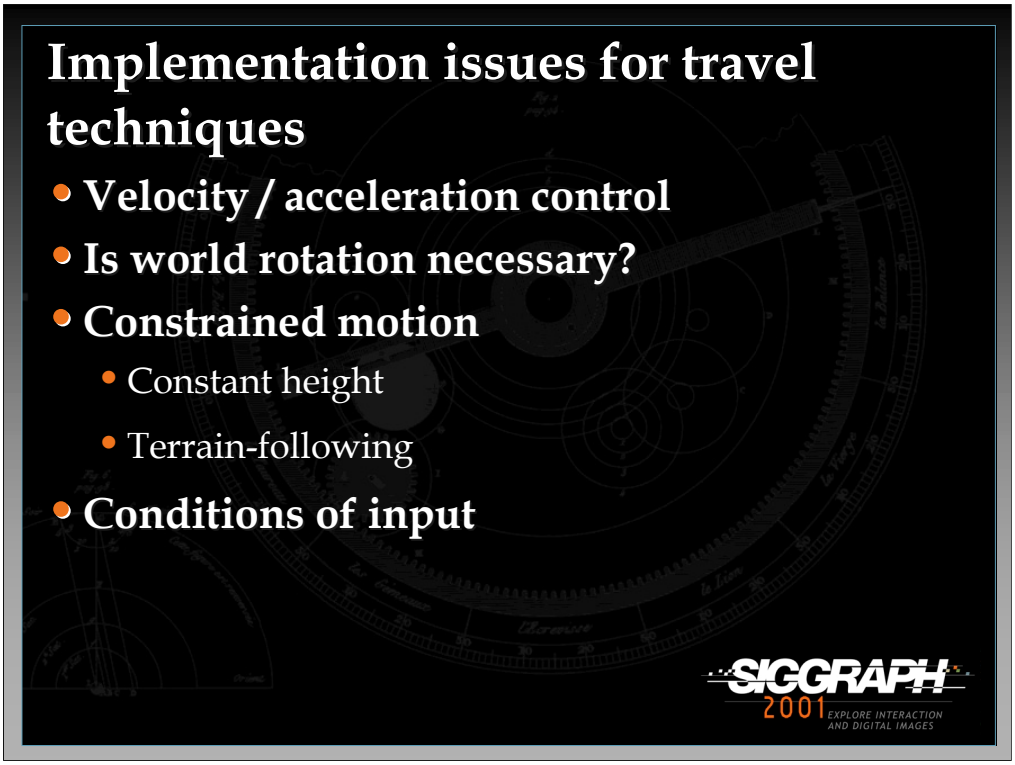
However, the mapping between coordinate systems may be more complicated than a single transformation. This is especially true when using scene graphs, where there may be many objects (and therefore many transformations) between the world coordinate system and the local coordinate system of an object. So, we need a general rule.

We see here that if we do two transformations to go from CS_1 to CS_3, then we can find a point's location in CS_1 by first multiplying the transformation from CS_1 to CS_2 by the transformation from CS_2 to CS_3 (in that order – remember that matrix multiplication is not commutative), then multiplying the result by the point's location in CS_3.



Therefore, the general rule is that we can form a composite transformation matrix between coordinate systems by postmultiplying each successive transformation matrix. (This is the opposite of the procedure if you are transforming the vertices themselves, instead of the CS). The openGL graphics package implicitly encourages this type of transformation through its use of transformation matrix stacks.

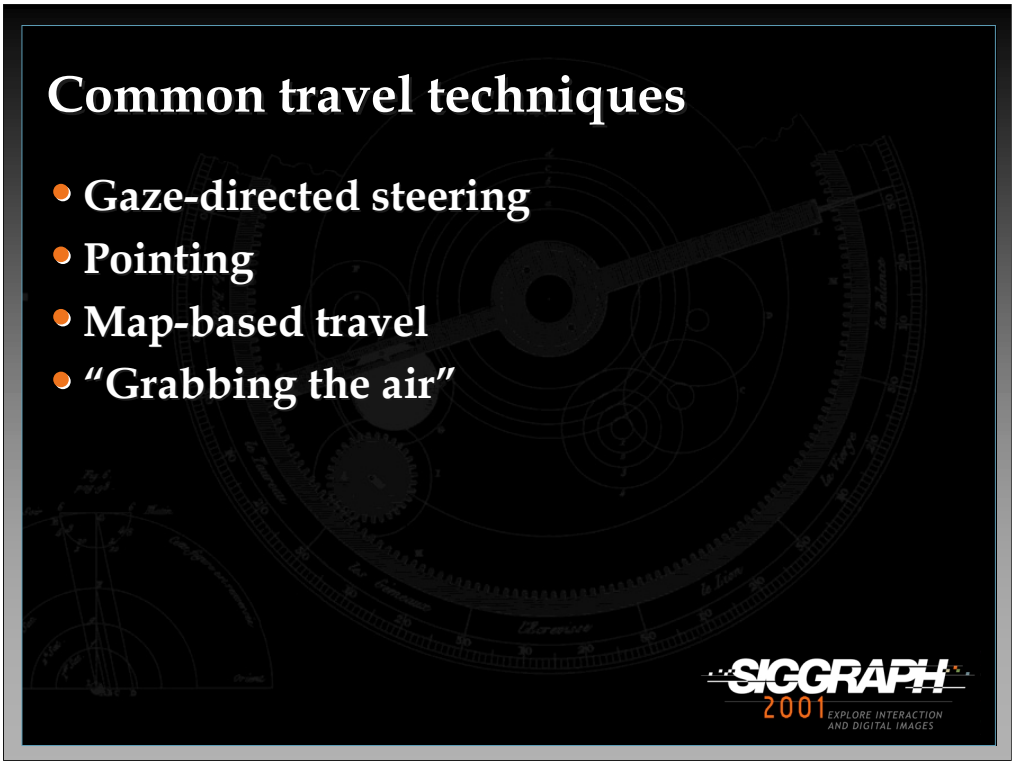
Remember, this general rule can be applied whenever a technique requires you to do a transformation between coordinate systems.



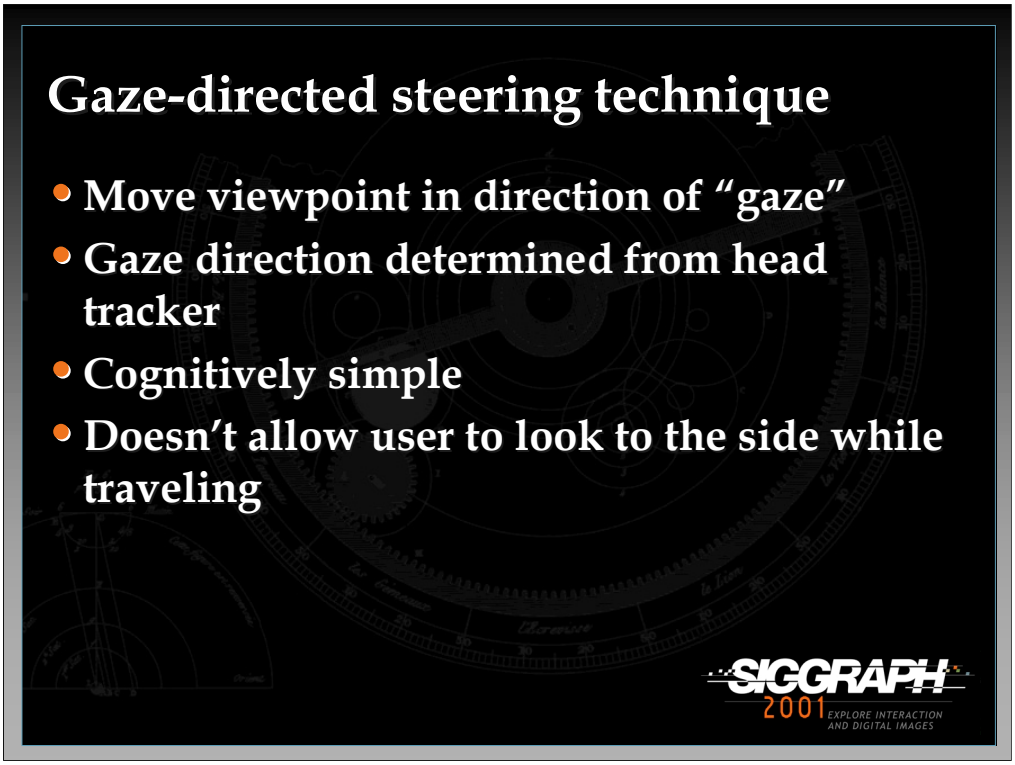
First, we'll talk about techniques for travel (viewpoint movement). We will cover only the implementation of the simple movement itself, but there are other implementation issues that must be considered in general.

One such issue is the control of velocity and/or acceleration. There are many methods for doing this, including gesture, speech controls, sliders, etc.

Another issue is that of world rotation. In systems that are only partially spatially surrounding (e.g. a 4-walled CAVE, or a single screen), the user must be able to rotate the world or his view of the world in order to navigate. In fully surrounding systems (e.g. an HMD) this is not necessary. Next, one must consider whether motion should be constrained in any way, for example by maintaining a constant height or by following the terrain. Finally, at the lowest-level the conditions of input must be considered – that is, when and how does motion begin and end (click to start/stop, press to start, release to stop, stop automatically at target location, etc.)?

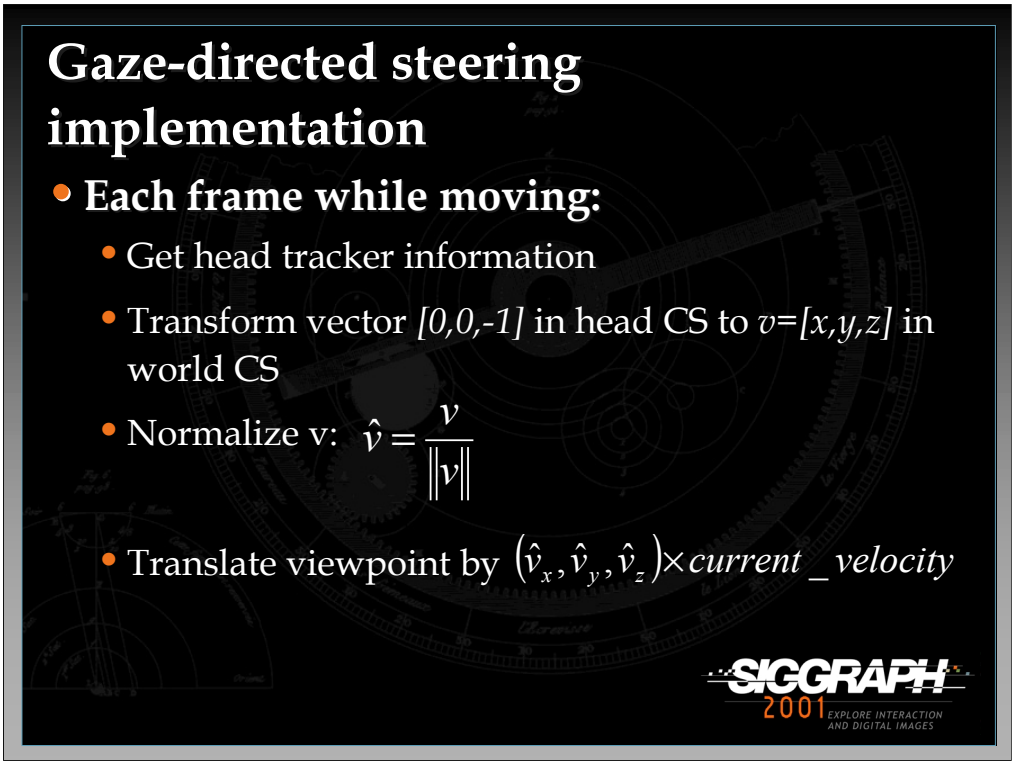


We'll discuss four common techniques.



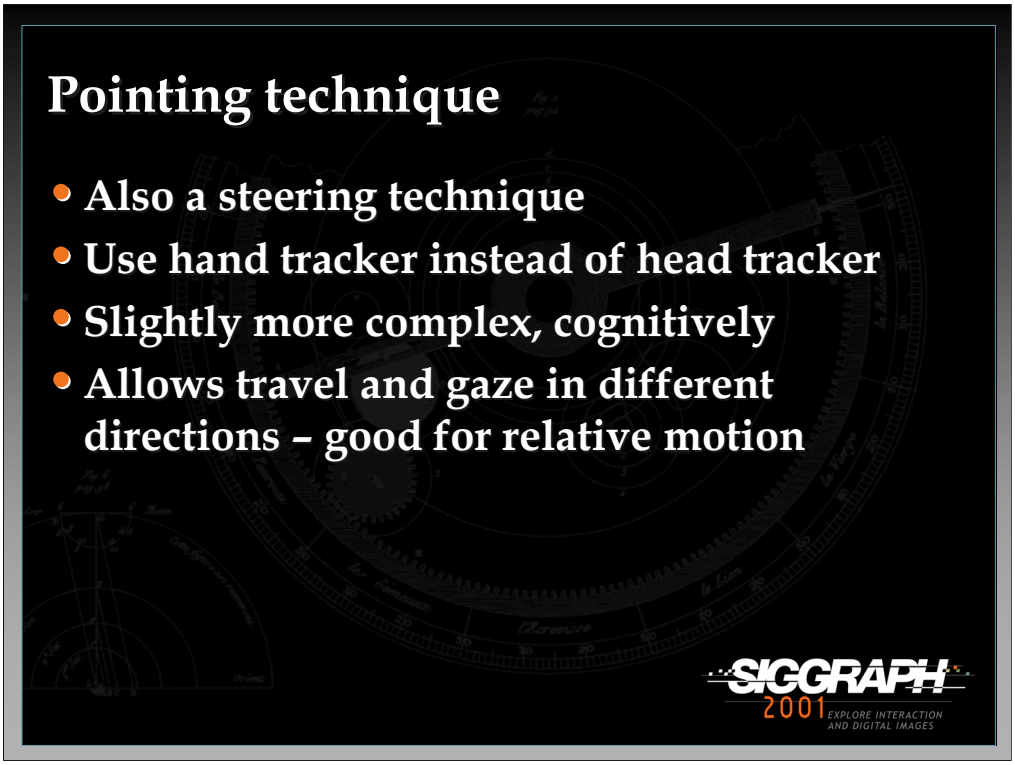
Gaze-directed steering is probably the most common 3D travel technique, although the term "gaze" is really misleading. Usually no eye tracking is being performed, so the direction of gaze is inferred from the head tracker orientation. This is a simple technique, both to implement and to use, but it is somewhat limited in that you cannot look around while moving.

