The Design and Implementation of Direct Manipulation in 3D

Course organizer

Paul S. Strauss
Pixar Animation Studios

Speakers

Paul Isaacs
Eyematic Interfaces, Inc.

John Schrag
Alias|Wavefront, Inc.

Paul S. Strauss
Pixar Animation Studios

The popularity of direct−manipulation interfaces in 3D applications has increased steadily over the last decade. A direct manipulation interface allows a user to use a pointing device to "reach in and grab" objects in a 3D scene and move or change them in a natural, intuitive manner. This course is intended for application developers who would like to incorporate direct manipulation into their programs.

It is fairly simple to design and implement rudimentary 3D manipulation. This course will help developers create truly useful and intuitive interfaces by presenting solutions to several subtle problems.

This course concentrates primarily on manipulators for affine transformations. It covers design issues such as using modes, creating a consistent graphic language, dealing with visibility and collision, and choosing useful interaction behavior. The implementation module covers issues such as the relationship between the manipulator and the model, projecting input events onto geometric shapes, locate highlighting, maintaining geometric integrity, and constraining motion.

Contents

I − Course Notes

A. Front Matter
B. Introduction
C. Designing Good Manipulators
D. Implementing Suites of 3D UI Tools

II − Supplementary Material (Printed and CDROM)

E. A*n Architecture for Direct Manipulation of 3D Objects*, by Paul Isaacs, Rikk Carey, Howard Look, and David Mott.
F. *Techniques for Handling Complexity and Robustness in 3D Widgets*, by Paul Isaacs, Alain Dumesny, and Rikk Carey.
G. *A Manipulator for 3D Transformations*, by Paul S. Strauss and Paul Isaacs.

III − Supplementary Material (CDROM only)

H. Color slides for Introduction
I. motionParallax.mpg: A demonstration of how motion parallax provides depth information.
J. rotationParallax.mpg: A demonstration of how rotational parallax provides depth information.

Prerequisites

First Module: Design

Participants should have basic knowledge of user interface design ideas. They should understand terms like "affordances" and "modes". Some high−school mathematics may be required.

Second Module: Implementation

Attendees should be familiar with basic scene graph concepts, coordinate systems and transformations, and fundamentals of programming interactive applications (such as picking, selection, and dragging).

Presenter Information

Paul Isaacs works for Eyematic Interfaces in San Francisco, where he is Product Manager and Senior Architect for Shout3D, a java−based 3D toolkit. Prior to this he worked at SGI, where his focus was on 3D user interface. Paul architected, designed, and implemented the 3D UI for SGI's Inventor Toolkit, InPerson, and CosmoWorlds products. Earlier work included development of techniques for combining physics with keyframe animation, and a stint as Technical Director at Digital Productions. Paul received his Bachelor degree from Harvard and a Master's from Cornell.

John Schrag is an interaction designer for Alias|Wavefront in Toronto, Canada, where he has worked for eleven years. During that time he has worked primarily on the UI design and architecture of new products for 3d animation and visualization, and has taught several courses on user−interface design and usability practices.

Paul S. Strauss works on internal system architecture as a graphics software engineer in the Studio Tools group at Pixar Animation Studios. Before joining Pixar, he worked at SGI, where he was one of the principal architects of the Inventor scene graph toolkit and the Webspace Author/CosmoWorlds products. Paul received a Bachelor's degree from Brown University, a Master's from the University of California, and a Ph.D. from Brown.

Syllabus

First Module: Design

- **Introduction** (30 min) Strauss
  - What is 3D direct manipulation?
  - Advantages
  - Challenges
  - History

- **Design Issues** (75 min) Schrag
  - What makes a manipulator good?
  - Temporal and spatial modes
  - Manipulator appearance
  - Designing useful transformation behavior
  - Usability

Second Module: Implementation

- **Implementation Issues** (105 min) Isaacs
  - Using scene geometry: landmarks, follow−the−cursor
  - Maintaining consistency: look and feel, underlying architecture, commands, manipulators, constraints
  - Providing variety
  - Avoiding modes

The Design and Implementation of Direct Manipulation in 3D

Introduction

Paul S. Strauss
Pixar Animation Studios

What Is Direct Manipulation?

- WYSIWYG interaction with visual data
  - [Schneiderman ’82]
- "Direct": interaction is in same visual context as data
- "Manipulation": mapping from input valuator to changes to data
- Typically uses a *manipulator*

Why Is It A Good Thing?

- Result directly coupled to input motion
- User’s focus stays in work area
- Can guide/constrain interaction in natural ways

Haven’t I Seen This Before?

- Common in 2D
  - Handle boxes in drawing programs for scale, stretch, rotate, and drag
  - Drag–and–drop
- Becoming more common in 3D
  - View navigation
  - Manipulators for transforming/editing objects

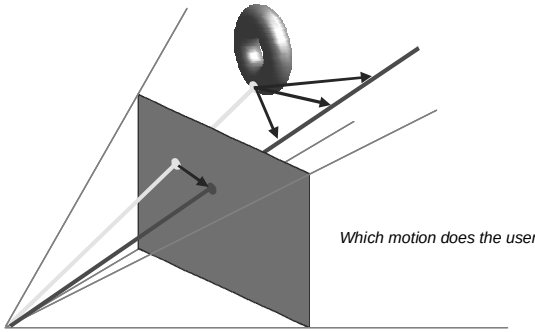
Isn't It Pretty Easy?

- Easy in 2D
  - One−to−one mapping from cursor position to spatial domain
  - Easy intersection testing
  - Straightforward interpretation of motion
  - No perspective distortion

Why Is It Hard in 3D?

- 2D view of virtual 3D world
- Ambiguity − infinite mappings from cursor position to spatial domain (line)
- Even worse when motion is considered

Ambiguous Input Mapping in 3D



Which motion does the user want?

Is That All?

- Perspective problems
  - How large handles appear relative to data
  - How large handles appear relative to each other
- Occlusion problems
  - Handles obscured by data
  - Entire manipulator obscured
- Precision problems
  - How to deal with exact placement
  - Mathematical instabilities

A Little Ancient History

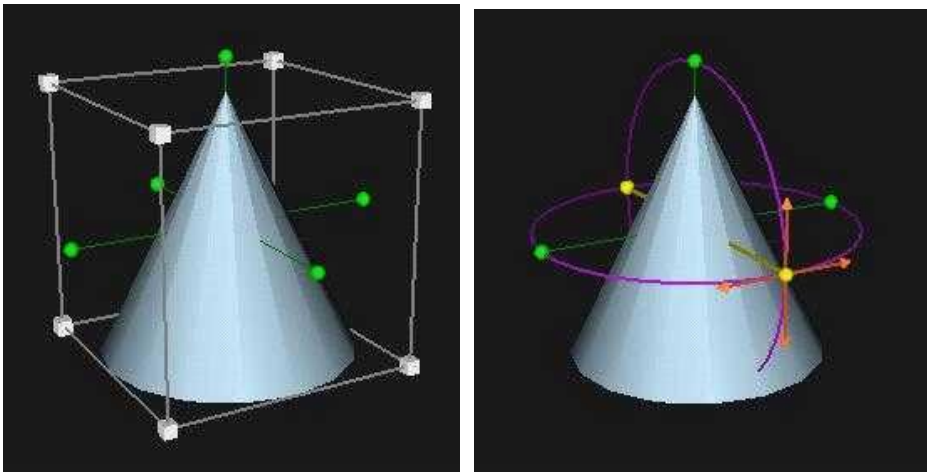
- 3D dragging
  - Softimage [’88]; Snap−dragging [Bier ’90]
- Direct rotation
  - Virtual sphere [Chen et al. ’88];
    Arcball [Shoemake ’92]
- Direct general 3D transformation
  - Inventor [Strauss/Carey ’92]
- Other 3D manipulation
  - 3D Widgets [Conner et al. ’92, Snibbe et al. ’92]

Related Areas

- Immersive environments (VR)
- Two−handed input
- Haptics

Designing good manipulators

John Schrag
Alias|Wavefront



(these Images courtesy of Paul Strauss, Pixar)

Designing good manipulators

This module of the course talks about how to design the appearance and behavior of your manipulators. It is easy to design a manipulator, but it takes some thought and work to design a **good** manipulator.

What is a manipulator?

For the purposes of this course:

A manipulator is a visible graphic representation of an operation on, or state of, an object, that is displayed together with that object. This state or operation can be controlled by clicking and dragging on the graphic elements (handles) of the manipulator.

What makes a manipulator good?

A good manipulator:

- fits naturally into the 3d (or 2d) world
- displays its functionality to you clearly
- supports, rather than obscures, your view of your work
- feels natural when you operate it
- provides immediate and clear feedback as to what it is doing

Assumptions:

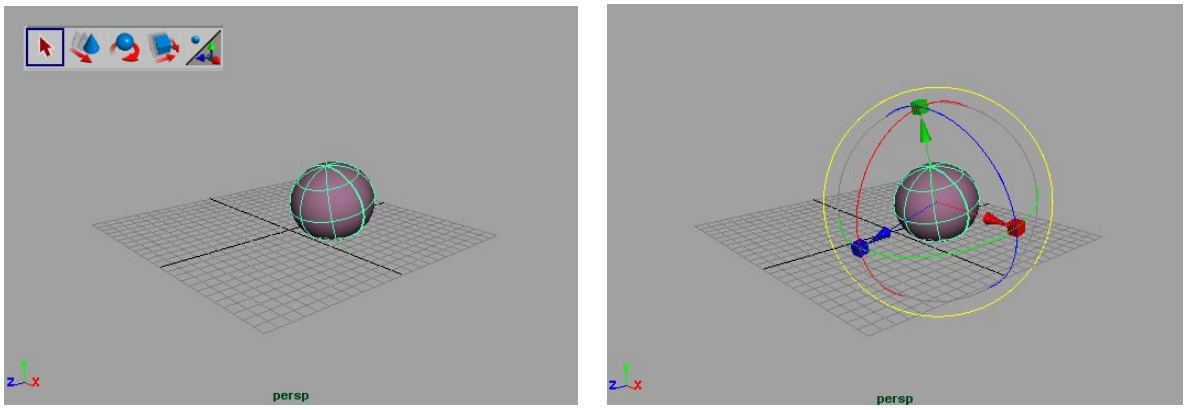
For the purposes of this course, I'm going to assume that you are working on standard systems with keyboard, screen and mouse only. If you have things like six-degrees-of-freedom input devices, or trackballs, or other fancy stuff, the rules change a bit --- sometimes a lot.

Why build manipulators?

Manipulators are generally used to replace tools, dialog boxes, menu items, or other kinds of 2d UI element. Here are some of the advantages of manipulators:

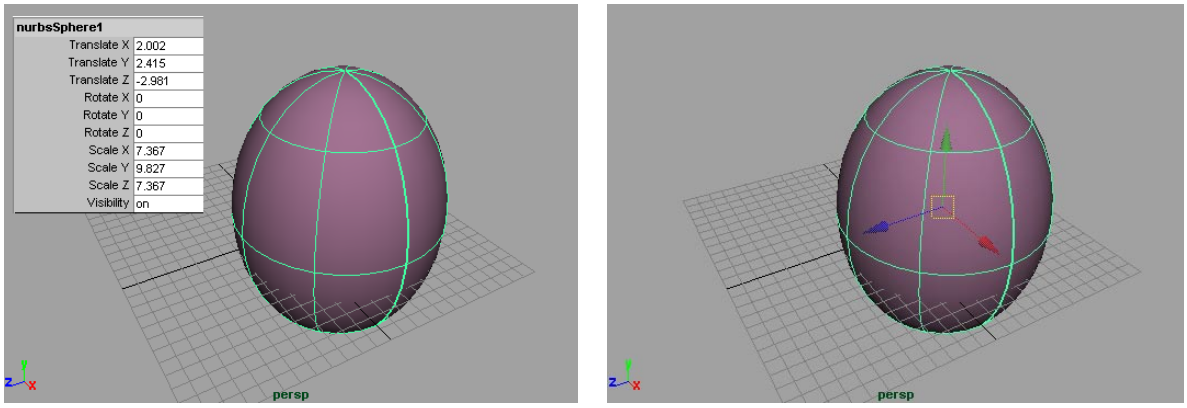
- Manipulators are located in the scene with the objects they control. This means that when you edit these objects, your locus of attention stays with the object, not off to the side with the tools. This reduces the amount of mouse traffic, and reduces mode error.