See: Mine, M. (1995). *Virtual Environment Interaction Techniques* (Technical Report TR95-018): UNC Chapel Hill CS Dept.



To implement gaze-directed steering, you set up a callback function that executes before each frame is rendered. Within this callback, you first obtain the head tracker information (usually in the form of a 4x4 matrix). This matrix gives you a transformation between the base tracker CS and the head tracker CS. By also considering the transformation between the world CS and the base tracker CS (if any), you can get the total composite transformation. Now you consider the vector [0,0,-1] in head tracker space (the negative z-axis, which usually points out the front of the tracker). This vector, expressed in world coordinates, is the direction you want to move. Now all that's left to do is to normalize this vector, multiply it by the speed, and then translate the viewpoint by this amount in world coordinates.

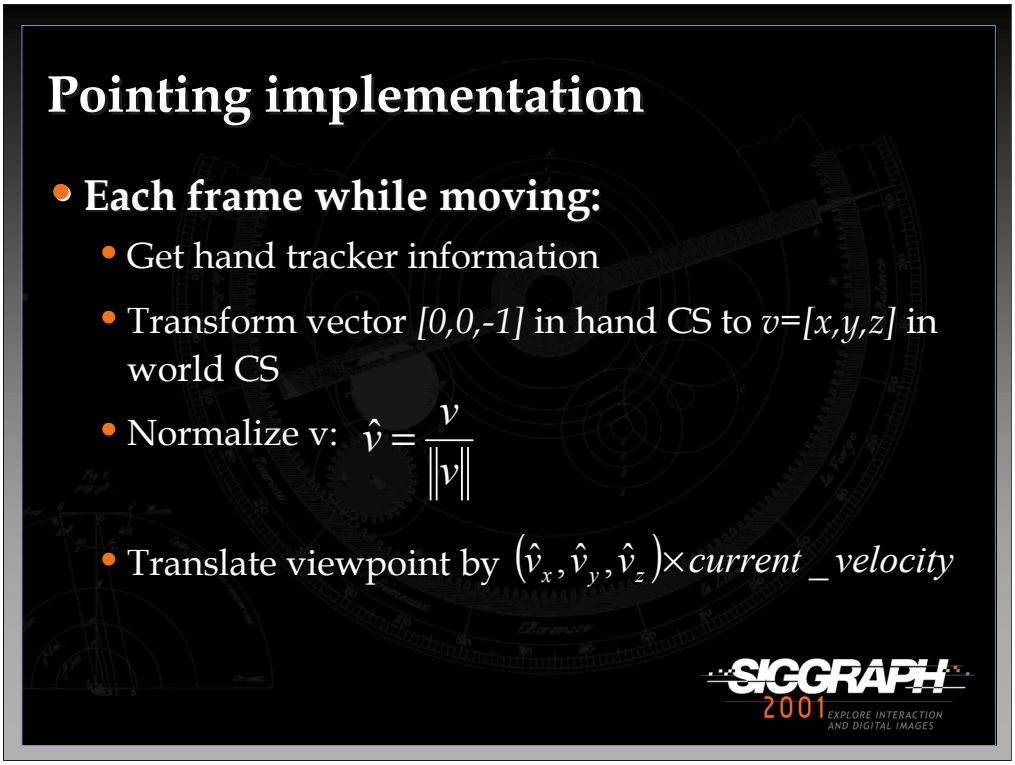
Note: current "velocity" is in units/frame. If you want true velocity (units/second), you must keep track of the time between frames and then translate the viewpoint by an amount proportional to that time.



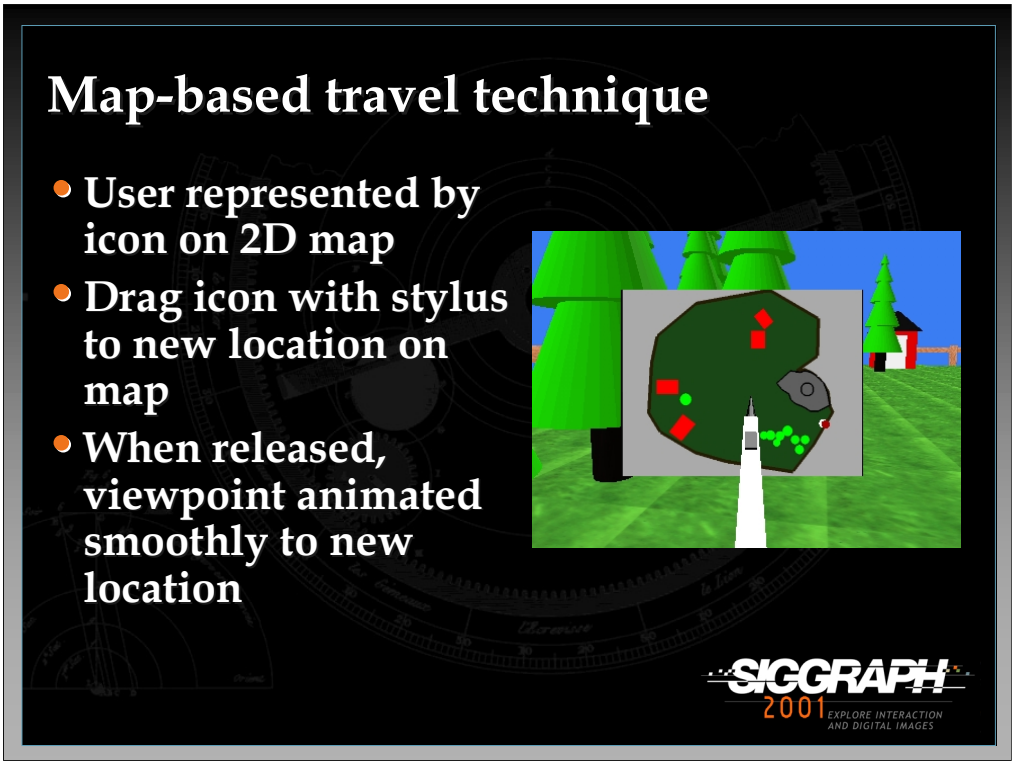
Pointing is also a steering technique (where the user continuously specifies the direction of motion). In this case, the hand's orientation is used to determine direction. This technique is somewhat harder to learn for some users, but is more flexible than gaze-directed steering.

See: Mine, M. (1995). *Virtual Environment Interaction Techniques* (Technical Report TR95-018): UNC Chapel Hill CS Dept., and

Bowman, D. A., Koller, D., & Hodges, L. F. (1997). *Travel in Immersive Virtual Environments: an Evaluation of Viewpoint Motion Control Techniques.* Proceedings of the Virtual Reality Annual International Symposium, 45-52.

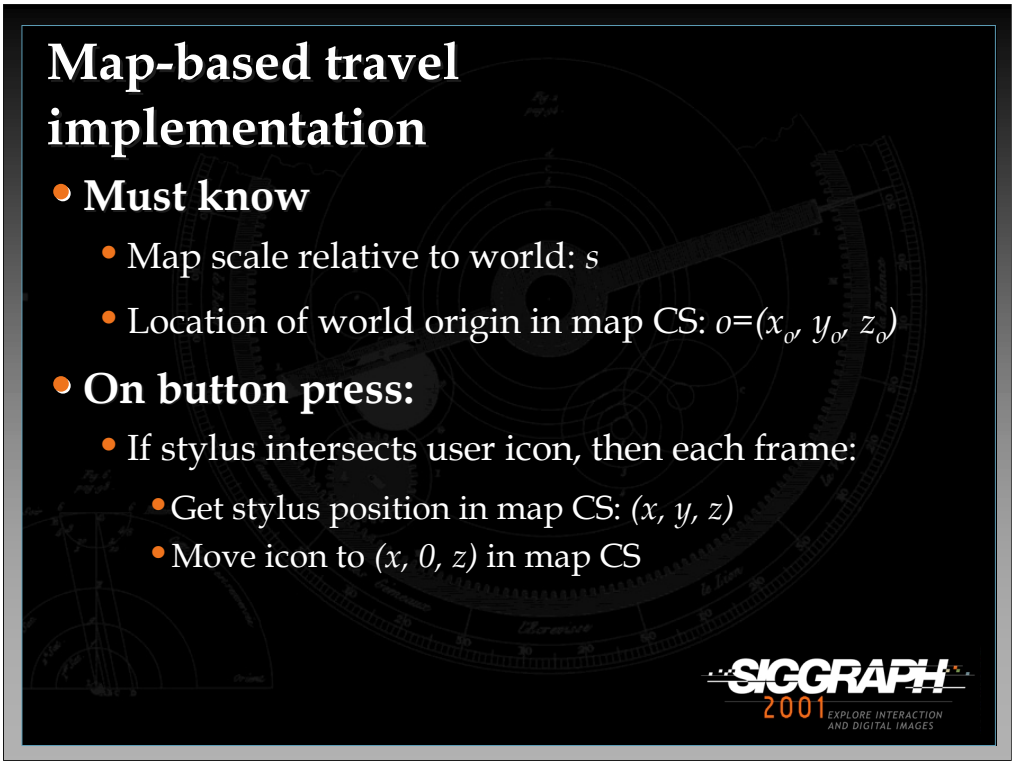


Pointing is implemented in exactly the same way as gaze-directed steering, except that the hand tracker is used instead of the head tracker.



The map-based technique is a target-based travel technique. The user is represented as an icon on a 2D map of the environment. To travel, the user drags this icon to a new position on the map. When the icon is dropped, the system smoothly animates the user from the current location to the new location indicated by the icon.

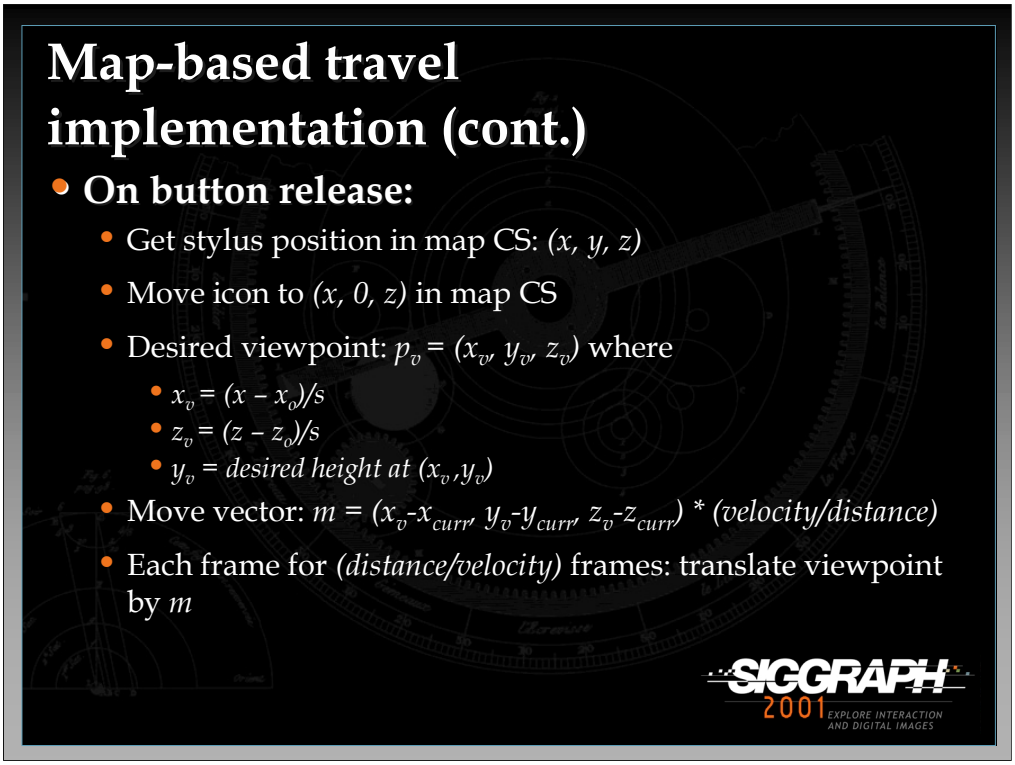
See: Bowman, D., Wineman, J., Hodges, L., & Allison, D. (1998). Designing Animal Habitats Within an Immersive VE. *IEEE Computer Graphics & Applications, 18*(5), 9-13.