- When you use a well-designed manipulator, you have a number of different controls available at the same time, so you can perform any one of several related operations at any time without an extra click to change tools. This gives you a more satisfying sense of control, and cuts the number of clicks significantly. It can also reduce memory load, since all the possible controls are displayed right where they are needed.



The same functions, done with tools and with a manipulator. On the left, your cursor has to move back and forth quite a bit; on the right, all the functions are available right where you are working.

- Manipulators can graphically show you **what** they are operating on and **how** they will work. They take full advantage of your visual perception to give you operational feedback, in the context of the work being done. If you want to know which control will make the object go "up", you can tell at a glance. You can also see how far you need to move it. This is particularly important in 3d graphics.



Which way is up? On the left, you have to know what field you need to type in to move the sphere in a particular way --- and you need to know what number to type in. On the right, you just grab the obvious handle, and pull it the right distance.

- Manipulators can show you what operations are possible, in a given context, and can give you additional feedback about intermediate states. They can often use real-world physical analogies that make their operation obvious. This reduces training time and errors, and increases user confidence.

- Manipulators invite experimentation. They make using the software more enjoyable.

What we will cover today (75 minutes)

A little user interface theory: **Temporal Modes and Spatial Modes** (5 min)

A little perceptual psychology: **How we perceive 3d** (5 min)

Applying the above: **How manipulators should look** (15 min)

And: **How manipulators should behave** (20 min)

Basic usability**: Making Manipulators Great**

Finally: **Some exercises** (20 min) and **questions** (10min)

Temporal modes and spatial modes

What are modes in UI design?

Think about all the 3d software you've seen or used. You take the mouse, click and drag in the window, and something happens. You may be changing the view, or creating a new object, or moving something in space, or applying color.

In all these cases you're providing the same input --- a click and a drag --- but different things result. **Each mapping of an input to a possible output is a mode**. For example, in one mode, a click and drag moves something. In another mode, a click and drag paints a line of color.

Why does software need modes?

Software needs modes because we have very limited channels of communication to our computers. Our computers can talk to us with big full color shaded images, stereo sound, and formatted text. But the only way we can talk to them is with keystrokes, a few buttons on the mouse, and the mouse position. That's a very narrow channel of communication. So we need to make our limited input channels go a long way.

Temporal modes and spatial modes

Tools are called **temporal modes**, because the program behavior is determined by **time**. When you select a tool, that selection decides the behavior of the program for the next while --- until you select another tool. When you click and drag in the 3d window, the result depends on what tool was last selected --- or, in other words, on the time that you clicked.

The alternative to temporal modes is **spatial modes**. In a spatially-moded system, it doesn't matter **when** you click, it matters **where** you click.

For an example of this, instead of looking at the 3d workspace, look at the tool palette itself. When you click in the tool palette, what will happen? Well, that depends on where you click. If you click in one region, you select the brush tool. If you click in a different region, you select the eraser tool. Time doesn't matter – you can change tools at any time --- but where you click on the palette does.

Manipulators are a way of replacing temporal modes with spatial modes in the 3d view.

What's wrong with temporal modes?

Temporal modes are a fine way of separating unrelated activities. But they are often used when you want to do a set of closely related activities, forcing you to switch back and forth between them. For these tasks temporal modes can be frustrating. Here's why:

- **Too much mouse traffic**. Often when you are editing something, you want to do a series of related tasks, and need to switch back and forth between them. This is especially true of operations like sizing and positioning, where one adjustment affects the other. With temporal modes you must always perform an explicit tool-switch between each operation. This can mean a lot of back-and-forth with the mouse, and a lot of extra clicks that don't mean anything other than "I'm going to do X next".

- **Temporal modes put a burden on human memory** --- you have to remember what mode you're in all the time. How often have you clicked in a program and been surprised because you had the wrong tool selected? I do it all the time.

  You also have to remember the different mappings in the tools --- for example, what the different mouse buttons do in the tool, not to mention the modifier keys.

  (These problems can be ameliorated somewhat by use of good mode feedback, such as changing the shape of the cursor)

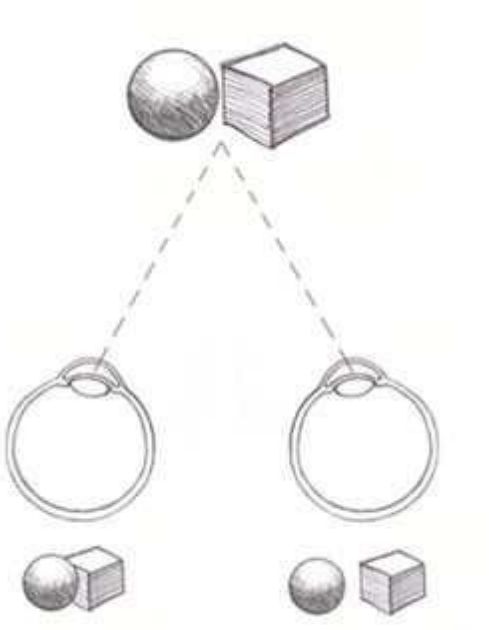
- **Complex operations can trap you**. Some software makes you go through a series of steps to do an operation. Finishing each step pushes you into the next mode, which requires more input, and there is not always a clear way to escape or go back. If you do abort, your partial input may be lost. Even when this works, you get the sense that the computer is driving the operation, not you.

How we perceive 3d

A good manipulator fits naturally into the 3d world. To make manipulators that look right, designers need to understand how people perceive 3d. If you don't, you can end up with manipulators that are completely misleading as to their position in space or their operation.

How do humans perceive 3d? Our eyes are cameras -- they see flat, 2d images like photographs. And yet, our perception of what is around us has depth. Everything we perceive is a construct of our brains, created from the information provided by all of our senses. Where our sensory input is incomplete, our brains can fill in missing areas. For example, we don't really see much color around the edge of our visual field, but we perceive it to be there --- our brain interpolates from other things in our visual field. Similarly, our brain generates the perception of depth from what we see, what we hear, and even kinesthetic cues.

Most of us learned in school that the brain uses **binocular disparity** to determine depth. Binocular disparity is the difference in the images that you see in your left and your right eyes. Since the eyes are spaced apart, each sees the world from a different angle. Your brain looks at the differences between the images and determines the distance of objects relative to whatever object you are looking at.

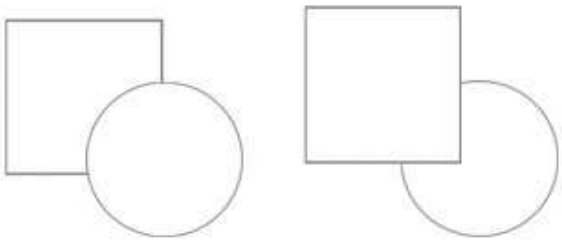


Binocular disparity. Left and right eyes see the same objects from different angles, and your brain combines the pictures to create a sense of depth

But binocular disparity is only one of the cues that your brain uses to perceive depth, and it's not even the strongest. There are many other cues available. Knowing what they are can help you design manipulators with a good sense of location in space, and avoid optical illusions which can affect their usability.

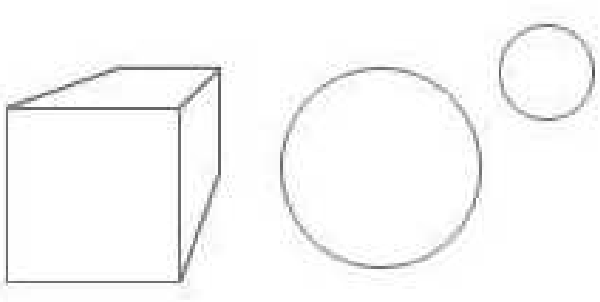
Here are some other depth-cues[1]:

Interposition: One object partially blocks the view of another, so you know which one is closer.



The objects are the same size in each image, but interposition makes one appear to be in front of the other.

Linear perspective: parallel lines appear to converge on the horizon. Objects farther away appear smaller than objects close up.



Linear perspective. Smaller sphere appears farther away, all else being equal.

[1] This information taken from a much more comprehensive description in: David Drascic and Paul Milgram *Perceptual Issues in Augmented Reality*. Published in SPIE Volume 2653: Stereoscopic Displays and Virtual Reality Systems III, San Jose, California, USA, January - February 1996 pp 123-134

Texture perspective: Objects that are farther away seem to be more densely textured

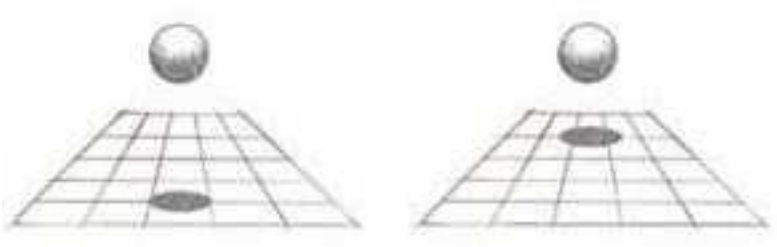
The spheres are a similar size, but the texture density makes the sphere on the right appear to be larger and farther away.

Aerial (atmospheric) perspective: Objects that are farther away lose clarity and their color shifts towards grey-blue as the atmosphere between scatters light.

Atmospheric perspective: objects fade into the distance. Notice how the faded sphere appears larger and farther away.

Shading and shadowing: The way that shadows fall across objects give clues as to its shape. Also, where shadows are cast can give important clues about the relative positions of objects.

Just by changing the shadows on the object surface, the same circle can have many shapes.



Shadows also give important cues to help us figure out depth.

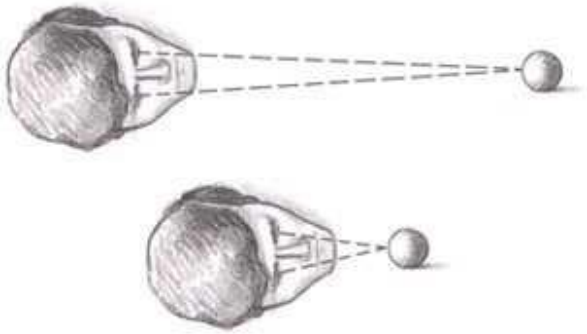
Relative motion parallax: When you are moving, distant objects appear to follow you while closer ones move away.

Rotational parallax: When an observed thing is rotating, different parts of it move at different rates and directions relative to other parts. This is a very strong depth cue.

Motion perspective: When a set of objects is moving together, the difference in their perceived speeds provides depth information.

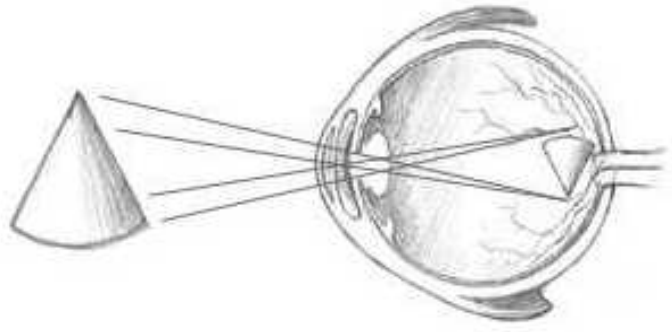
There are other depth cues that we can't make reasonably make use of, such as:

Convergence: When you look at a particular object, your eyes both need to point at it. The angle that your eyes have to converge tells you how far away the object is.



Convergence. The angle between the eyes tells you how far away the object of focus is.

Accommodation: When you look at a close object, the muscles in your eye pull to flatten the lens so that you can focus on it. For distant objects, these muscles relax. Your brain uses even this tiny bit of information to help it create a 3d world of the perceptions around you.



Your eye tenses and relaxes muscles that warp the lens, allowing you to focus on objects at various distances.

As was stated above, some of these depth cues (such as convergence and accommodation) can't be controlled on a computer display. And others (such as binocular disparity) would require additional equipment with today's technology. But we can make use of many of the other depth cues, such as color, interposition, shading, and shadowing.

The next section shows you how.

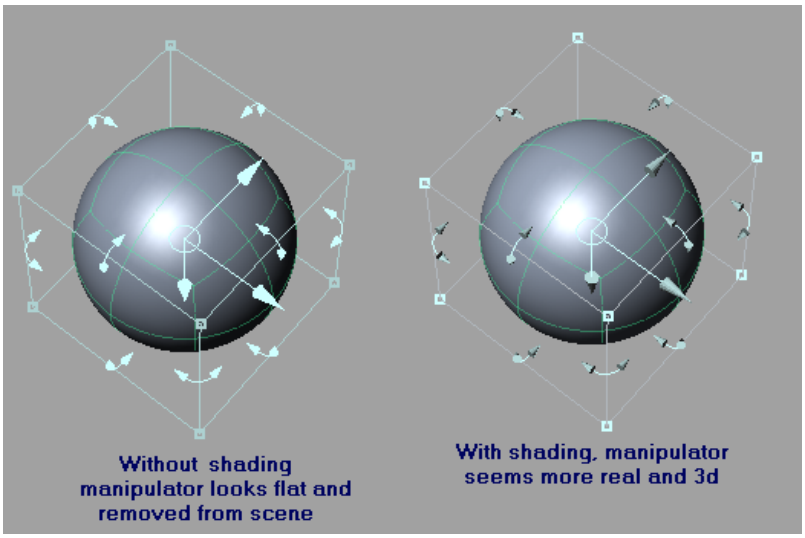
How manipulators should look

You should be able to distinguish the manipulator from the data

The users of your software should be able to tell at a glance which things in the scene are your data or your model, and which things in the scene are manipulators. The manipulators should visually invite the user to click on their different hotspots. The trick is to make the manipulators look different enough to be distinguishable, while still fitting into the scene. There are several strategies for differentiating:

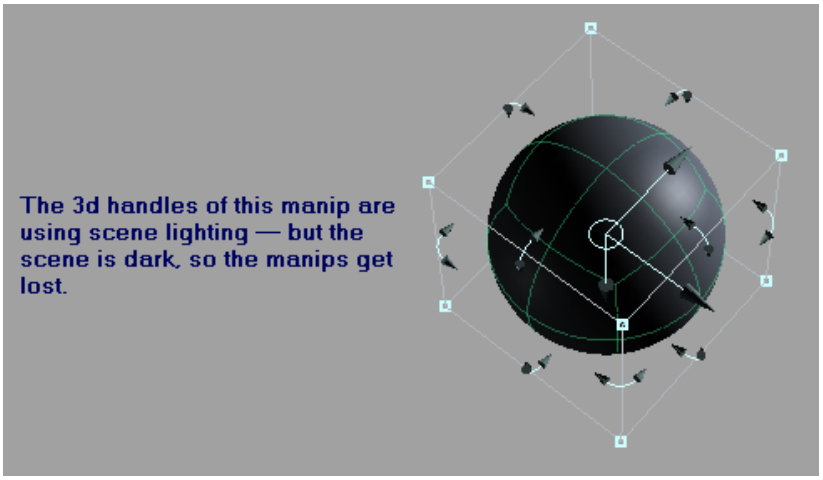
Shading When you are choosing a shading model for your manipulator, you probably want a flat shading model of some kind. If you use a shading model that has "hot spots", you can easily end up with a hot spot over a manipulator handle, or manipulator lines that disappear against hot spots in the background.

It is important to have shading, though. If you work without a shading model, the manipulator will look flat and not a part of the scene. (Remember that shading is one of our depth-cues)



Shading

Lighting Manipulators can be lit consistently at all times, ignoring whatever light sources are in the scene. This is probably a good idea in any case; if your manipulators use scene lighting, you can get into situations where they are lost in shadow, and therefore unusable. Consistent lighting (say, from the camera) allows your manipulator to have a natural look, but stand out from other objects in the scene. (The manipulator is still shaded, providing a 3d sense, even if it is not shaded by the same lights)

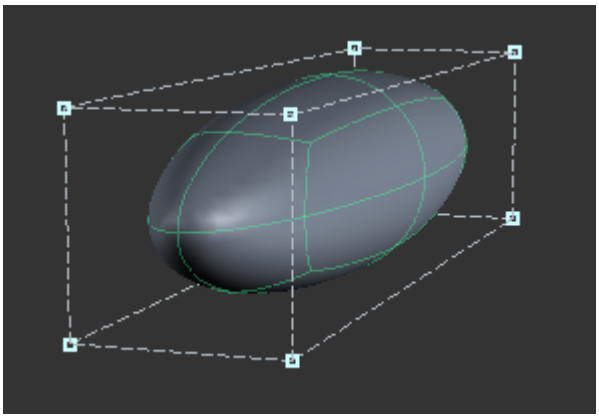


Lighting

You can also give manipulator a slight glow, but be careful about this --- too much glow will make it look like the unshaded manipulator above.

Color Depending on your application, it may be feasible to use color to distinguish your manipulators. Usually 3d graphical data comes in a variety of colors, so I wouldn't count on it. You must also take into consideration the fact that a significant proportion of humans are color-blind, so you should never encode critical information in hue alone. (If your colors are differentiated by brightness, too, that can work)

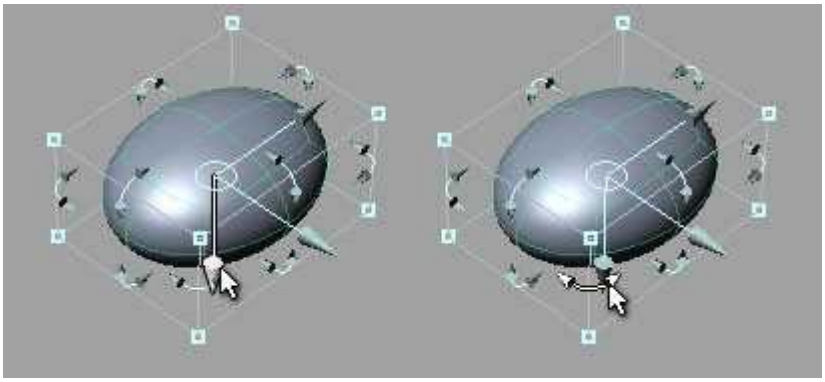
Style You can draw your manipulator to look very different – perhaps more sketchy --- than objects in the scene. For example, you could draw it with dashed lines, or in thin wireframe when the rest of the data is smooth shaded. Be careful, however --- If you are using dashed lines, it can really ruin the illusion of 3d unless you can get the dashes to be closer together as the line recedes.



The dashes don't get smaller as they recede into the background; this somewhat "flattens" the manipulator. Dots would have worked better. (It also doesn't help that the back handles are the same size as the front ones!)

Rollover You can set up your manipulator so that parts of it light up when the cursor moves over it. There are a number of good reasons to do this. First, it alerts the user that there is something special about that spot. Second, it encourages the user to click and find out what

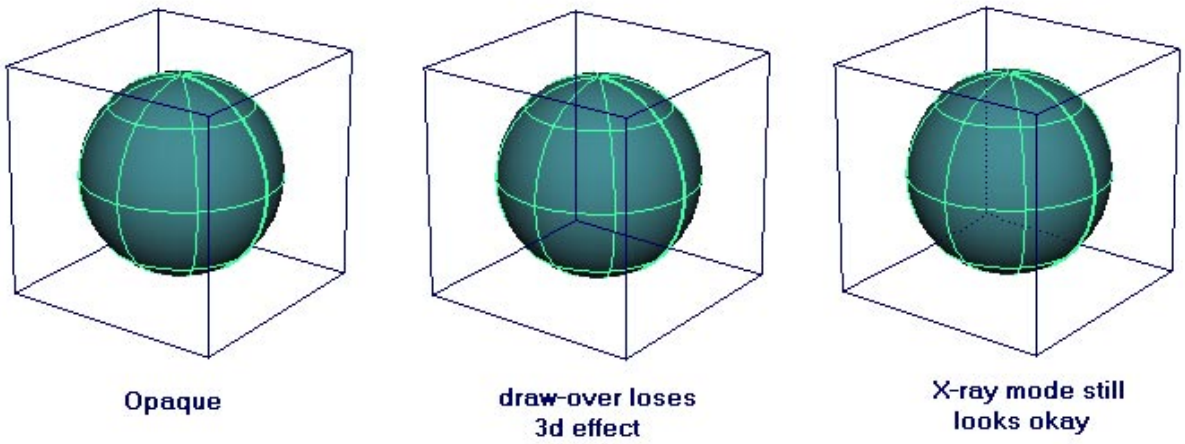
that is. This can lead users who haven't seen manipulators before into using them properly. Also, systems with this kind of rollover are perceived by users as being much more interactive and responsive.



The different handles "light up" as the cursor passes over them. This lets the user know which handle will be picked if she clicks on the mouse button.

X-Ray Should you be able to see a manipulator when it is behind an object? This can be a tough problem. The fact is that interposition is one of the strongest of all the depth cues, so if you draw your manipulator right through an object, it can completely destroy the 3d illusion.

But sometimes you need to have access to those controls on the back. One solution that works pretty well is to draw the obscured parts of the manipulator in a kind of "ghosted" fashion. This creates the illusion that the object itself is partly transparent, but maintains the sense of interposition.

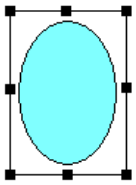


Drawing manipulators through objects (x-ray)

Environments Many 3d software programs provide multiple ways of viewing the 3d data. This can include wireframe and smooth shaded modes, x-ray modes, false color, etc. You need to check if your manipulator will work in each of these environments, and adjust it appropriately for each.

Use a consistent graphic language

Find out what other software your target users frequently use. If you can use visual cues from that software, use them. It will reduce training time and errors when they use yours.



Standard "scaling" manipulator, used in most 2d graphics software.

Manipulator handles that look the same should behave the same. And vice-versa. This lets your users build up some confidence that they know what will happen if they click on any particular handle, and how they should move it.

One handle should do one thing. Sometimes it will occur to an engineer that it would be efficient to pack a lot of functionality into just a few handles, by allowing different ways of interacting with each handle. This is just asking for trouble and confusion --- putting modes on modes.

Other recommendations

Don't be too subtle. It's very easy to get caught up in metaphor design, and how you can encode a lot of information into the shape and color of each handle so that a clever user can decode it.

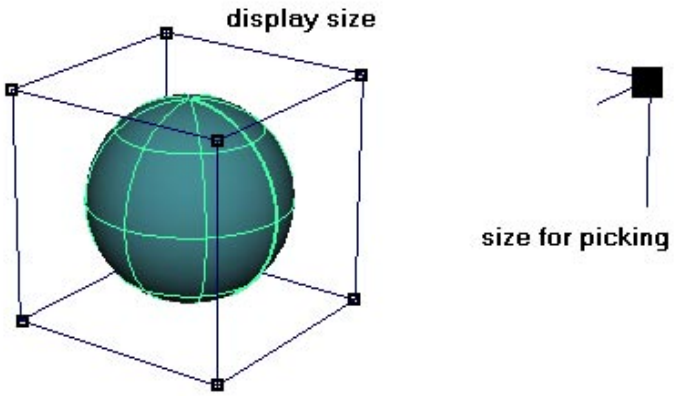
Don't draw more than you need to. When you are designing a manipulator, draw only what needs to be there to visually communicate with the user. For example, when the manipulator is not being used, the user needs to see all the handles (to know what functions are available). But while she is dragging the manipulator, the other handles are superfluous, so you can hide them (if that is not visually distracting).

Similarly, you may choose to only draw numeric feedback while the user is dragging. You may also add additional lines during drag, perhaps showing the path of a constrained handle, or showing the original position of the handle to allow the user to get back to it more easily.

Draw additional lines where needed. Not very part of a manipulator needs to be a selectable handle. We can draw additional lines to show relationships between the parts of a manipulator, or to indicate operation (such as the path along which a handle will travel). This may seem to contradict the recommendation not to over-draw, but it doesn't --- the idea is to think about the information that the user needs at any given moment, and give her precisely that information and nothing else.

Position handles carefully. In many manipulators, the position of a handle can seem like an arbitrary choice. But often having a good algorithm for placing the handle can make the difference between a usable and a hard-to-use manipulator. If you have a choice, ensure handles are on-screen, and if possible close to the cursor, which is the user's locus of attention.