To implement this technique, you need to know two things about the way the map relates to the world. First, you need to know the scale factor. Second, you need to know which point on the map represents the origin of the world CS. We assume here that the map model is originally aligned with the world (i.e. the x direction on the map, in its local CS, represents the x direction in the world CS).

When the user presses the button and is intersecting the user icon on the map, then you need to move the icon with the stylus each frame. You cannot simply attach the icon to the stylus, because you want the icon to remain on the map even if the stylus does not. To do this, you must first find the position of the stylus **in the map CS.** This may require a transformation between coordinate systems, since the stylus is not a child of the map. The x and z coordinates of the stylus position are the point to which the icon should be moved.

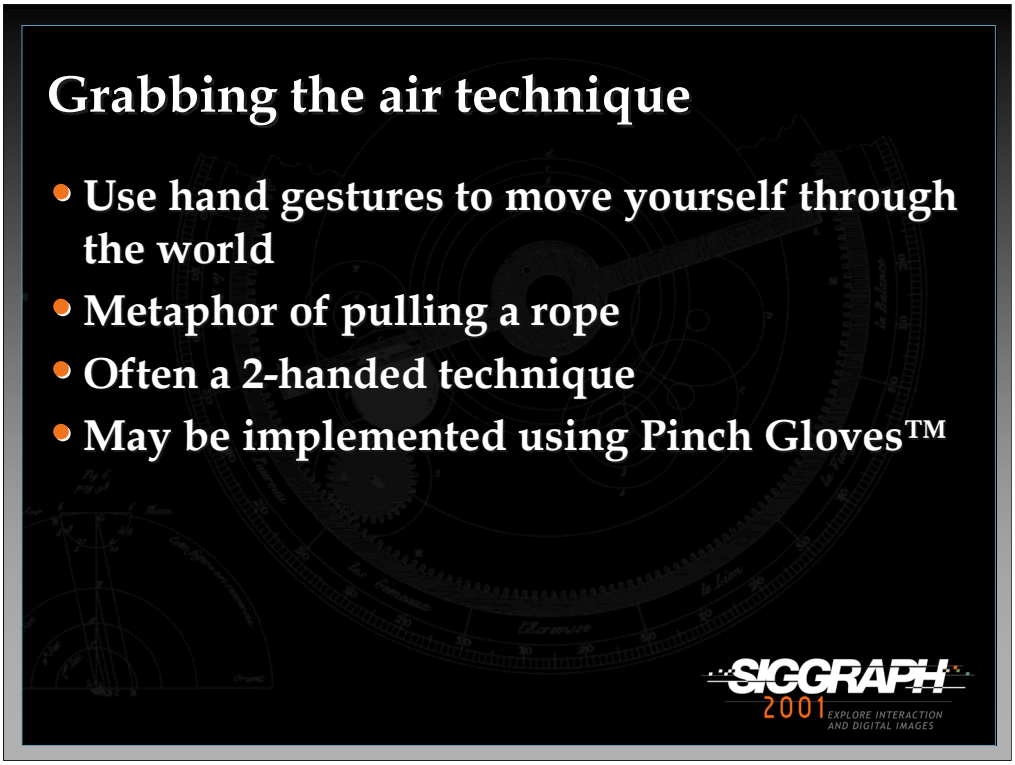
We do not cover here what happens if the stylus is dragged off the map, but the user icon should "stick" to the side of the map until the stylus is moved back inside the map boundaries, since we don't want the user to move outside the world.



When the button is released, the icon is set to its final position, using the same method as before.

Now we need to calculate the desired position of the viewpoint in the world. This position is calculated using a transformation from the map CS to the world CS, which is detailed for you here. First, you must find the offset in the map CS from the point corresponding to the world origin. Then, you divide by the map scale (if the map is 1/100 the size of the world, this corresponds to multiplying by 100). This gives us the x and z coordinates of the desired viewpoint position. Since the map is 2D, we can't get a y coordinate from it. Therefore, the technique should have some way of calculating the desired height at the new viewpoint. In the simplest case, this might be constant. In other cases, it might be based on the terrain height at that location or some other factors.

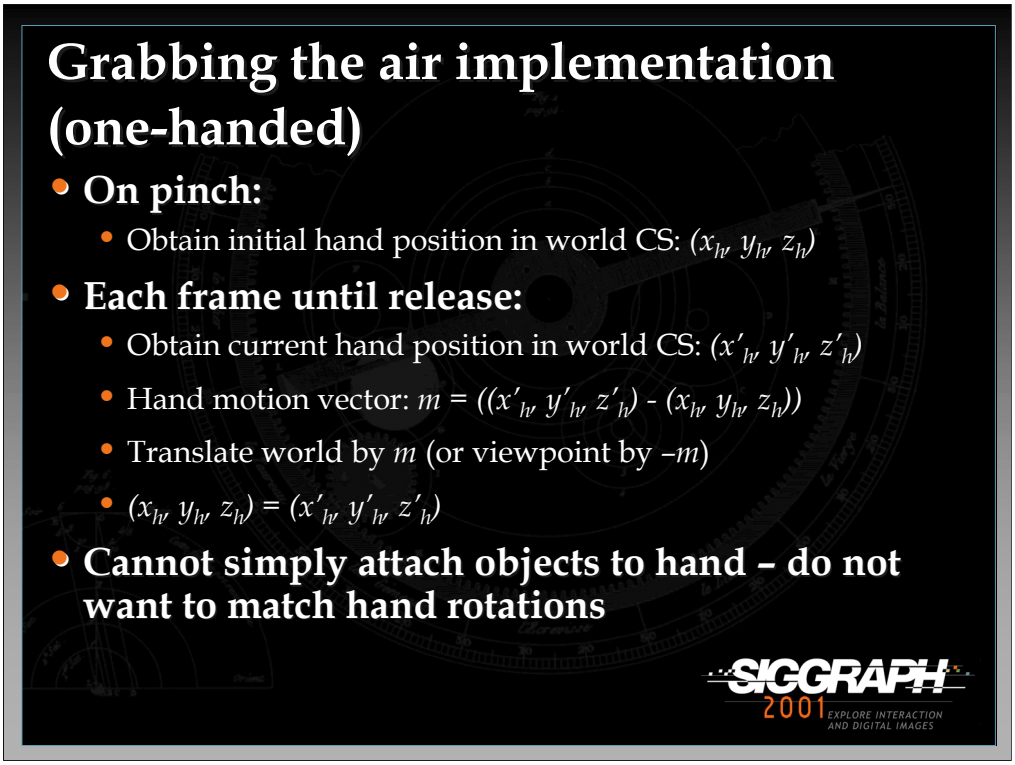
Now that we know the desired viewpoint, we have to set up the animation of the viewpoint. The move vector *m* represents the amount of translation to do each frame (we are assuming a linear path). To find *m*, we subtract the desired position from the current position (the total movement required), divide this by the distance between the two points (calculated using the distance formula), and multiply by the desired velocity, so that *m* gives us the amount to move in each dimension each frame. The only remaining calculation is the number of frames this movement will take: distance/velocity frames. Note that again velocity is measured here in units/frame, not units/second, for simplicity.



The "grabbing the air" technique uses the metaphor of literally grabbing the world around you (usually empty space), and pulling yourself through it using hand gestures. This is similar to pulling yourself along a rope, except that the "rope" exists everywhere, and can take you in any direction.

This technique may be done with one or two hands, and is often implemented using Pinch Gloves™.

See: Mapes, D., & Moshell, J. (1995). A Two-Handed Interface for Object Manipulation in Virtual Environments. *Presence: Teleoperators and Virtual Environments, 4*(4), 403-416.



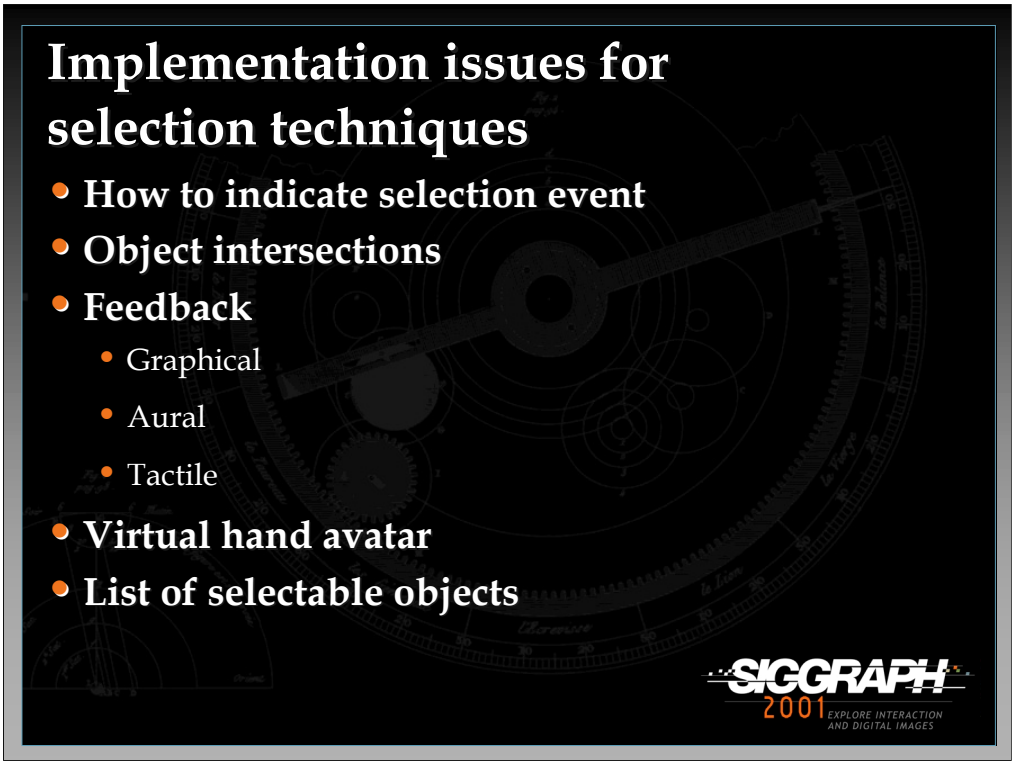
We'll describe a simple one-handed technique. This could be extended to two hands by running this algorithm for each hand separately, making sure that only one hand is activated at a time.

When the initial pinch or button press is detected, simply obtain the position of the hand in the world CS.

Then, every frame until the pinch is released, get a new hand position, subtract it from the old one, and move the objects in the world by this amount. Alternately, you can leave the world fixed, and translate the viewpoint by the opposite vector. Before exiting the callback, be sure to update the "old" hand position for use on the next frame.

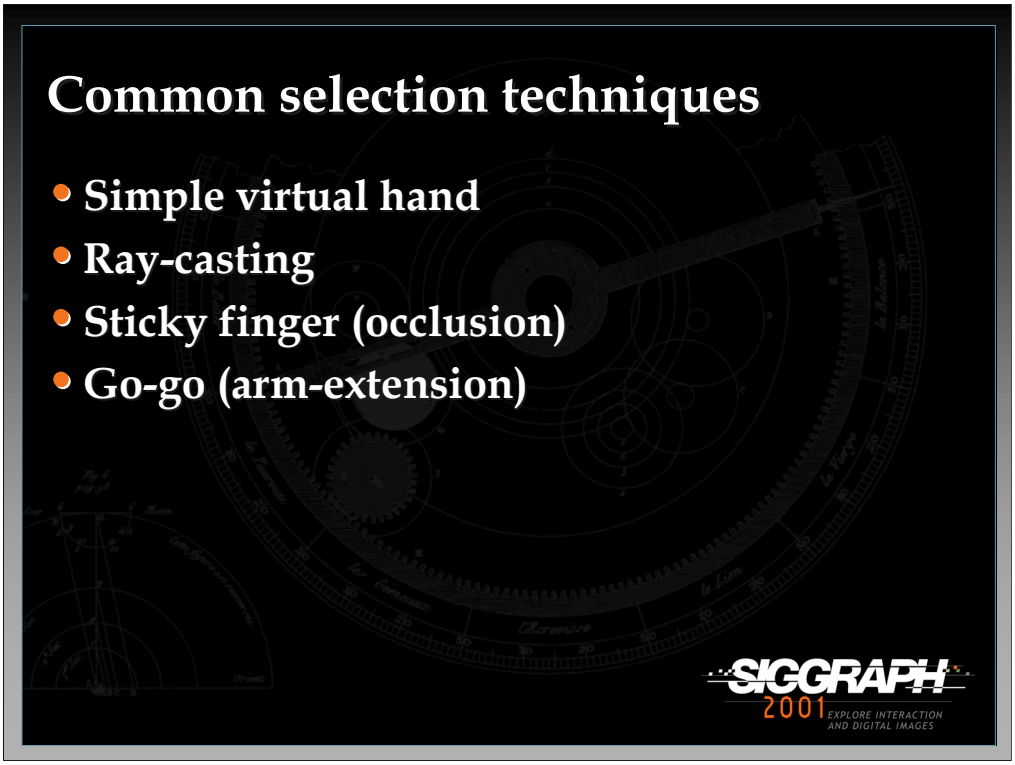
It is tempting to implement this technique simply by attaching the world to the hand, but this will have the undesirable effect of also rotating the world when the hand rotates, which can be quite disorienting.

You can do simple constrained motion simply by ignoring one or more of the components of the hand position (e.g. only consider x and z to move at a constant height).