Make sure your handles are big enough. It's very tempting to make tiny little handles that look nice, but are just about impossible to hit with a mouse. Remember Fitt's Law, which basically says that bigger targets are easier to hit.[2] Sometimes you will find that targets that are big enough to be hit easily may look pretty ugly. You can deal with this by having the active zone (hittable area) be larger than the displayed size of the handle. You can find the "right" size by user-testing --- it's often surprisingly large.



The handles are drawn small and pretty, but when picking they can be treated as if they are larger.

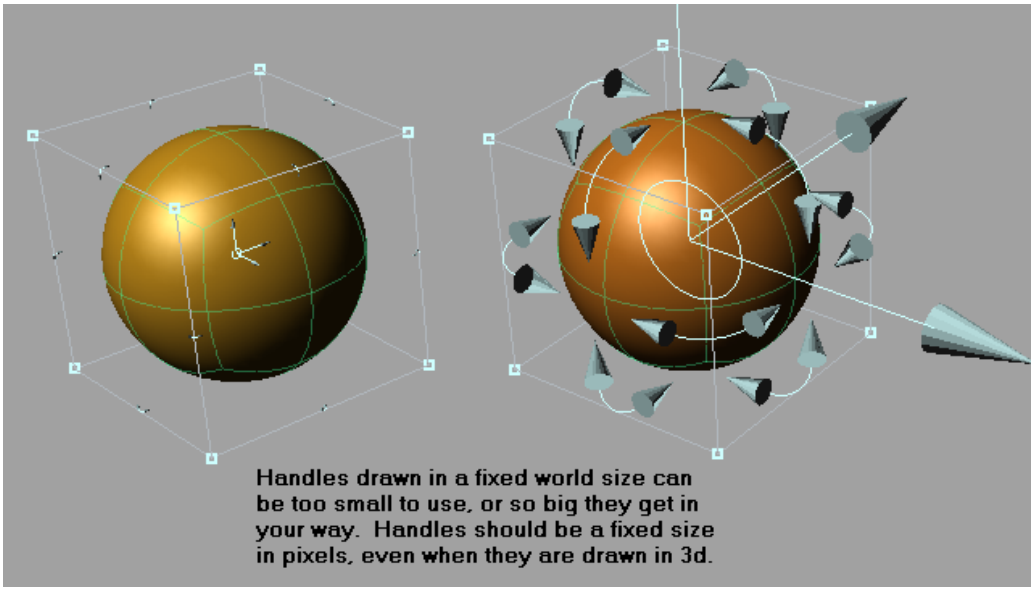
Problems and illusions

In what scale should manipulator handles be drawn? Manipulators are usually a funny mix of sizes. Some of the elements of the manipulator represent real-world sizes (such as the extents of a bounding box) and therefore must be drawn to real-world scale. But the handles are there for the user to interact with, and have no "real" size.

If you draw manipulator handles using fixed size in world-space, you have a problem. The objects you are manipulating can be big or tiny, and you may be looking at them from far away or from close-up --- so if the handle size is fixed, you can end up with handles that are too small to see, or too big to be manipulated.

---

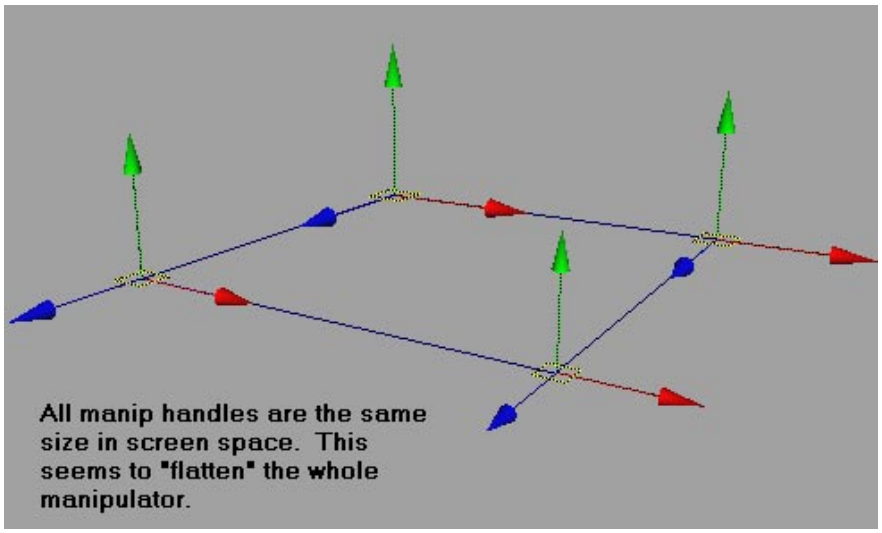
[2] Actually, it says a lot more than that, and provides a mathematical treatment of the subject. See Fitts, P.M. (1954*). The information capacity of the human motor system in controlling the amplitude of movement.* Journal of Experimental Psychology, 47, 381-391.



How big should handles be drawn?

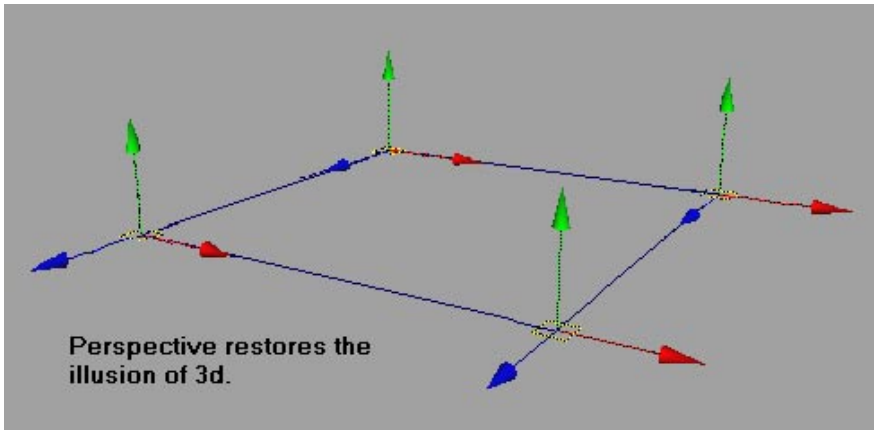
One way to deal with this is to use screen-based sizes. You draw the handles of your manipulators so that they are a constant size in screen pixels. This can work pretty well in many cases.

However, this solution can result in a nasty illusion when your manipulator contains handles both close-up and farther off --- because of perspective, the handles in the back appear unnaturally large, which messes up the 3d illusion.



This isn't a real manipulator, just a diagram to illustrate a point.

You can solve this by calculating the screen-based size once for the manipulator, and then drawing all the handles at a size relative to that. For example, you calculate that in the middle of the manip, 50 pixels equals 3 units in world space. Then you draw all the handles as 3 units in world space. This produces a pleasing result in most circumstances, with no handle too big or too small, and perspective preserved.

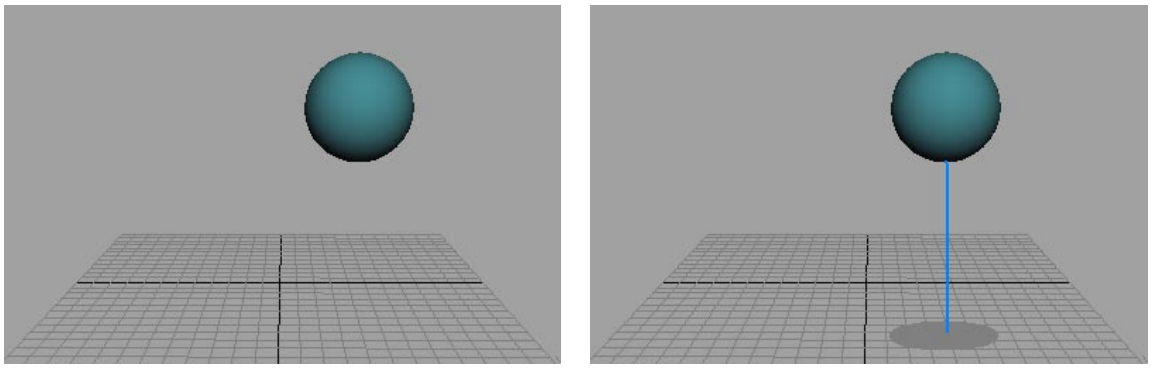


Handles that are farther away are drawn smaller --- but the size is calculated in screen space once per object.

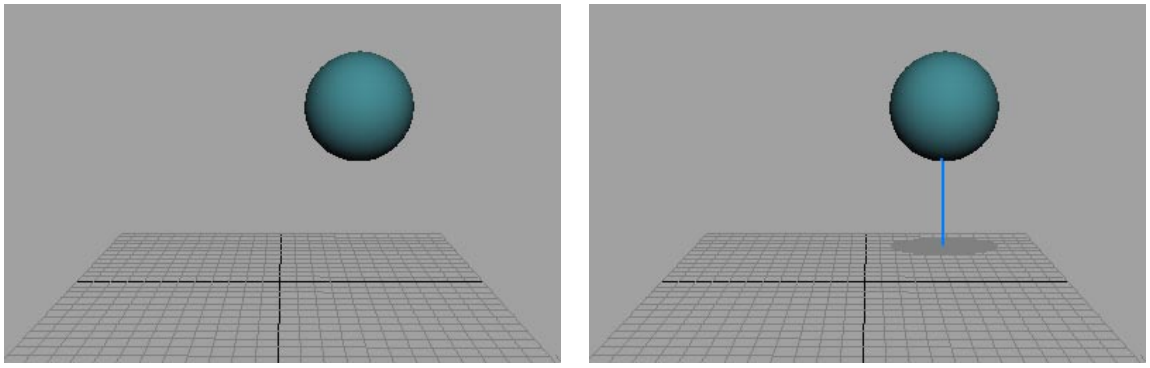
How do you give your manipulators a sense of location in space? Manipulators are often hovering in place in a scene, not clearly related to any particular object. This is especially true of manipulators that move things. It is very easy to move a manipulator handle somewhere, and be completely mislead as to where it ended up.

To solve this, we go back to our 3d perceptual cues. Unfortunately, we can't use the really strong cues (such as rotational parallax), but we can use a number of other cues.

One good strategy is to connect the manipulator to objects in the scene whose positions are known. This can be done by casting a shadow, or by drawing some kind of connecting lines.



Where is the sphere, and how big? You need more information than is provided in the first picture. In the second picture, a connecting line gives you the answer.



This sphere looks the same in the first picture, but the connecting line shows you it is both bigger and farther away.

How do you deal with collisions? One of the problems with having interactive controls live in your 3d space it that from certain perspectives, the handles will obscure each other, or there will be two handles on top of each other. This frequently leads to usability problems, as people click on the handle they want, end up getting the wrong handle, and are totally frustrated by the result.

One way to deal with this is to have the handles that are behind disappear when they are obscured. This should make it clear to the user that the functionality isn't available, but it doesn't always work well in practice – people often learn where handles are by their position on the manipulator, and will make the mistake anyhow. After they do, they can be frustrated because the handle they want isn't available.

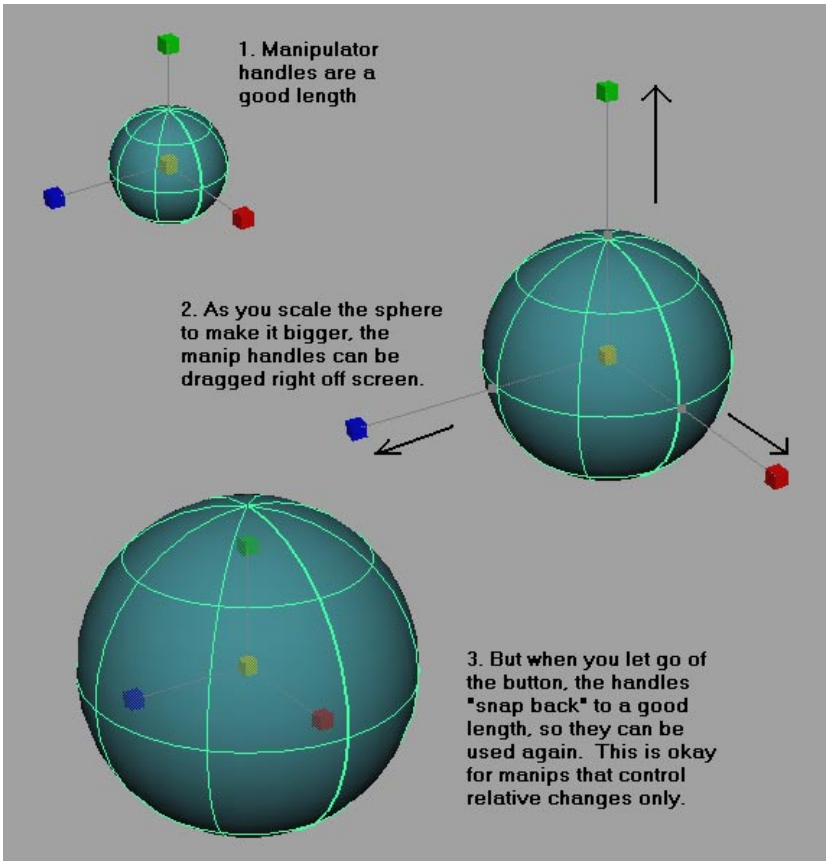
Another solution is to have the handles automatically move out of the way of each other when they get close. This can be a very nice solution, but it is a coding nightmare to get right, and the handles can sometimes end up in funny locations.

Another solution is handle priority. If you know that one handle will be used much more frequently, give it picking preference.

The best solution to this I've used is rollover. When the cursor gets over the handles, one of them lights up, so the user knows what will happen when she clicks the mouse button. People quickly learn to jiggle the mouse until the handle they want lights up before clicking.

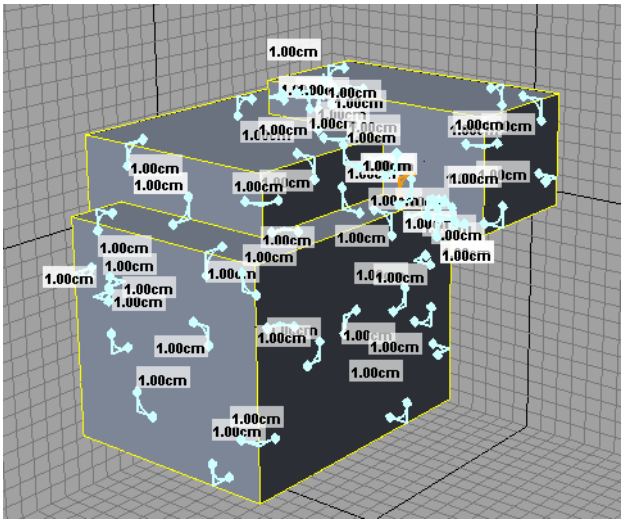
What do you do when handles go off-screen? Sometimes when you need to get a closer view of an object to do an operation, some of the manipulator handles will be pushed off-screen. I do not have a good general-purpose solution for this; I haven't found one that works better than letting people back off a little.

There are cases where a manipulator handle could feasibly be located in one of several different places. In these cases, the manipulator draw code should check how "good" each potential location is (based on location, collision, size, angle of view) and choose the best one. (You may also wish to hide handles that are unusable in the current point of view)



What do you do when your manipulators block the data? This can happen when you have a dense manipulator, or when you try to manipulate too many things at once.

Break down the functionality into groups, or just re-think your whole manipulator. Do you really think you user needs all of those controls at the same time? There is more about this in the next section.



Too many simultaneous handles can make a manipulator hard to use.

How manipulators should behave

Mappings

When you click on a manipulator, where you click determines what operation is performed. After that, you drag the pointer to control how much, or how far, or what direction, depending on the operation.

The fundamental problem of manipulator design is this: how do you map a fundamentally 2d input device --- a mouse --- into all those different operations with different degrees of freedom? This section looks at specific mapping strategies, and makes some recommendations.

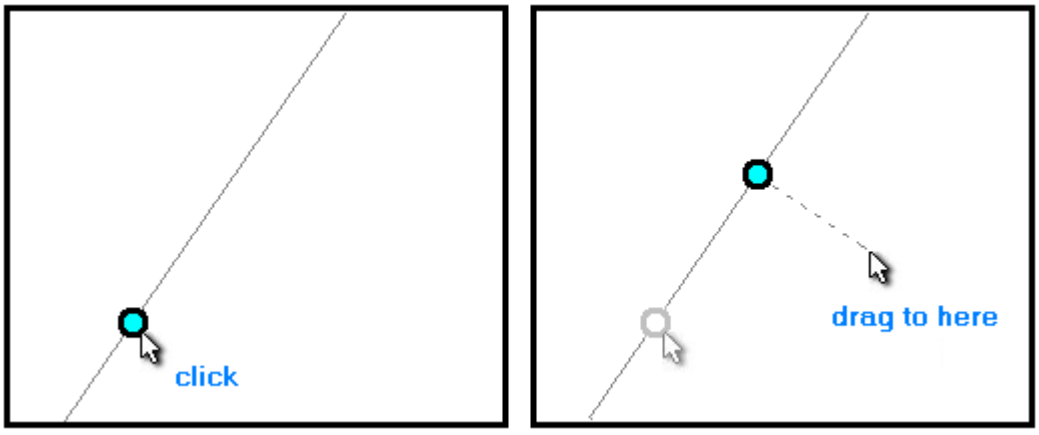
Getting the right mapping is critical to building natural, intuitable manipulators.

Positioning

The simplest manipulation is positioning -- moving an object from one place to another. In 2d graphics packages, the mapping is trivial --- the 2d position of the mouse maps to the 2d position on the screen. In 3d, things are never that simple.

1d positioning

Let's say you have an object that only has one degree of freedom --- it can only move along a line. The obvious mapping in this case is the correct one; you take the cursor position, find the closest point on the path line, and continuously move the object to that point. That will give the user a very natural sense of control.

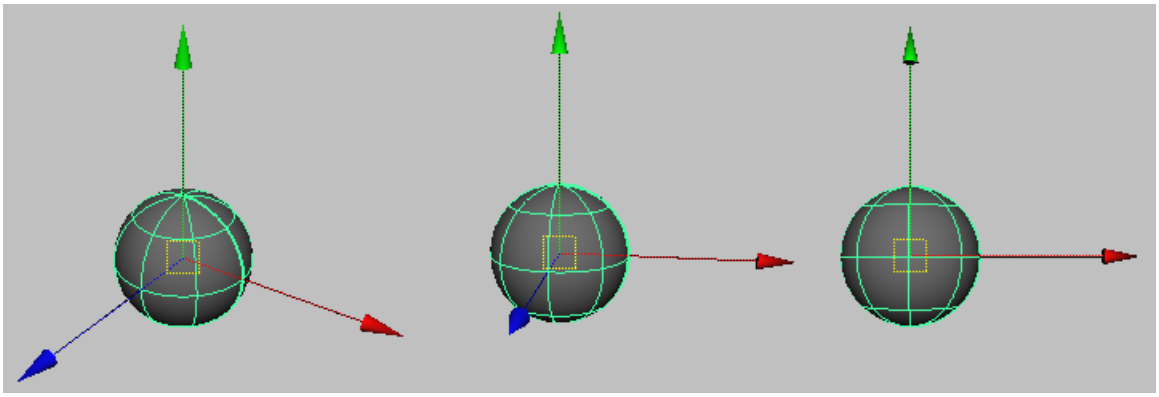


As you drag the cursor, the handle is constrained to a line. It moves to the point on that line closest to the cursor (as seen from the user's perspective)

Of course, there's always a catch. This strategy works well only when the path line is not too parallel to the view direction. When the path line is close to the view direction, the mapping can become extreme --- a movement of only a few pixels can push the object a hundred meters away, or fling it behind the

camera. In these cases, users often have no idea what just happened; it seems to them that the object disappeared.

The best strategy for dealing with this is to make the manipulator vanish when it gets too close to the view direction. This works well, because it encourages the user to rotate the whole view to a better position to use the manipulator.



The manipulator handle vanishes when it gets too close to being parallel to the view direction. This prevents ridiculous mapping where the motion of a few pixels sends the object flying into space.

Another strategy that seems reasonable at first is to change the mapping in the extreme cases, to cap the amount of the motion to a reasonable value. This strategy doesn't work as well, because it becomes difficult to tell how much you have moved the object from that point of view. I don't recommend it.

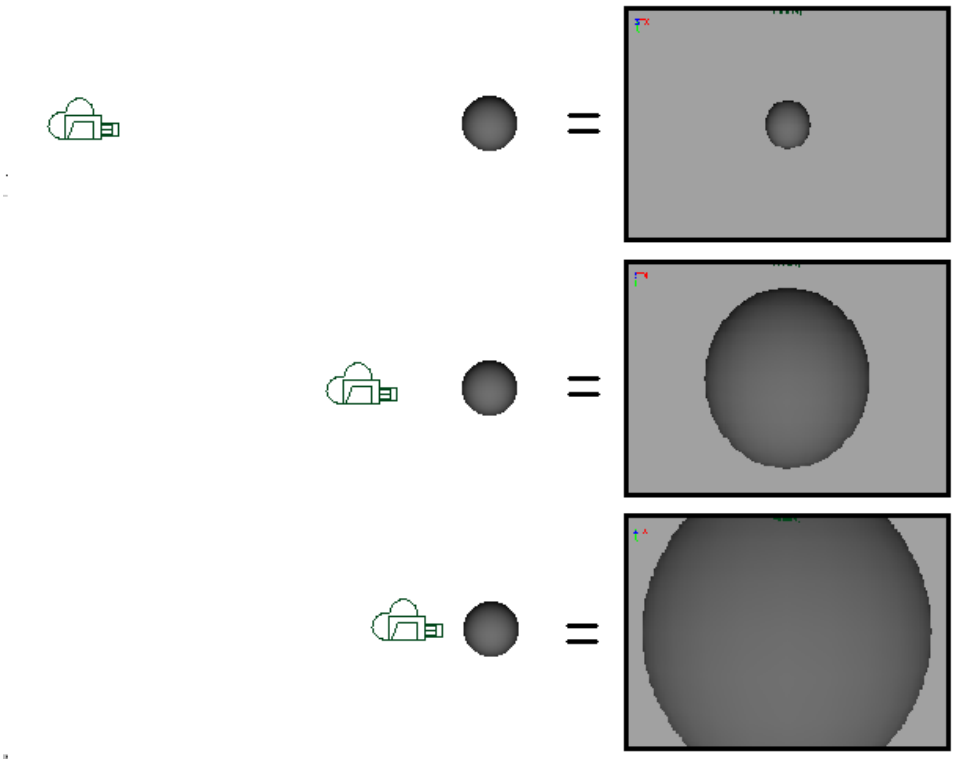
Zooming

Zooming or dollying the view in and out is a special case of 1d positioning. We have to map the up/down/left/right motion of the mouse into a forwards-backwards motion of the camera. How do we do that?

You can argue that any choice is as good as any other. In older software made by my company, if you drag the mouse left, you zoom out, and when you drag it right, you zoom in. This works well enough, but it turns out it wasn't necessarily the best choice.

There is another issue when dollying closer and farther. It turns out that if you map the cursor motion directly to camera motion, the result looks wrong. Things seem to come towards you way too fast, and they don't back off far enough or fast enough.

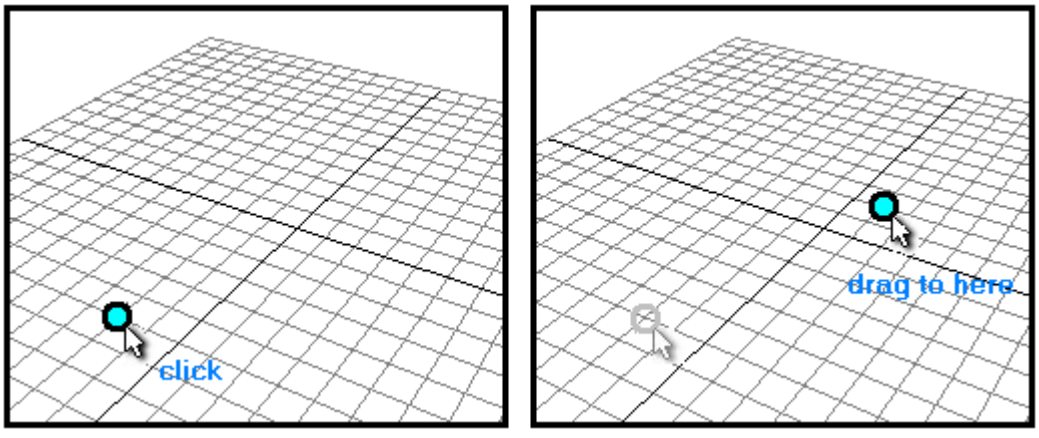
Think about zooming in to an object. What you really want is for the object to get closer and closer to you, without ever actually hitting you, sort of like Zeno's paradox. Another way of looking at it is this: you want to map the cursor motion to the apparent size of objects on the screen. That is, every unit you move the cursor should make the object look one unit bigger.