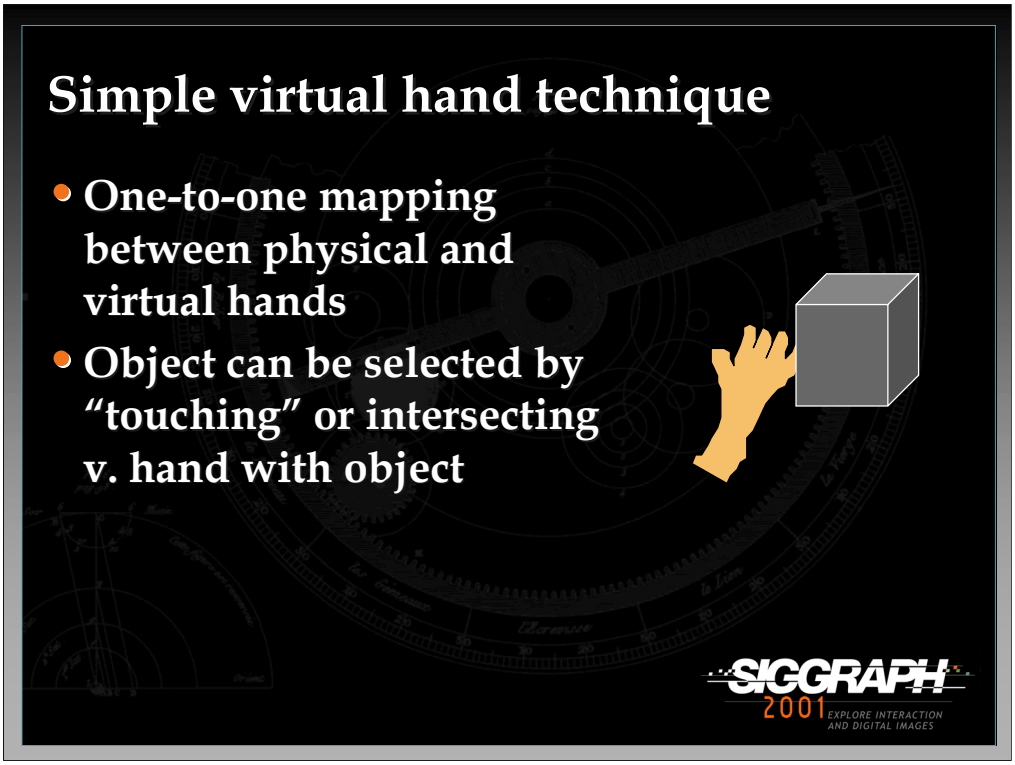


Next, we'll discuss object selection. We must first note that selection and manipulation are intimately related, and that several of the techniques described here can also be used for manipulation.

There are several common issues for the implementation of selection techniques. One of the most basic is how to indicate that the selection event should take place (e.g. you are touching the desired object, now you want to pick it up). This is usually done via a button press, gesture, or voice command, but it might also be done automatically if the system can infer the user's intent. You also have to have efficient algorithms for object intersections for many of these techniques. We'll discuss a couple of possibilities. The feedback you give to the user regarding which object is about to be selected is also very important. Many of the techniques require an avatar (virtual representation) for the user's hand. Finally, you should consider keeping a list of objects that are "selectable", so that your techniques do not have to test every object in the world for selection, increasing efficiency.

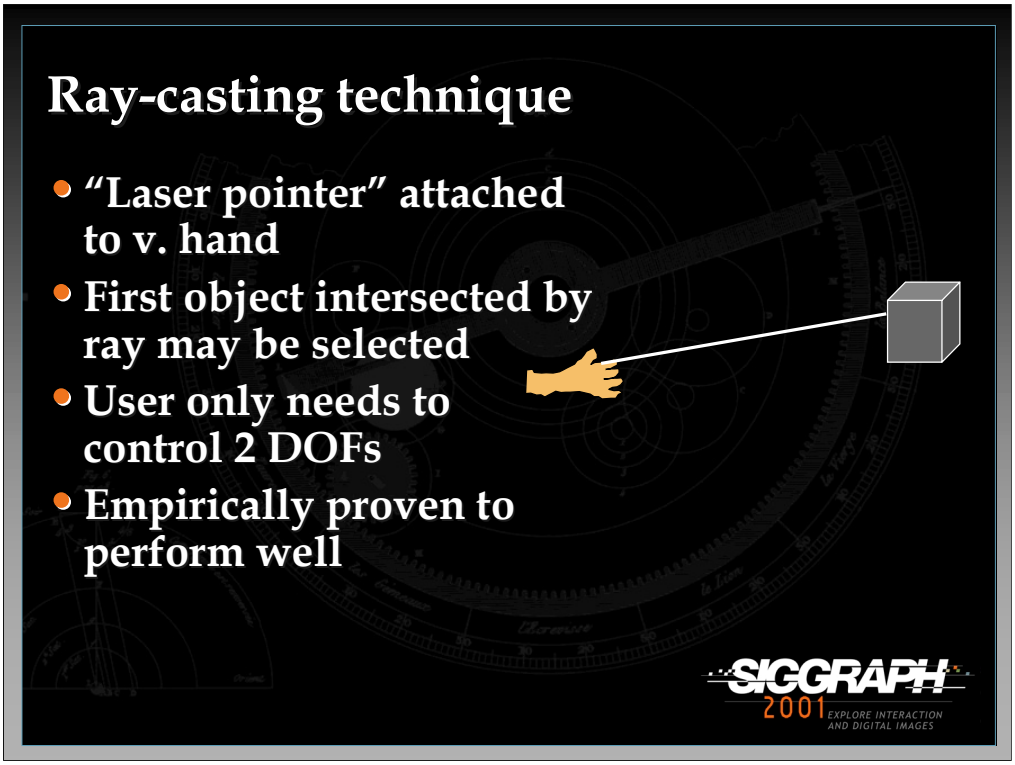


We'll discuss four selection techniques.



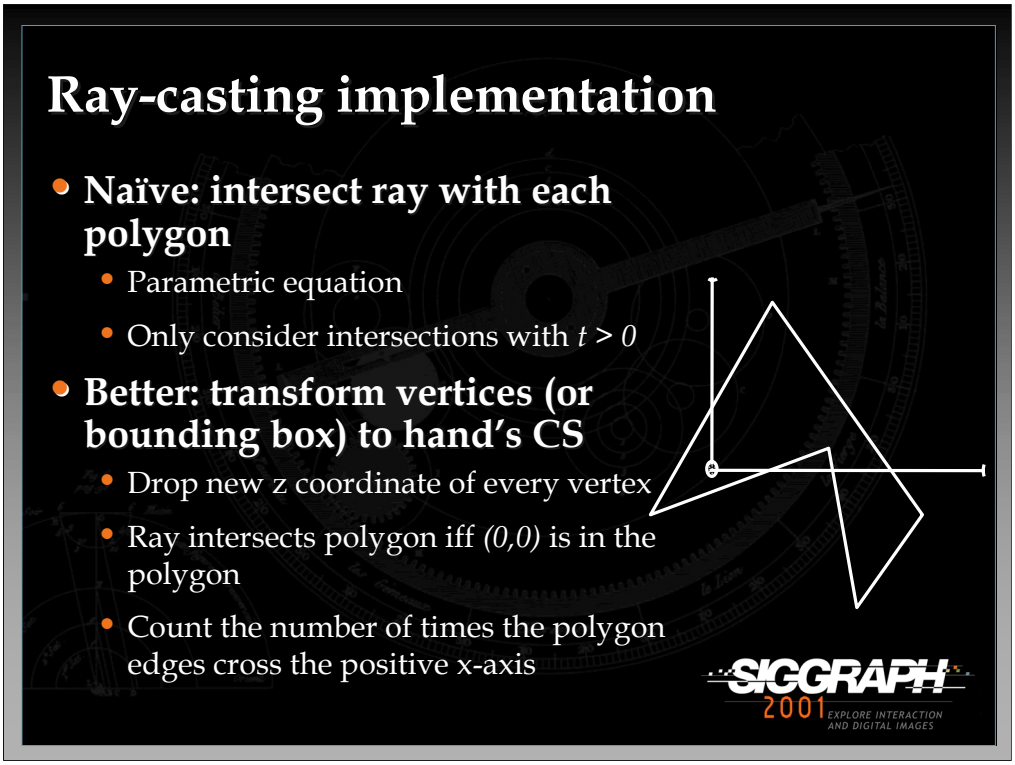
The most common technique is the simple virtual hand, which does "real-world" selection via direct "touching" of virtual objects. In the absence of haptic feedback, this is done by intersecting the virtual hand (which is at the same location as the physical hand) with a virtual object.

Implementing this technique is simple, provided you have a good intersection/collision algorithm. Often, intersections are only performed with axis-aligned bounding boxes or bounding spheres rather than with the actual geometry of the objects.



Another common technique is ray-casting. This technique uses the metaphor of a laser pointer – an infinite ray extending from the virtual hand. The first object intersected along the ray is eligible for selection. This technique is efficient, based on experimental results, and only requires the user to vary 2 degrees of freedom (pitch and yaw of the wrist) rather than the 3 DOFs required by the simple virtual hand and other location-based techniques.

See: Mine, M. (1995). *Virtual Environment Interaction Techniques* (Technical Report TR95-018): UNC Chapel Hill CS Dept.



There are many ways to implement ray-casting.

A brute-force approach would calculate the parametric equation of the ray, based on the hand's position and orientation. First, as in the pointing technique for travel, find the world CS equivalent of the vector *[0,0,-1]*. This is the direction of the ray. If the hand's position is represented by (x_h, y_h, z_h), and the direction vector is (x_d, y_d, z_d), then the parametric equations are given by:

$$x(t) = x_h + x_d t$$
$$y(t) = y_h + y_d t$$
$$z(t) = z_h + z_d t$$

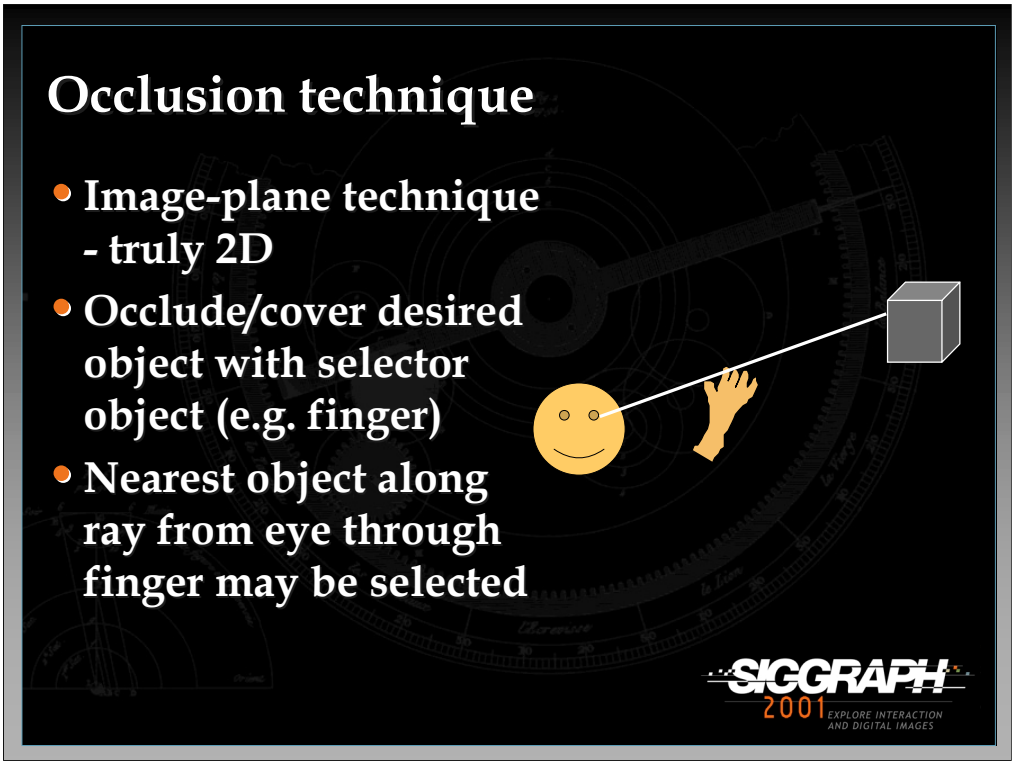
Only intersections with t>0 should be considered, since we don't want to count intersections "behind" the hand. If you use this method and you want to determine whether the actual geometry has been intersected, you should first test the intersection with the bounding box so that many cases can be trivially rejected.

-continued on the next page

Another method might be more efficient. In this method, instead of looking at the hand orientation in the world CS, we consider the selectable objects to be in the hand's CS, by transforming their vertices or their bounding boxes. This might seem quite inefficient, because there is only one hand, while there are many polygons in the world. However, we assume we have limited the objects by using a selectable objects list, and the intersection test we will describe is much more efficient.