To create a zoom that "feels" right, the camera needs to move asymptotically towards the object. This makes the size of the object (in screen space) grow linearly.

2d positioning

Let's say you have an object or handle that can move in 2 dimensions on a plane. Once again, the obvious mapping is the right one. You project the cursor point onto the plane, and move the object to the intersection.

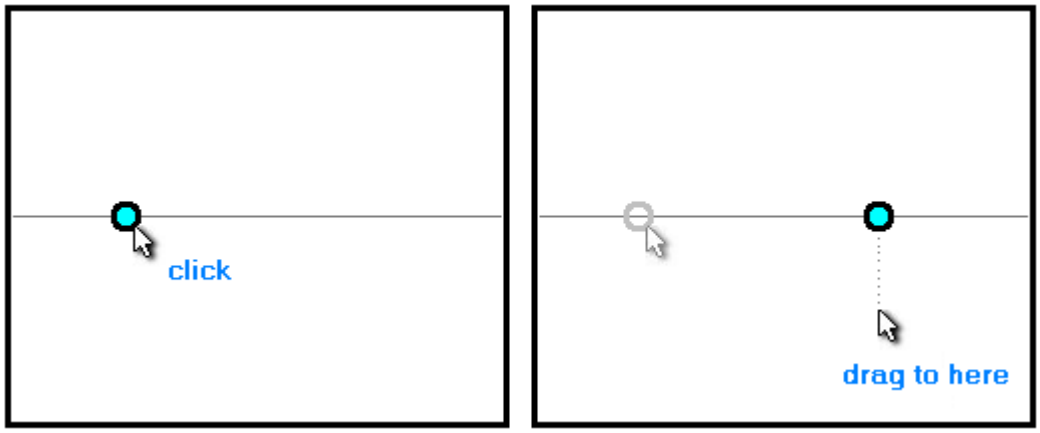


The handle is constrained to a plane. As you drag the cursor, the handle can stay with it, since there is an obvious mapping from the cursor position to a point on the plane.

Again we have the perspective problem. What do you do when you're looking at the plane of motion edge-on, or close to edge on? You end up in a situation where a tiny motion of the cursor can cause huge unexpected changes in the model.

There are two strategies for dealing with this; the first one is to make the manipulator disappear when seen close to edge-on. Again, this strategy encourages the user to rotate the whole view to a more useful orientation. (However, it may prevent the manipulator from ever showing up, which would be bad.)

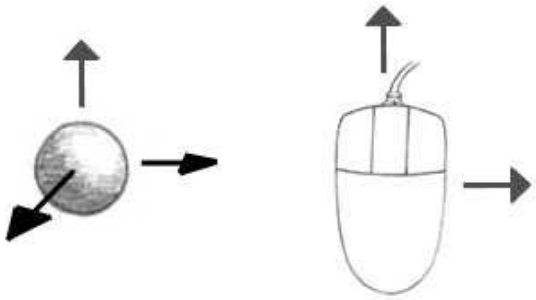
A better strategy is to limit the motion of the manipulator in the extreme case. When you are looking at the plane-of-motion edge on, limit motion to a line that runs through the plane, at right angles to the view direction. This works surprisingly well; users don't expect things to move towards and away from them when that dimension of motion isn't visible.



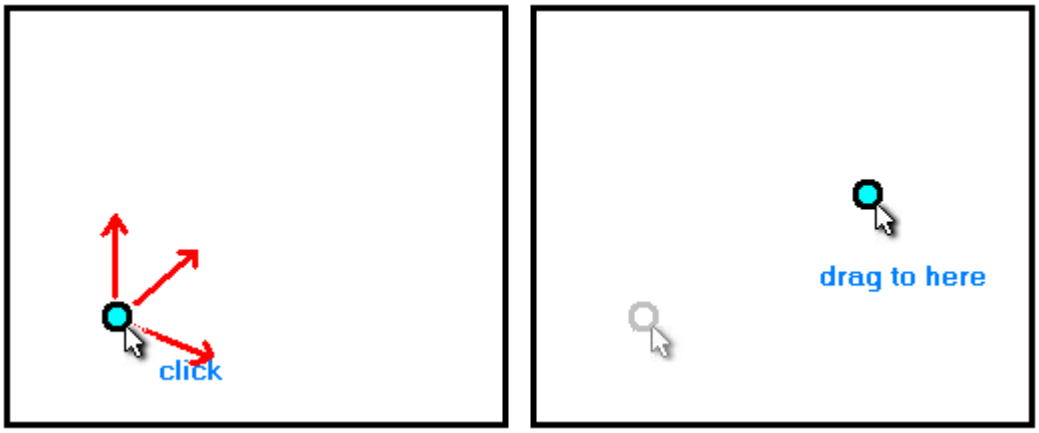
The handle here is also constrained to a plane, but in this case we are looking at that plane edge-on. In this case, the handle should never move towards or away from the user; instead we map its motion to a single line, which is the intersection of the plane of motion with a plane perpendicular to the camera view direction, passing through the original handle location.

3d positioning

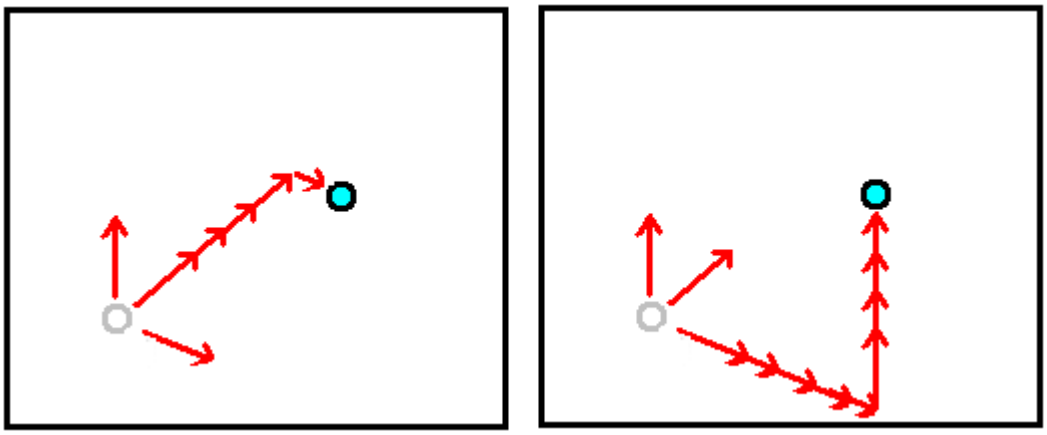
Let's say you have an object that is free to move in 3 dimensions freely when you click and drag on it. This poses a real problem, because the mouse is only giving you 2 degrees of input, and you are using them to control 3 degrees of freedom. So your manipulator must provide this extra information.



An object can move any direction in 3d, but the mouse that controls it can only move up, down, left, or right.



In this case, the handle is free to move in all three directions (x, y, z). You can drag it to a new location, and it seems to follow the cursor. Seems easy, right? But the trouble is that there are an infinite number of possible positions for this handle that satisfy the 2d-to-3d mapping.

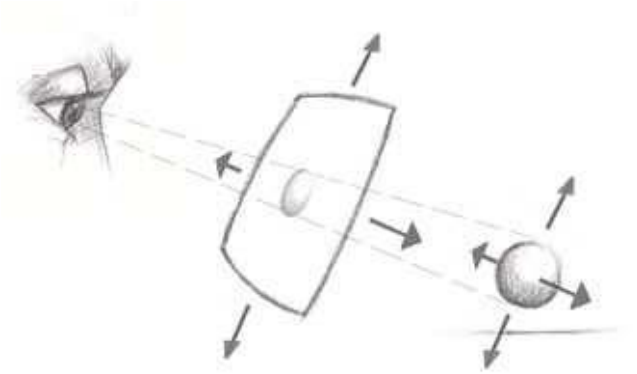


For example, did the handle move mainly back and a little bit over, or did it move right and up? In user testing, we found that users assumed that the object moved where they intended it to move, even when perspective in the image indicated otherwise. User expectations are also non-constant, even across the same task.

There are a number of strategies you can employ:

- Don't do it. For many applications, I think this is the best strategy. What you do instead is provide a plane handle and a 1-d handle, or three 1-d handles. This forces the user to make deliberate choices when moving the object.

- Move the object in the current view-plane. This approach is appealing to engineers, because it is non-arbitrary and mathematically clean. The trouble is, unless you are looking at a scene in a very dead-on way, this results in a motion that is completely non-intuitive, and frequently misleading. In usability testing, our testers never ended up putting the objects where they thought they were putting them.



A common, but not very good solution. The 2d motion of the mouse is used to move objects parallel to the 2d plane of the screen. This looks okay, but objects end up in unexpected positions.

- Use obvious structures in the scene to determine the plane of motion. For example, if there is a drawing plane, move objects parallel to it. If you are positioning something on a surface, then "snap to" that surface. This kind of solution depends on the application domain.

- Guess, based on initial cursor movement. Any kind of guess would require you to come up with a heuristic for determining which way to go, and these are always risky. It is possible to use heuristics that guess the user's intentions nine times out of ten, or even nineteen out of twenty, but those remaining cases can be very frustrating.

  If you do decide to use a heuristic, then it should have these properties: first, it should make its guess very soon after the cursor starts to move (or else users will perceive the delay as a non-functional handle or annoying lag), and second, once it makes a guess it should stick to it. This is to provide a good isomorphic mapping. (More on this below).

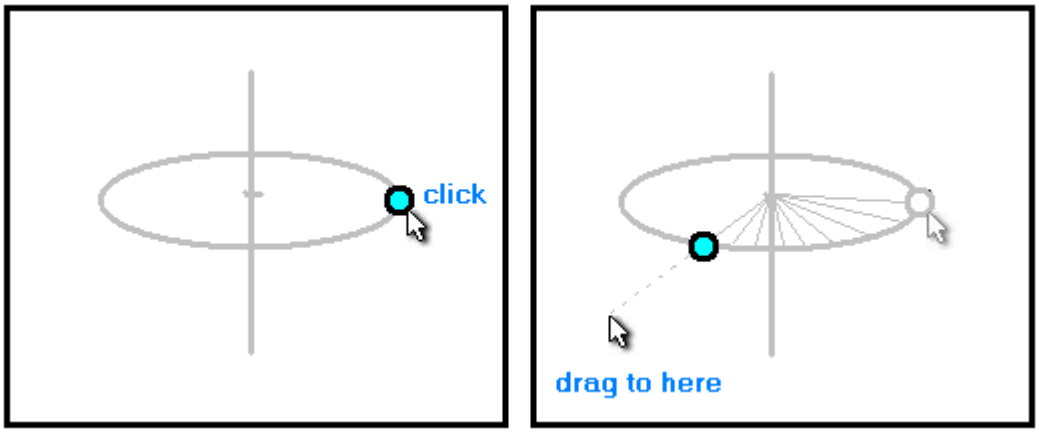
If you are looking at your data in an orthographic projection, you should limit any motion to the plane of that projection.

Rotation

Rotation around an axis

Let's say you have an object that is constrained to rotate around a fixed axis. This is one degree of freedom, so we're okay. There are two basic strategies.

- **Radial mapping**. Project the cursor onto the plane of rotation, and get the radial distance it travels around the axis. Rotate the object by that amount. This has the advantage of allowing you be as precise as you want by holding the cursor far from the axis (giving you a longer movement arm). It also feels very natural.