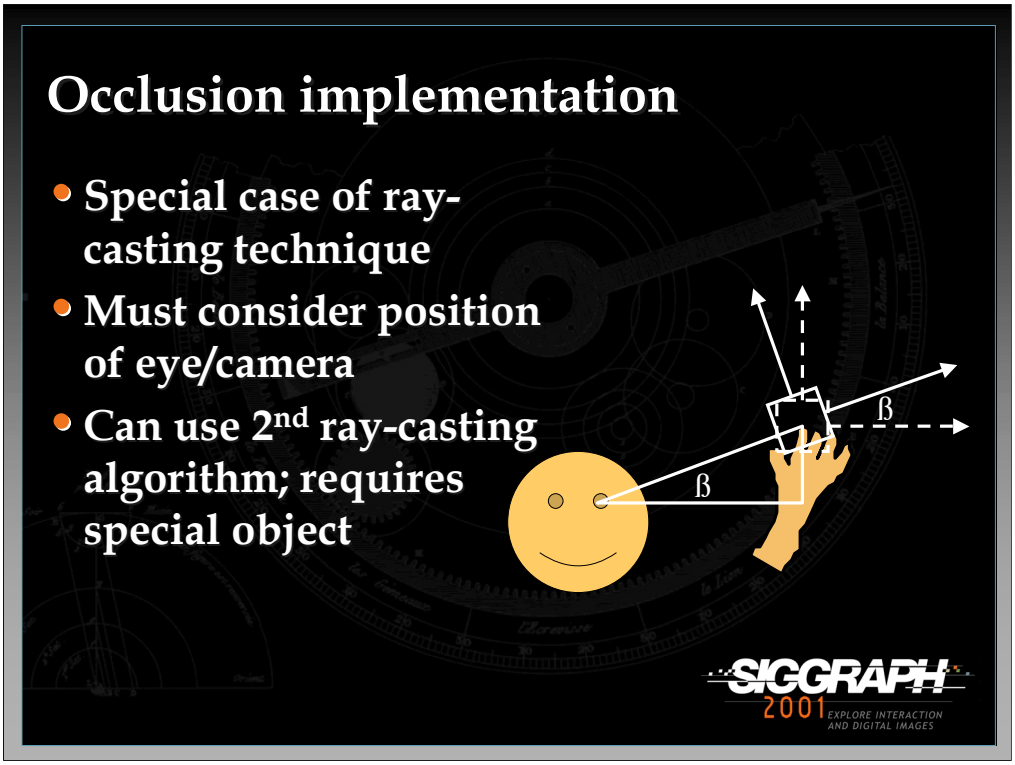
Once we have transformed the vertices or bounding boxes, we drop the z coordinate of each vertex. This maps the 3D polygon onto a 2D plane (the xy plane in the hand CS). Since the ray is *[0,0,-1]* in this CS, we can see that in this 2D plane, the ray will intersect the polygon if and only if the point *(0,0)* is in the polygon (see the figure). We can easily determine this with an algorithm that counts the number of times the edges of the 2D polygon cross the positive x-axis. If there are an odd number of crossings, the origin is inside, if even, the origin is outside.

Not discussed on this slide is a third method of implementation. The openGL graphics package includes a "selection" concept. By rendering the selectable objects to an offscreen buffer from the point of view of the hand, you can determine which objects are intersected by the ray, and which one is closest.



Next, we'll cover the occlusion technique (also called the "sticky finger" technique). This technique works in the plane of the image – that is, you select an object by "covering" it with the virtual hand so that it is occluded from your point of view. Geometrically, this means that a ray is emanating from your eye, going through your finger, and then intersecting an object.

See: Pierce, J., Forsberg, A., Conway, M., Hong, S., Zeleznik, R., & Mine, M. (1997). *Image Plane Interaction Techniques in 3D Immersive Environments*. Proceedings of the ACM Symposium on Interactive 3D Graphics, 39-44.

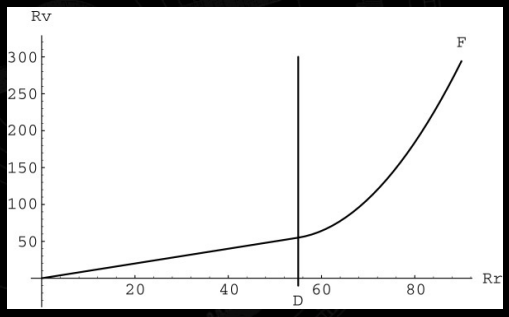


This technique can be implemented in the same ways as the ray-casting technique, since it is also using a ray. If you are doing the brute-force ray intersection algorithm, you can simply define the ray's direction by subtracting the finger position from the eye position.

However, if you are using the 2nd algorithm, you require an object to define the ray's coordinate system. This can be done in two steps. First, create an empty object, and place it at the hand position, aligned with the world CS (the dotted lines in the figure). Next, determine how to rotate this object/CS so that it is aligned with the ray direction. In the figure, a 2D example is given. The angle can be determined using the positions of the eye and hand, and some simple trigonometry. In 3D, two rotations must be done in general to align the new object's CS with the ray.

Go-Go technique

- Arm-extension technique
- Like simple v. hand, touch objects to select them
- Non-linear mapping between physical and virtual hand position
- Local and distant regions

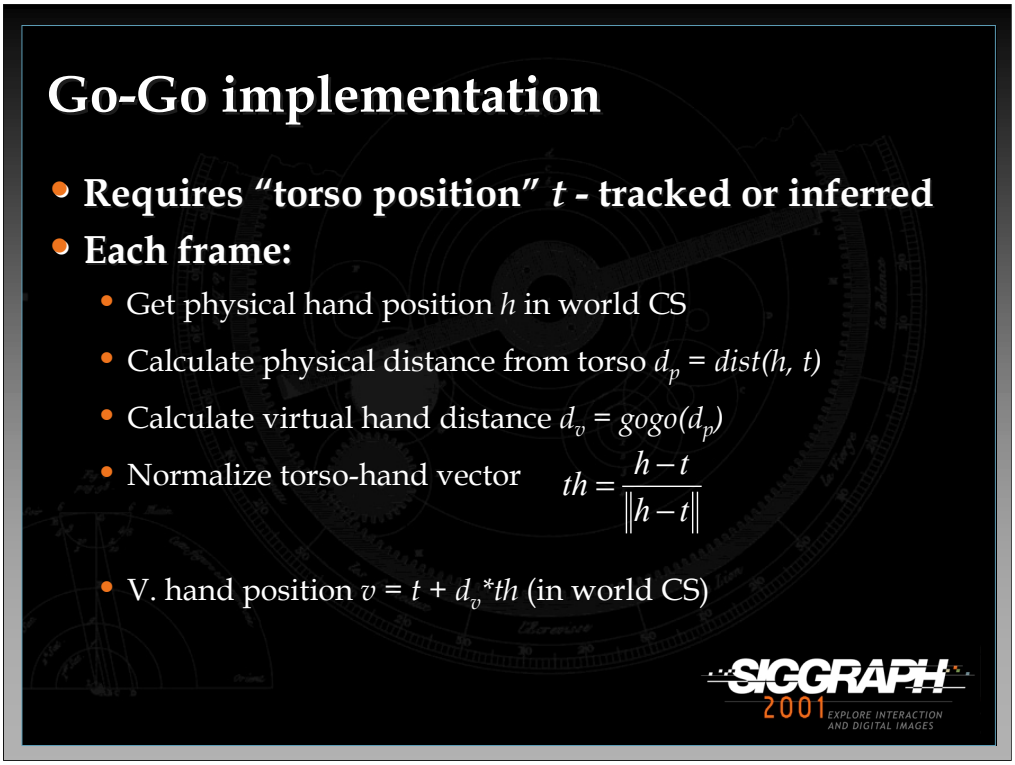


SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

The Go-Go technique is based on the simple virtual hand, but it introduces a non-one-to-one mapping between the physical hand and the virtual hand, so that the user's reach is greatly extended. This is called an arm-extension technique.

The graph shows the mapping between the physical hand distance from the body on the x-axis and the virtual hand distance from the body on the y-axis. There are two regions. When the physical hand is at a depth less than a threshold 'D', the one-to-one mapping applies. Outside D, a non-linear mapping is applied, so that the farther the user stretches, the faster the virtual hand moves away.

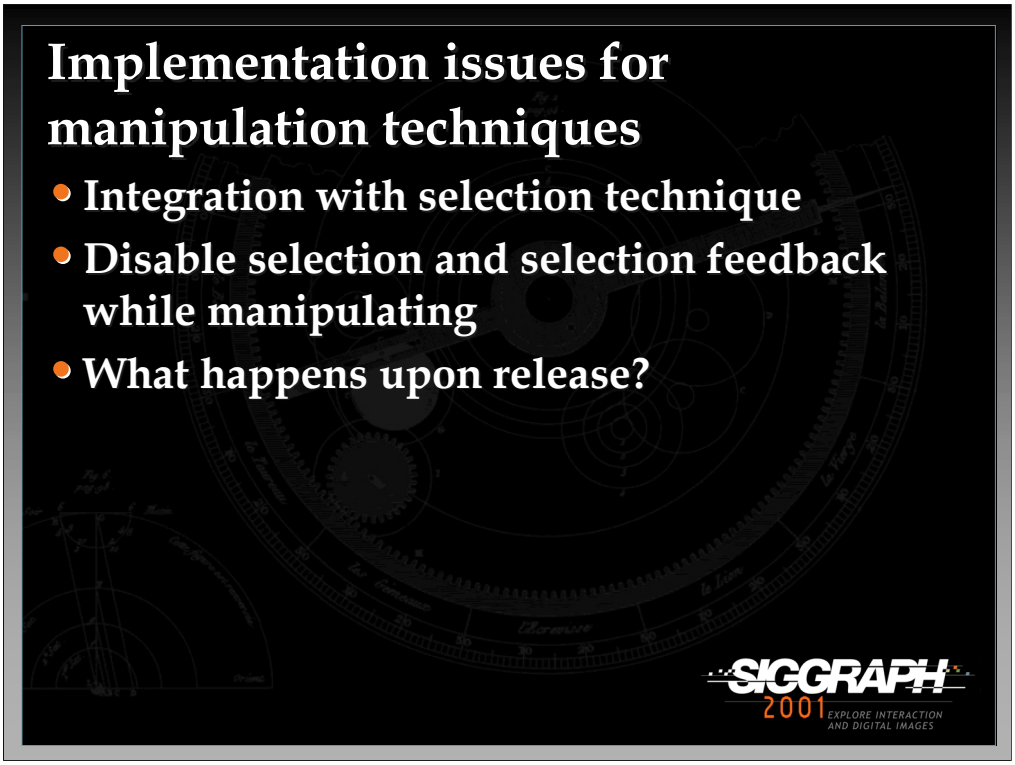
See: Poupyrev, I., Billinghurst, M., Weghorst, S., & Ichikawa, T. (1996). *The Go-Go Interaction Technique: Non-linear Mapping for Direct Manipulation in VR.* Proceedings of the ACM Symposium on User Interface Software and Technology, 79-80.



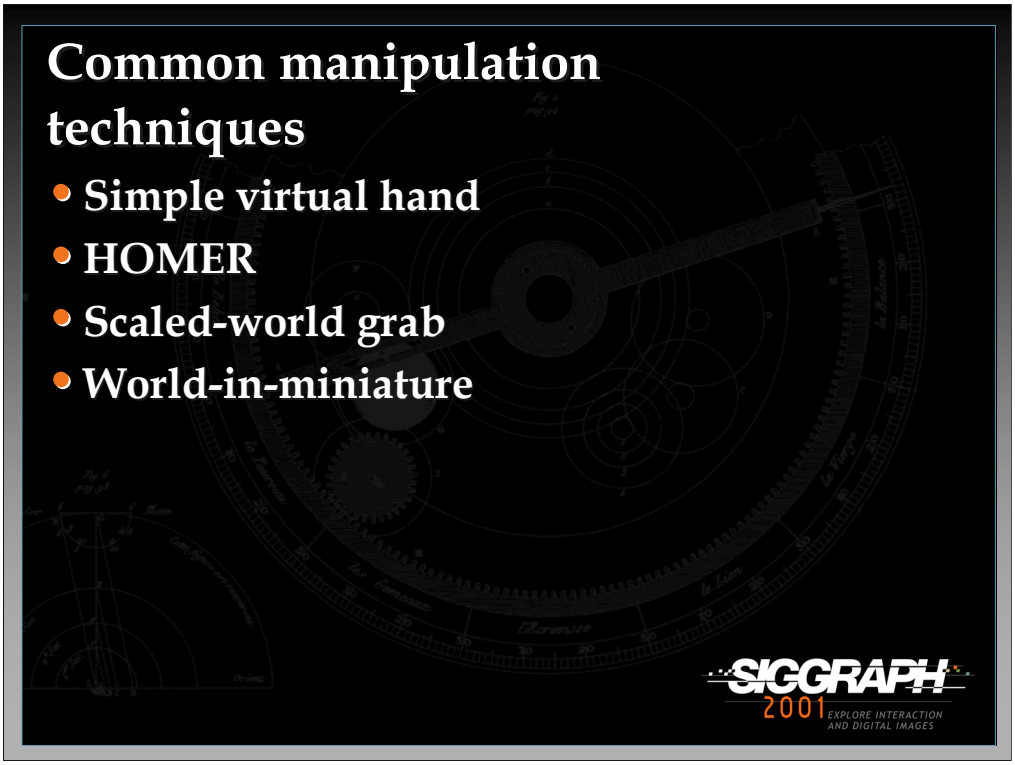
To implement Go-Go, we first need the concept of the position of the user's body. This is needed because we stretch our hands out from the center of our body, not from our head (which is usually the position that is tracked). I have implemented this using an inferred torso position, which is defined as a constant offset in the negative y direction from the head. You could also place a tracker on the user's torso.

Before rendering each frame, you get the physical hand position in the world CS, and then calculate its distance from the torso object using the distance formula. The virtual hand distance can then be obtained by applying the function shown in the graph on the previous slide. I have used the function $d^{2.3}$ (starting at the point (D,D)) as a useful function in my environments, but the exponent used depends on the size of the environment and the desired accuracy of selection at a distance.

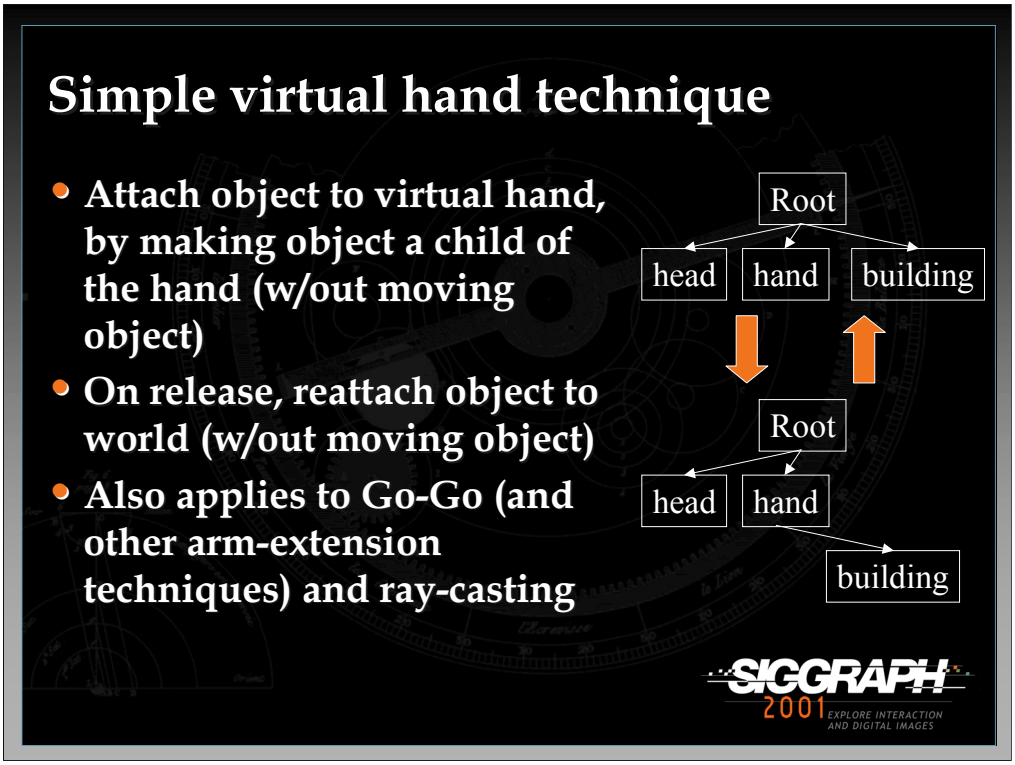
Now that we know the distance at which to place the virtual hand, we need to determine its position. The most common implementation is to keep the virtual hand on the ray extending from the torso and going through the physical hand. Therefore, if we get a vector between these two points, normalize it, multiply it by the distance, then add this vector to the torso point, we obtain the position of the virtual hand.



Next we turn to techniques for 3D object manipulation. As we noted earlier, manipulation is connected with selection, because an object must be selected before you can manipulate it. Thus, one important issue for any manipulation technique is how well it integrates with the chosen selection technique. Many techniques, as we have said, do both: e.g. simple virtual hand, ray-casting, and go-go. Another issue is that when an object is being manipulated, you should take care to disable the selection technique and the feedback you give the user for selection. If this is not done, then serious problems can occur if, for example, the user tries to release the currently selected object but the system also interprets this as trying to select a new object. Finally, you should think in general about what happens when the object is released. Does it remain at its last position, possibly floating in space? Does it snap to a grid? Does it fall via gravity until it contacts something solid? The application requirements will determine this choice.



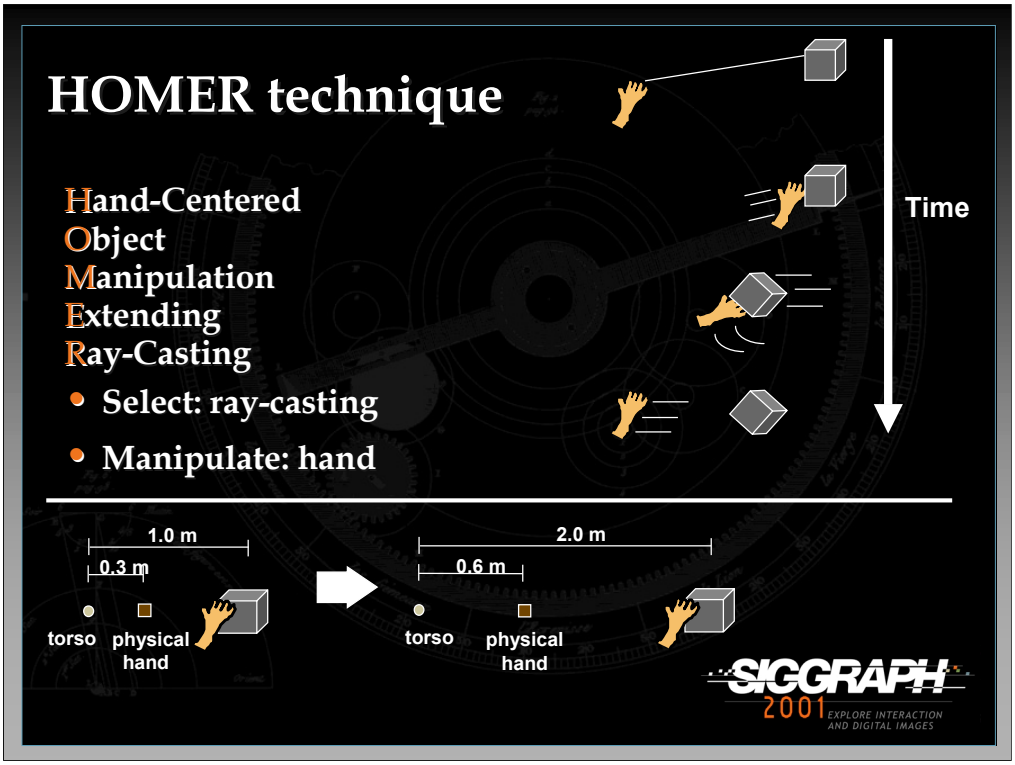
We'll discuss four 3D object manipulation techniques.



We already saw the simple virtual hand technique for selection. When this technique is used for object manipulation, the implementation is quite easy. It simply involves making a change to the scene graph by attaching the selected object to the virtual hand. Then, as the virtual hand moves and rotates, the selected object will inherit those transformations. When the object is released, it should just be reattached to its earlier location in the tree.

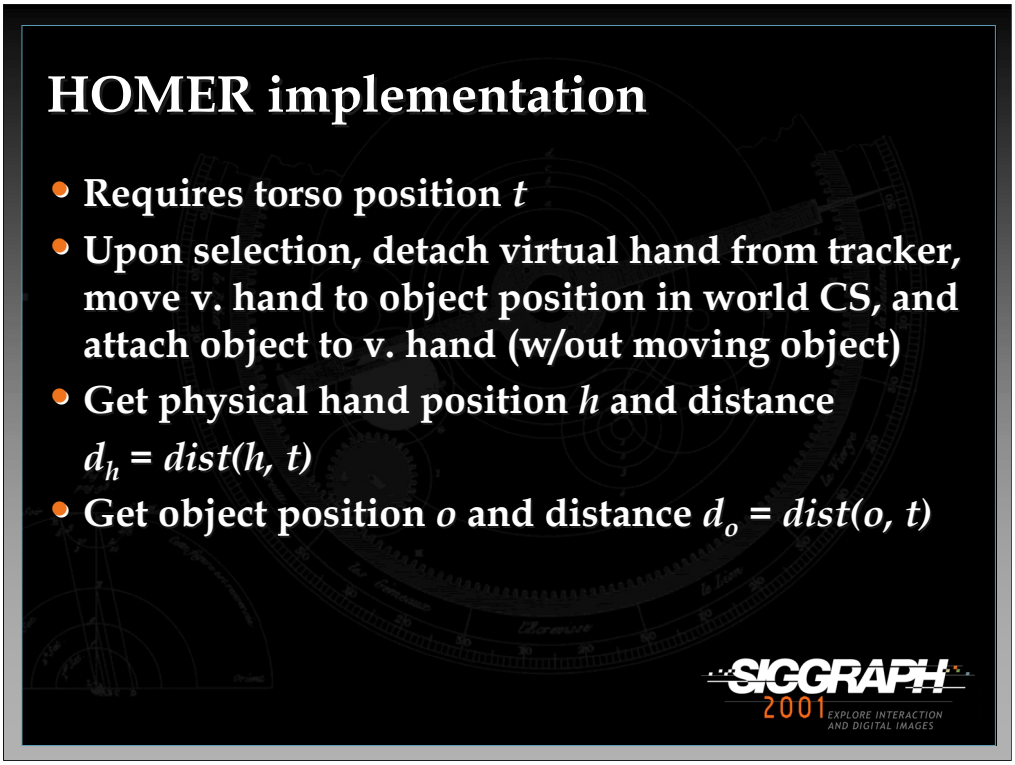
The only tricky issue here is that you must ensure when grabbing or releasing the object that it does not move (in the world CS). If you simply make the object a child of the hand, it may move since its position is now being interpreted relative to a new CS (the hand's). To be completely general, then, you must get the object's position *p* in the world CS first, then do the attachment, then calculate *p*'s location in the hand CS, then move the object to that position (relative to the hand). The opposite transformation is done upon release.

This same basic procedure works for other techniques that simply attach the object to the selector, like Go-Go and ray-casting.



The HOMER technique uses ray-casting for selection and then moves the virtual hand to the object for hand-centered manipulation. The depth of the object is based on a linear mapping, as shown in the figure at the bottom. The initial torso-physical hand distance is mapped onto the initial torso-object distance, so that moving the physical hand twice as far away also moves the object twice as far away. Also, moving the physical hand all the way back to the torso moves the object all the way to the user's torso as well.

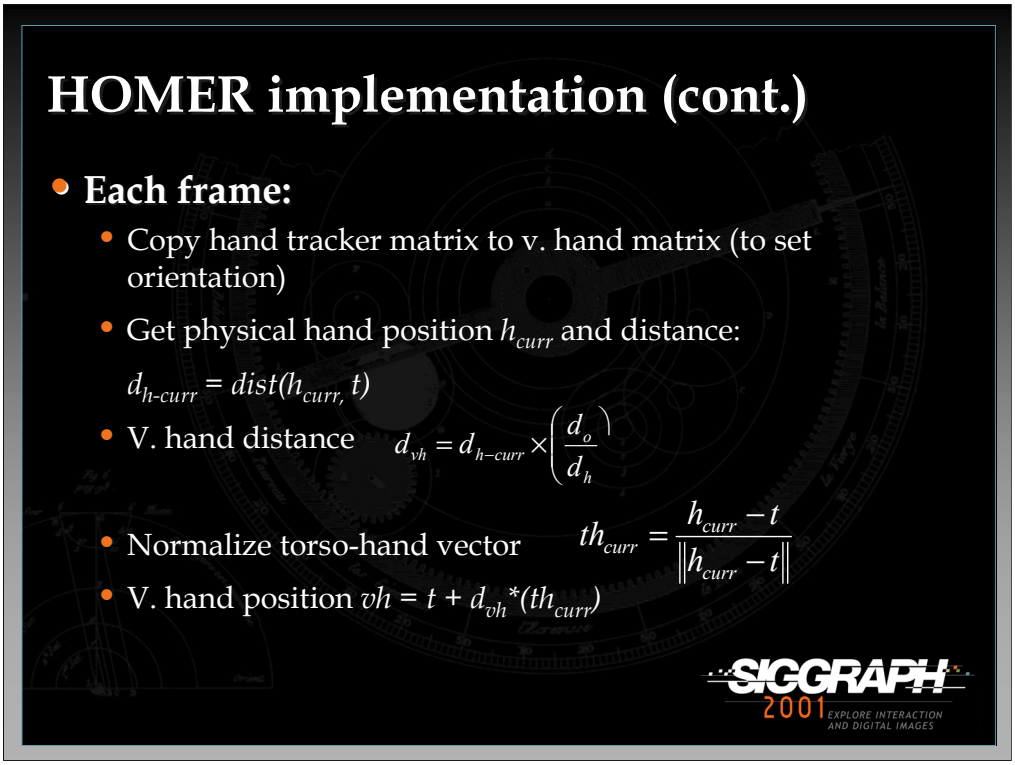
See: Bowman, D., & Hodges, L. (1997). *An Evaluation of Techniques for Grabbing and Manipulating Remote Objects in Immersive Virtual Environments.* Proceedings of the ACM Symposium on Interactive 3D Graphics, 35-38.



Like Go-Go, HOMER requires a torso position, because you want to keep the virtual hand on the ray between the user's body (torso) and the physical hand. The problem here is that HOMER moves the virtual hand from the physical hand position to the object upon selection, and it is not guaranteed that the torso, physical hand, and object will all line up at this time. Therefore, in my implementation, I calculate where the virtual hand would be if it were on this ray initially, then calculate the offset to the position of the virtual object, and maintain this offset throughout manipulation.

When an object is selected via ray-casting, you must first detach the virtual hand from the hand tracker. This is due to the fact that if it remained attached but you move the virtual hand model away from the physical hand location, a rotation of the physical hand will cause a rotation and translation of the virtual hand. You then move the virtual hand in the world CS to the position of the selected object, and attach the object to the virtual hand in the scene graph (again, without moving the object in the world CS).