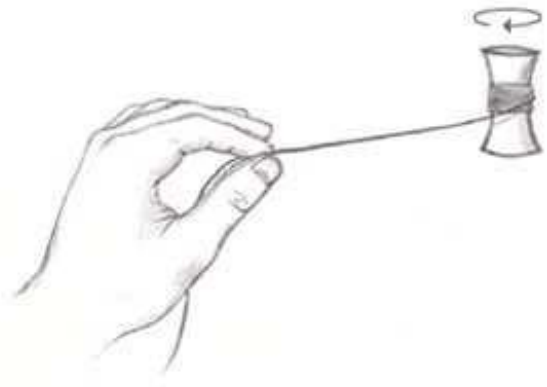


The handle is constrained to rotate around a fixed axis. Project the cursor onto the plane of rotation, and draw a line from it to the axis to calculate the new handle position.

But once again we have the edge-on problem. When we are too close to the edge, radial mapping produces unhappy results; things jump around or get stuck.

- **Linear mapping**. In this case, the distance the cursor travels left or right determines the amount of rotation. Some number of pixels equals a fixed amount of rotation. This works pretty well, but doesn't give you quite the same sense of control that radial mapping does, since you're one step removed from the rotation. The trick to this one is figuring out how much cursor motion maps to how much rotation.

There are a few different ways to do this; one fairly natural-feeling one is the unwinding-thread algorithm. If the object being manipulated has an obvious circumference, then you can let that distance map to a 360 rotation. (You may also wish to "amplify" this somewhat, possibly speeding up the spin as you get farther from the object. If you do this, make sure you preserve the isomorphic quality of the mapping)

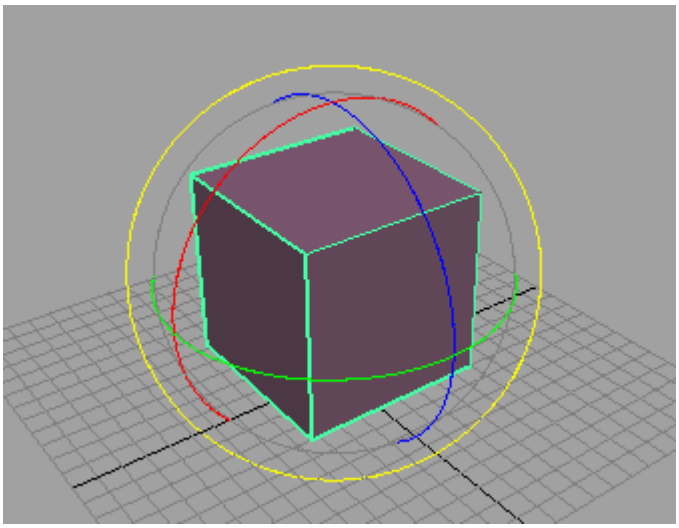


As the hand (mouse) pulls to the left, the object rotates. The rate of its rotation depends on its circumference. This produces a natural-feeling rotation for edge-on cases.

Radial mapping and linear mapping also work together quite well. You can use radial mapping until the plane of rotation gets close to an edge case, then switch to the other algorithm. You might think that would be confusing but it works very well in usability testing.

Free rotation

Let's say you have an object that can rotate freely in 3 dimensions. You could obviously put three rotation handles on it, one for each axis, and in fact that works very well, and gives people a good sense of control.



Simple three-axis rotation manipulator. (The yellow circle is an additional forth handle that lets you rotate the object in the view-plane --- an action that is almost always a bad idea.)

But there are ways to allow free rotation --- controlling 3 degrees of rotation with our 2-degrees-of-freedom mouse.

- **Arcball**. The Arcball[3] is a very elegant way of doing free rotation of an object, based on quaternion math. The idea is pretty simple; you have a spherical manipulator; you click on one part of it and drag to any other part. The object rotates along the shortest path between these points. This demos really well, but I have found it to be problematic in actual testing. When people try to use it for real positioning tasks, objects can quickly end up in funny tilted orientations, and users seem to find it difficult to get the objects just to where they want for precise positioning tasks.

In Maya, where we provide both Arcball and axis rotate handles, the rotate handles are used almost exclusively in practice.

- **Latitude/longitude rotation**. (azimuth/elevation)This can work well in situations where the object you are manipulating has a natural "up" direction, something you'd probably like to be able to get back to. What you do here is map horizontal cursor motion to rotation around the vertical axis, and vertical cursor motion is mapped to "tilt" --- bringing the north pole down to the equator. Since this mapping only allows 2 degrees of freedom, you cannot get all possible rotations, but for many applications you can get all you need. The interaction feels very natural, but can be frustrating if you are trying to get to one of the "disallowed" orienations.

---

[3] Ken Shoemake, *ARCBALL*, Proceedings of the conference on Graphics interface '92, p 151-156, September 1992, Vancouver, Canada

Scaling

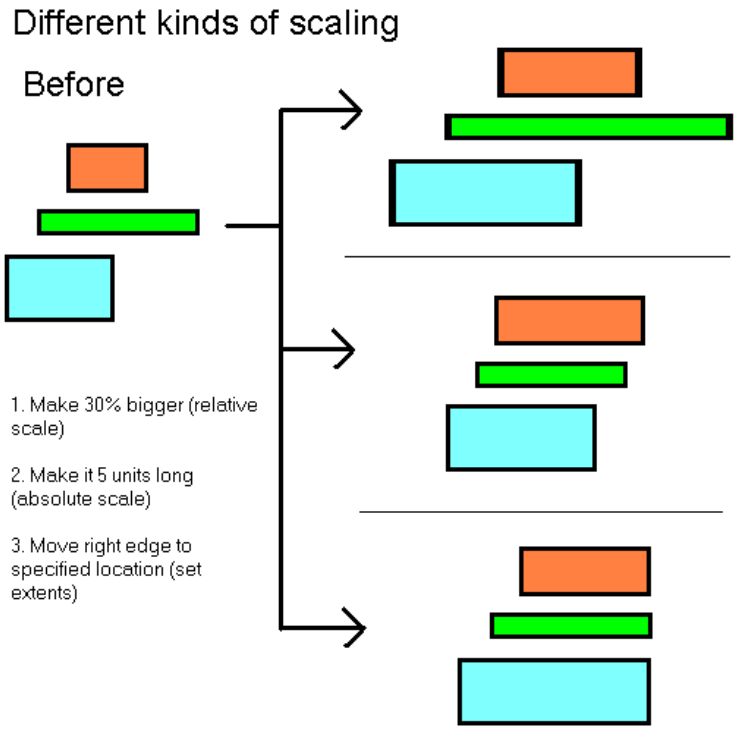
Scaling in 1d

Let's say you have an object, and it can grow or shrink along one axis. This is one degree of freedom, so it should be pretty straightforward. However, there is a basic question you must ask before you can design any scaling manipulator. Are you controlling the **scale** of the object, or its **size**, or its **limits**?

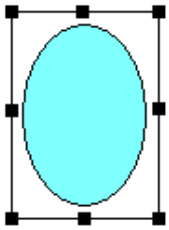
These are not the same thing. Imagine your users making the following statements.

- "I want to make these objects 30% longer"
- "I want these objects to be 5 meters long"
- "I want to move the right edge of these objects to be here"

These are all scaling operations, but they require different inputs and have different results. If your application needs to support more than one of these operations, you might consider having different ways of doing the different operations.



For this discussion, I'm going to use #3, wanting to control the extent of the object. This concept is familiar to just about anyone using a graphics program, because you've all seen it done beautifully in 2d.



Standard 2d scaling manipulator, found in many drawing packages.

We all know how to operate this manipulator, and this metaphor can be extended into 3d. By the way, notice how the different manipulator handles limit the operation --- some allow 1d dragging, and some 2d.

3d

What happens when you extend this to 3d? In Alias Sketch!, a program that was originally released 10 years ago, we provided a cubic manipulator. You could drag on a face to scale in 1d, or on an edge to scale in 2d, very naturally.

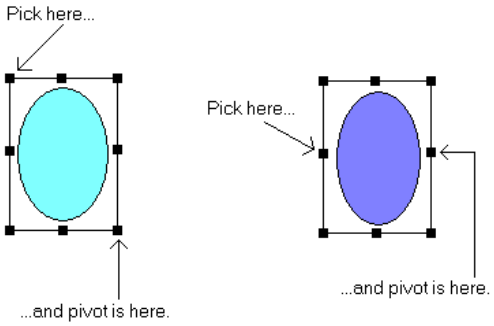
But what if you dragged on a corner? You'd want to scale in 3d, but by how much in each direction? This is exactly equivalent to the problem of positioning in 3d. In the 3d case, we decided that the scaling would always be proportional. (As it turns out, people don't really want to control three axes of scaling in one operation; that's just too hard in general.)

Pivots

Scaling and rotation operations require one more piece of information --- a pivot. It's not only a question of scaling how much, but of scaling how much around what point.

The cheap way of dealing with this is to say that the center of the world is always the pivot. That's a lazy programmer solution, and doesn't generally lead to happy users.

For scaling, it is often enough just to provide a variety of pivots in useful places. If you take another look at the standard 2d scale box, you will see that the different handles all perform the same operation, but picking different handles explicitly chooses different pivot points. There is a bit more to it than that, which we'll get to in the next section.



Every handle does the same thing – scaling – but different handles implicitly set different pivot points.

For rotation, things are a little different. Usually an object will have a natural point or axis of rotation. (Think of a model of a wrist, or a car door) So in computer models of real-world objects, it usually makes sense for the pivot location to be a property of the object being manipulated, and not the manipulator itself.

But when you do need to exactly control a pivot, how do you do it? Putting a manipulator on the pivot point can be confusing – the user doesn't know if she's moving the object or the manipulator. For this we can go back to our basic principles; we can make the handle look different in some way than the move handle, and make the pivot very visible when it is being moved.

Other mappings

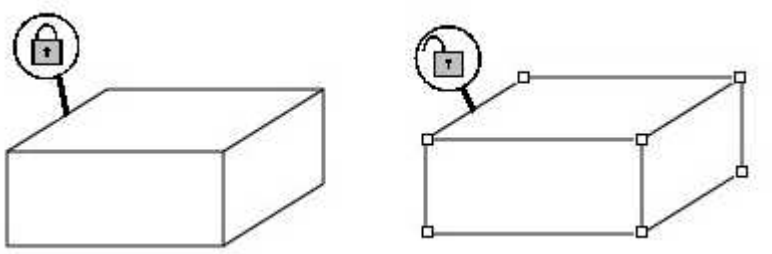
- **Constrained mappings:** This can be a point constrained to follow a line, a curve, or a surface. These mappings are straightforward, since they are 1d or 2d --- just map the handle to the part of the curve or surface closest to the cursor position.

- **Color control**: This is unusual, but in some cases it can work. You can have a little color swatch manipulator on an object. If you do this:

  - Draw the swatch in 2d, ignoring all lighting, shading, etc.

  - When you click on the swatch and a color-chooser window opens, try to make sure it doesn't open up on top of everything you are working on.

- **Options**: It is possible to have a manipulator handle control an option or a state, but this is tricky to get right. The form of the handle needs to somehow visually represent the state (unless its position is sufficient). For example, a lock/unlock toggle handle could look like a lock, in one of two states. (Use a 2d icon for this sort of representation; a 3d version will not be clearly recognizable at some angles in some scenes)

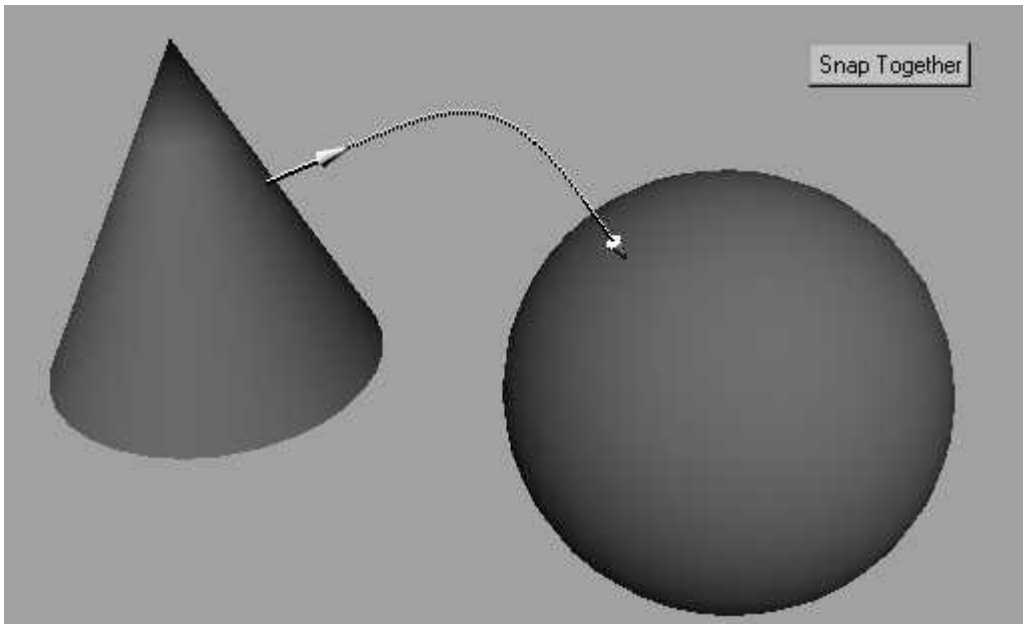


A concept sketch of a "lock" handle in a manipulator. You click on it to toggle the other handles from a locked to an unlock state. Question: should the button show the state the object is in, or the state it will switch to?