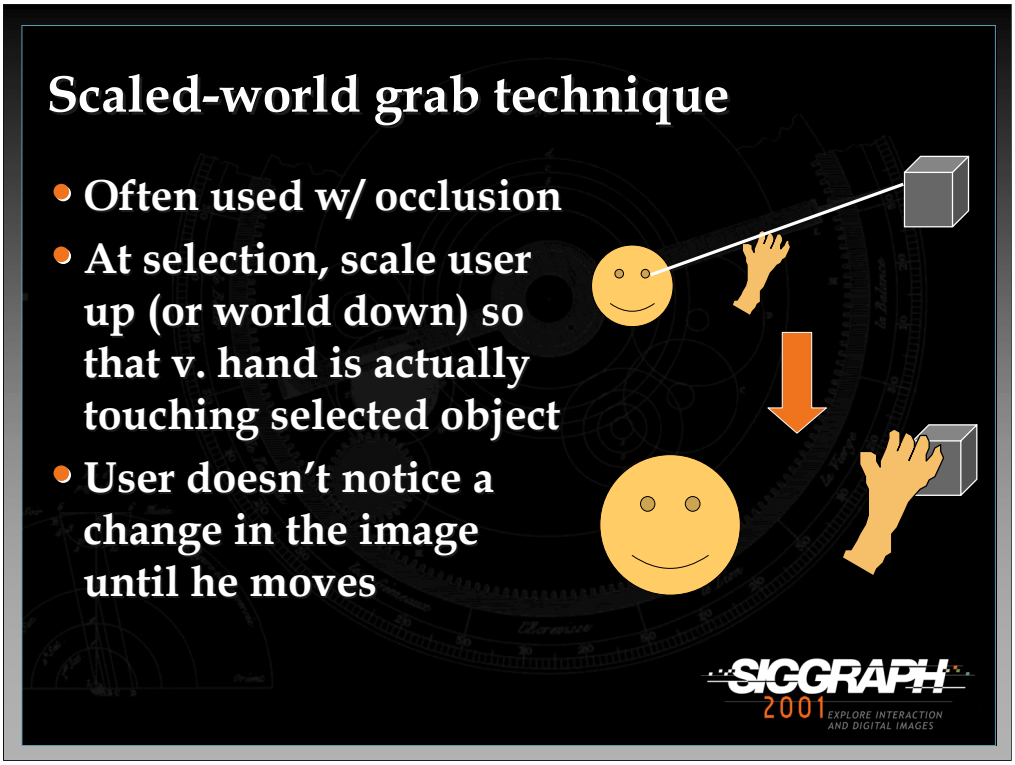
To implement the linear depth mapping, we need to know the initial distance between the torso and the physical hand, and between the torso and the selected object. The ratio d_o/d_h will be the scaling factor.



Now, each frame you need to set the position and orientation of the virtual hand. The selected object is attached to the virtual hand, so it will follow along.

Setting the orientation is relatively easy. You can simply copy the transformation matrix for the hand tracker to the virtual hand, so that their orientation matches.

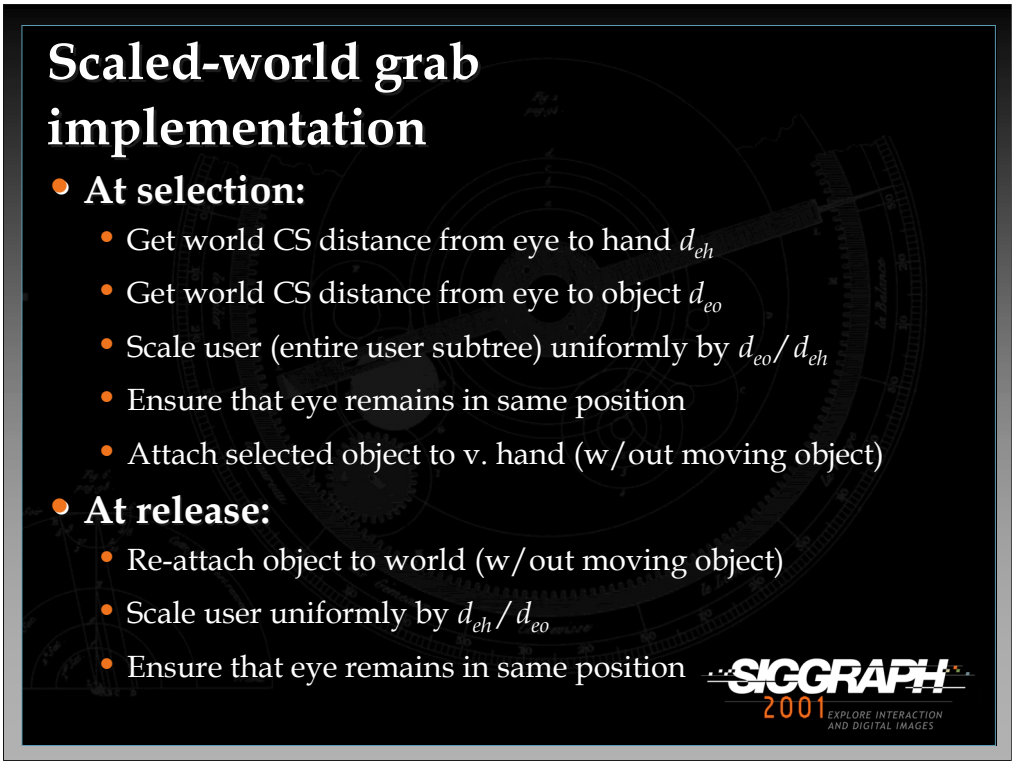
To set the position, we need to know the correct depth and the correct direction. The depth is found by applying the linear mapping to the current physical hand depth. The physical hand distance is simply the distance between it and the torso, and we multiply this by the scale factor d_o/d_h to get the virtual hand distance. We then obtain a normalized vector between the physical hand and the torso, multiply this vector by the v. hand distance, and add the result to the torso position to obtain the virtual hand position.



The scaled-world grab technique is often used with occlusion selection. The idea is that since you are selecting the object in the image plane, you can use the ambiguity of that single image to do some magic. When the selection is made, the user is scaled up (or the world is scaled down) so that the virtual hand is actually touching the object that it was occluding. If the user doesn't move (and the graphics are not stereo), there is no perceptual difference between the images before and after the scaling. However, when the user starts to move the object and/or his head, he realizes that he is now a giant (or that the world is tiny) and he can manipulate the object directly, just like the simple virtual hand.

See: Mine, M., Brooks, F., & Sequin, C. (1997). *Moving Objects in Space: Exploiting Proprioception in Virtual Environment Interaction.* Proceedings of ACM SIGGRAPH, 19-26, and

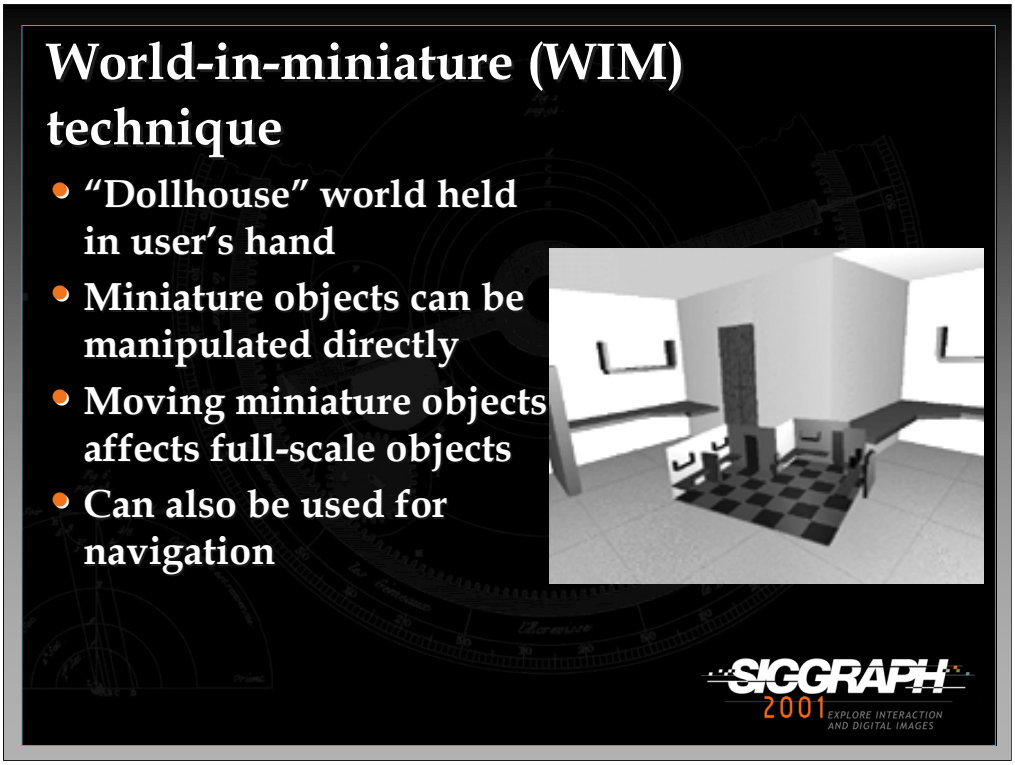
Pierce, J., Forsberg, A., Conway, M., Hong, S., Zeleznik, R., & Mine, M. (1997). *Image Plane Interaction Techniques in 3D Immersive Environments.* Proceedings of the ACM Symposium on Interactive 3D Graphics, 39-44.



To implement scaled-world grab, you must do the correct actions at the time of selection and release. Nothing special needs to be done in between, because the object is simply attached to the virtual hand, as in the simple virtual hand technique.

At the time of selection, you want to scale the user by the ratio (distance from eye to object / distance from eye to hand). This scaling needs to take place with the eye as the fixed point, so that the eye does not move, and should be uniform in all three dimensions. Finally, you attach the virtual object to the virtual hand as you would normally.

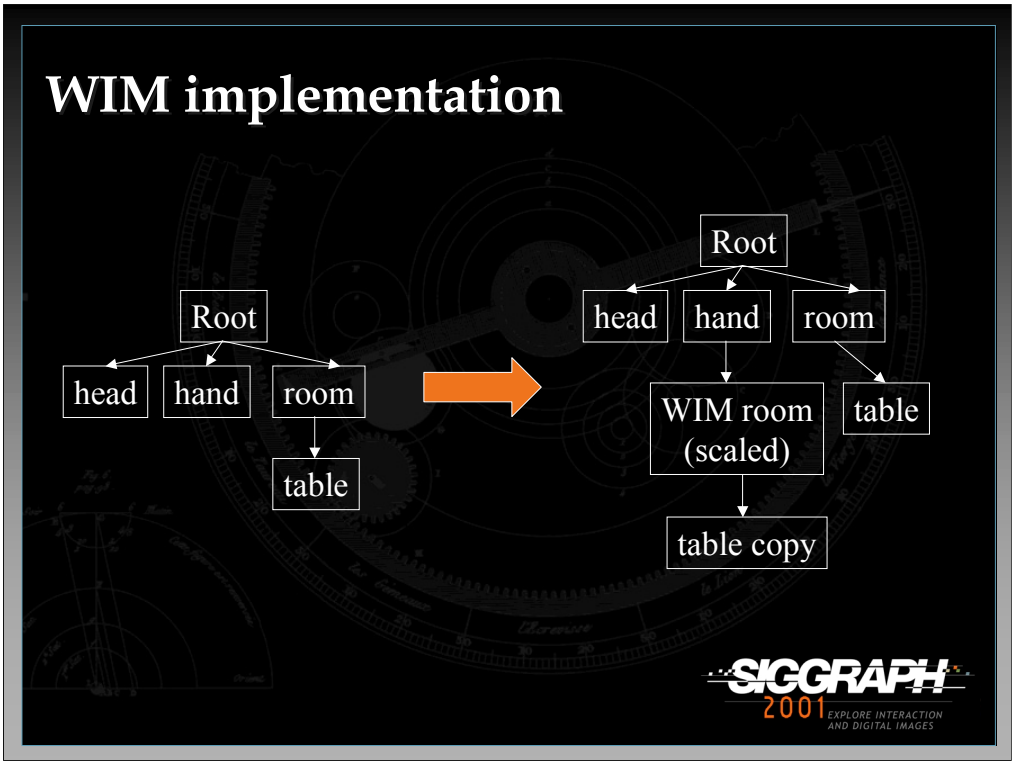
At the time of release, the opposite actions are done in reverse. You re-attach the object to the world, and scale the user uniformly by the reciprocal of the scaling factor, again using the eye as a fixed point.



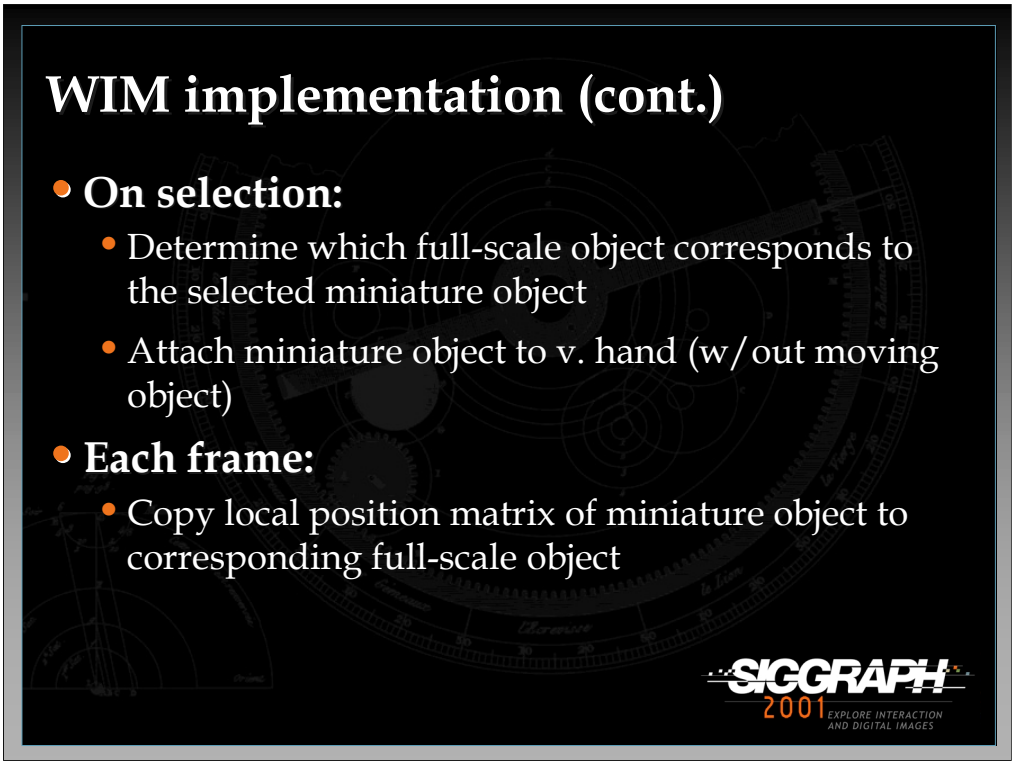
The world-in-miniature (WIM) technique uses a small "dollhouse" version of the world to allow the user to do indirect manipulation of the objects in the environment. Each of the objects in the WIM is selectable using the simple virtual hand technique, and moving these objects causes the full-scale objects in the world to move in a corresponding way. The WIM can also be used for navigation by including a representation of the user, in a way similar to the map-based travel technique, but including the 3rd dimension.

See: Stoakley, R., Conway, M., & Pausch, R. (1995). *Virtual Reality on a WIM: Interactive Worlds in Miniature.* Proceedings of CHI: Human Factors in Computing Systems, 265-272, and

Pausch, R., Burnette, T., Brockway, D., & Weiblen, M. (1995). *Navigation and Locomotion in Virtual Worlds via Flight into Hand-Held Miniatures.* Proceedings of ACM SIGGRAPH, 399-400.

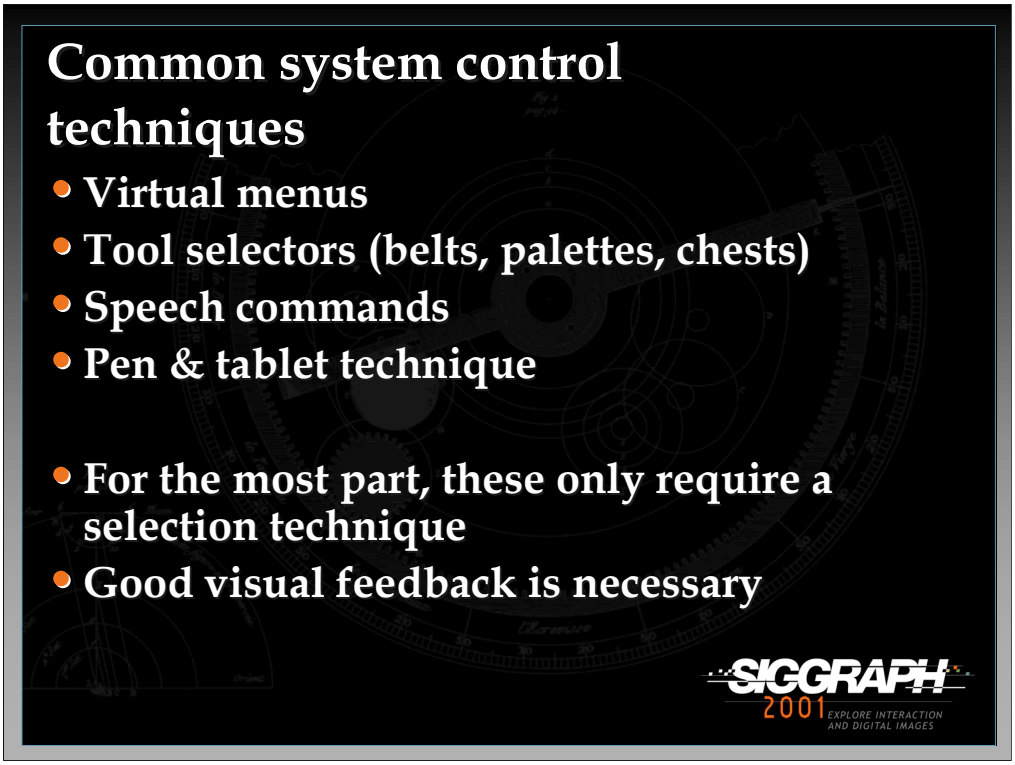


To implement the WIM technique, you first need to create the WIM. Consider this example, where you have a room with a table object in it. The WIM is represented as a scaled down version of the room, and it attached to the virtual hand. The table object does not need to be scaled, because it will inherit the scaling from its parent (the WIM room). Thus, the table object can simply be copied within the scene graph.



When an object in the WIM is selected using the simple virtual hand technique, you first have to match this object to the corresponding full-scale object. Keeping a list of pointers to these objects is an efficient way to do this step. The miniature object is attached to the virtual hand, just as in the simple technique.

While the miniature object is being manipulated, you can simply copy its position matrix (in its local CS, relative to its parent – the WIM) to the position matrix of the full-scale object. Since we want the full-scale object to have the same position in the full-scale world CS as the miniature object does in the scaled-down WIM CS, this is all that is necessary to move the full-scale object correctly.

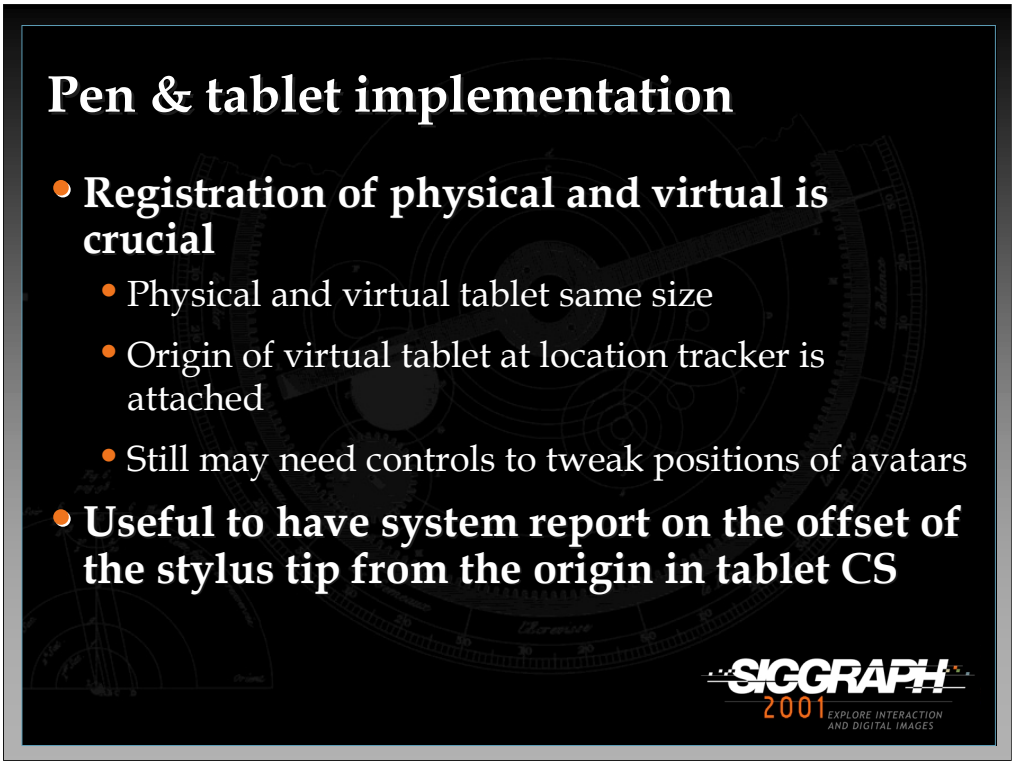


System control is a wide-ranging topic, and there are many different techniques, some of which are listed here. For the most part, these techniques are not difficult to implement, since they mostly involve selection, which we've already covered. For example, virtual menu items might be selected using ray-casting. For all of the techniques, good visual feedback is required, since the user needs to know not only what he is selecting, but what will happen when he selects it.

I only want to touch on one system control technique, because of its widespread use. The pen & tablet technique uses a physical pen and tablet (see left image). In the virtual world, the user sees a virtual pen and tablet, and a 2D interface on the surface of the virtual tablet (right image). The physical devices provide near-field haptics and constraints that make such an interface easy to use.

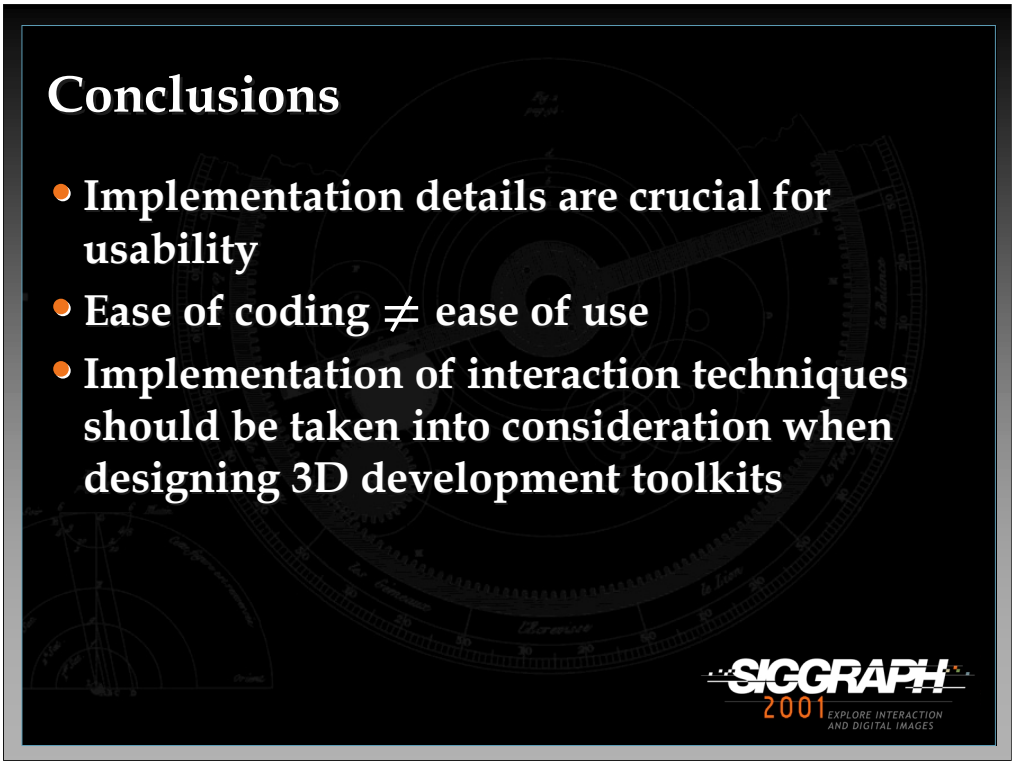
See: Angus, I., & Sowizral, H. (1995). *Embedding the 2D Interaction Metaphor in a Real 3D Virtual Environment.* Proceedings of SPIE, Stereoscopic Displays and Virtual Reality Systems, 282-293, and

Schmalsteig, D., Encarnacao, L., & Szalzvari, Z. (1999). *Using Transparent Props For Interaction with The Virtual Table.* Proceedings of the ACM Symposium on Interactive 3D Graphics, 147-154.



For the implementation of this technique, the most crucial thing is the registration (correspondence) between the physical and virtual pens and tablets. The tablets, especially, must be the same size and shape so that the edge of the physical tablet, which the user can feel, corresponds to the edge of the virtual tablet as well. In order to make tracking easy, the origin of the tablet model should be located at the point where the tracker is attached to the physical tablet, so that rotations work properly. Even with care, it's difficult to do these things exactly right, so a final tip is to include controls within the program to tweak the positions and/or orientations of the virtual pen and tablet, so that you can bring them into registration if there's a problem.

Another useful function to implement is the ability for the system to report (for example when a callback function is called) the position of the stylus tip in the tablet CS, rather than in the world or user CSs. This can be used for things like the map-based travel technique described earlier.



In this lecture, we've given an overview of the implementation of several important 3D interaction techniques. There are still more details that could not be covered here, but hopefully this gives the developer the basic tools needed to implement many types of 3D interfaces.

This information is also important because "the devil is in the details." A poorly implemented technique will likely have usability problems. For example, if the user is not scaled with the eye as a fixed point in the scaled-world grab technique, selection will cause the user to "jump", which could disorient users or even make them sick. Another important point is that the easiest techniques to implement are not necessarily the easiest to use, and vice-versa. It may take a lot of work to develop a usable technique. Finally, we hope that future tools for the development of 3D applications will include support for the implementation of interaction techniques, both through the inclusion of standard techniques and technique components, and through functionality that matches the needs of interaction designers.