If you want to represent more than 2 states, I don't think a manipulator is an effective way of doing it. You could have a button that steps through all the states, but most people expect buttons to either toggle states or initiate actions. You could also have a small pop-up menu. I have not tried this, so I can't comment on how well it would work.

- **Actions**: Sometimes you want to put a "GO" button of some kind into a manipulator. This can be useful when the manipulator is being used to set up the parameters of a complex procedure that will take a while to calculate. First you manipulate everything into place, then you click GO.

In these cases, I recommend that the button only be visible when the action is ready to be performed. It should look like other buttons in the system, even if it is located in the 3d view. It should be at a fixed position on the screen, out of the way, rather than up on your object blocking the view of it. (If you put the GO button off to one side, or in another dialog, it will probably be missed by many users.) Let the Return or Enter key be equivalent to pressing the button.



The arrows can be dragged around to indicate start and end points for a "snap together" operation. When the arrows are in place, you click on the "snap together" button to make it happen. In this case, the manipulator is controlling the parametric inputs of a complex function.

Temporal modes in a manipulator

Sometimes there is enough functionality in a manipulator that you can't reasonably have a handle for every function. In that case there may be a strong temptation to use temporal modes to allow you manipulator handles to support multiple functions, or to swap out one set of handles for another set doing a different job.

Clutched modes

If the additional functionality is just a variation on the existing functionality, used **clutched modes**. A clutched mode is a mode that you can switch into by pressing a key, such as Shift or Control. As soon as you let go of the button, you switch back to the regular mode. Clutched modes are superior to **switched** modes (such as tools) because they don't require the extra mouse movement; they take advantage of 2-handed input, and you don't get mode errors. People don't forget they're holding down the Option or Shift key, because they have the constant physiological feedback from their fingers.

Clutched modes are used very effectively and consistently in programs like Photoshop and Illustrator. In these cases, the mode is always a modification of the existing handle functions. And these modifications are constant across tools. For examples, Shift always means "constrain to be proportional", and Alt means "put the pivot at the center".

A disadvantage of clutched modes is that they are invisible to the user. To help with this problem, consider having the appearance of the manipulator change in some way to reflect the state of the modifiers. So when you press the Alt key, for example, you see the pivot point jump. This is useful because people often know there is a modifier, but they are not sure which modifier to use, so they try pressing different ones. It's nice to give them a clue as to what is going to happen. Also, it contributes to the nice sense of interactivity in the program.

The meaning of your modifiers should be consistent with those in other programs used by your target users.

Switched Modes

If you must support multiple functions in one manipulator, one option is to display a set of radio buttons that show all the modes. These could be in a dialog, or floating in in the view. This display should have these properties: first, it should show all the possible states. Second, it should show the currently selected state. Third, it should be clear what each mode is, and functionality should be grouped by the user's **task**.

It is tempting for engineers to groups operations together that are related by the internal structure of the data. But that's the wrong approach. Find out what tasks and sub-tasks your users are likely to be doing, and group together the manipulator functionality needed for that task. Then name that manipulator mode for that task.

General recommendations for manipulator behavior

- **Use isomorphic mappings**: Once you click on a manipulator handle and begin to drag, every place on the screen should map to a specific value in the manipulator. What I mean is, if you drag from A

to B you should get the same result, no matter what route you take to get there. And, if you drag back to A before you let go, the manipulator should return to its original value.[4]

Rate-control manipulators are also a bad idea. These are manipulators where your mouse motion controls how fast the action happens, rather than how much. You return the mouse to its original position to stop the motion. (Think of how your gas pedal works; you hold it still to provide constant motion.) Rate control is very difficult to do with a mouse, which is an isotonic input device. If you had a joystick as a controller, it would be a different story.

- **Provide numeric feedback**: In cases where eyeballing is not accurate enough, consider providing numeric feedback too. If you do, draw the numbers in screen space, not 3d, and make sure that they are readable on any color background. (Or draw your own background just for the numbers). I do not recommend using XOR for this; on textured backgrounds it becomes almost impossible to read. Numeric feedback needs to be provided close to the user's locus of attention, or it won't be noticed.

- **Constrain interactively**: In cases where the value being controlled by a manipulator handle is constrained, you should constrain the handle movement to match. For example, if the object can't get any bigger, then the scale handle would seem to run into a wall. (You might even add a little sound effect). This feels better than allowing the handle free motion, while constraining the value. The latter practice gives the impression that the manipulator is broken.

- **Provide snapping**: Manipulators should snap to commonly-used values or positions. (For example, 45 or 90-degree rotations, or back to the initial position). Which values should be snapped to (if any) depends on the particular application, of course.

- **Always provide undo**.

[4] In recent years there has been some research about using non-isomorphic mappings for many of these tasks, especially rotation. However, this work seems to mostly be done on 3DOF or 6DOF input devices. For an example of such a study, see Ivan Poupyrev , Suzanne Weghorst , Sidney Fels*, Non-Isomorphic 3d Rotational Techniques*, Proceedings of the CHI 2000 conference on Human Factors in computing systems, April 2000.

Making sure your manipulators are great

Usability is in the details.

In the section on mapping, we talked a lot about making the manipulators feel natural and responsive. You can design everything right, but you will never know if your manipulator has that slick quality until you test it with real users doing real tasks.

Every manipulator I have tested in this way was improved, usually by minor adjustments to the mapping or the form-factor. (I only wish we'd had the time to do this kind of work on all the manipulators we've released!)

When you are planning your development time for a manipulator, I recommend scheduling as much time for tweaking as you do for initial developing, if not more. Consider it part of the basic design, not a nice extra.

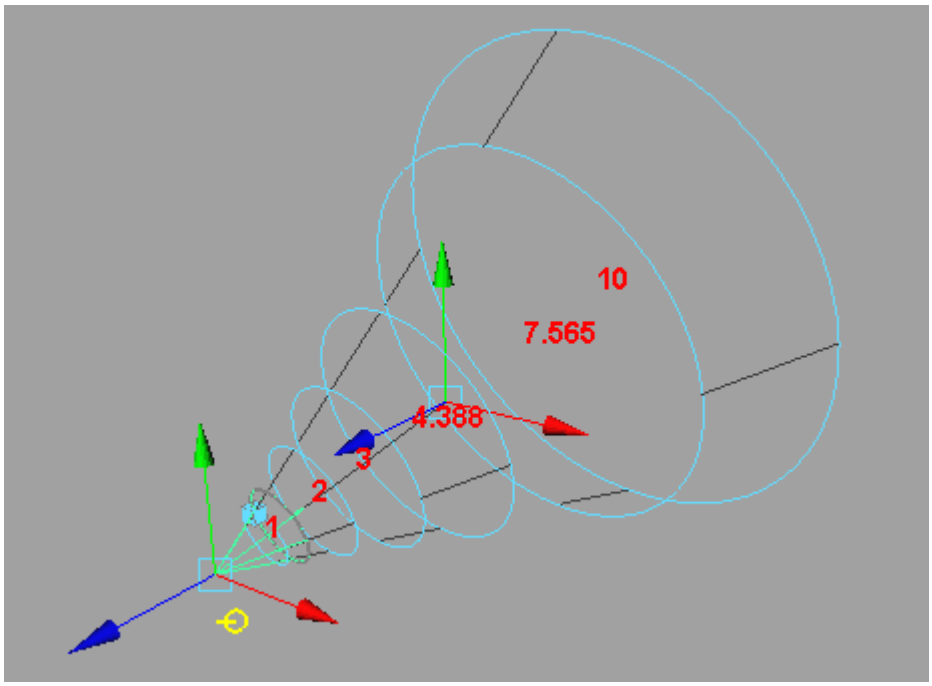
Be aware that different user populations will react very differently to the same manipulators. This is especially true between users who have experience in 3d, and those who have only learned to work in 2d (such as graphic artists). Just because your manipulator tests well with your engineers, that doesn't mean it will test will with your target users --- unless your target users are engineers.

Consider working with a designer to make your manipulators beautiful. In our testing, we've noted that aesthetically pleasing designs are perceived to be more usable, even when the behavior is no different.

Most importantly, always think about the task that your users are trying to accomplish, not the data you are trying to manipulate. This should be governing your design choices.

Exercises

Heuristic evaluation: What's wrong with this picture?

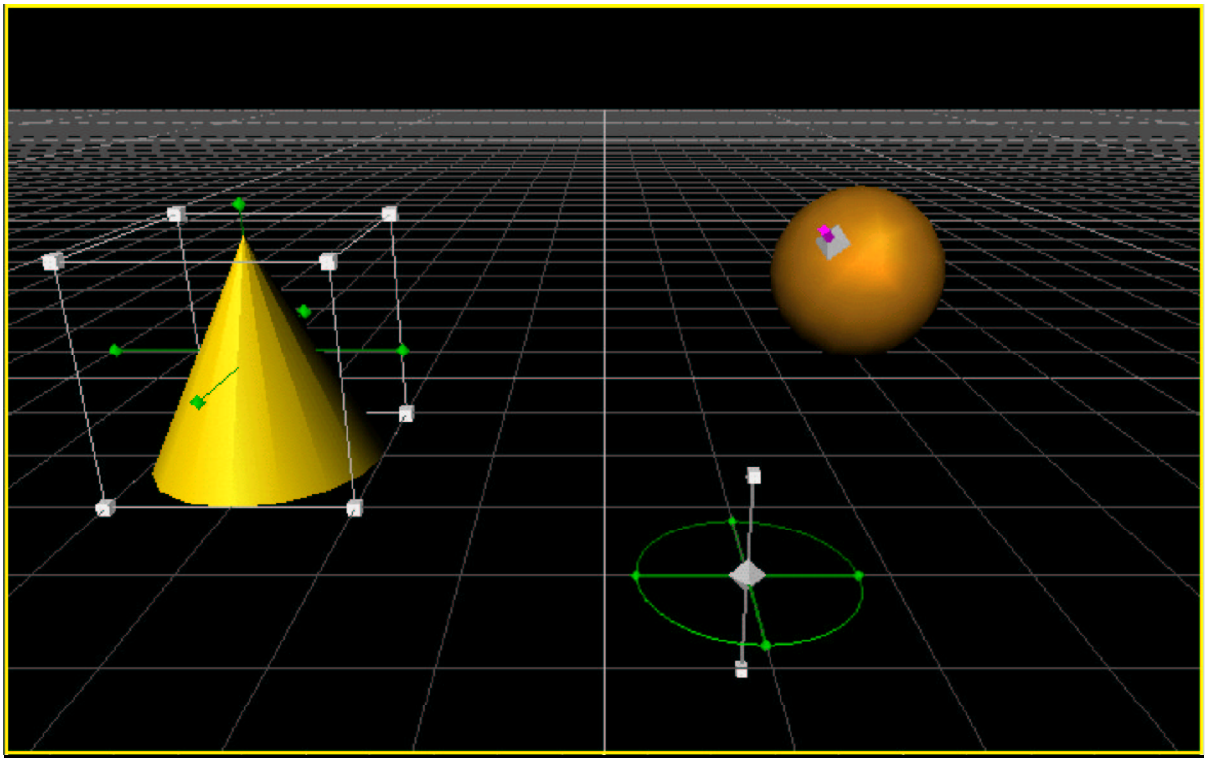


The Spotlight manipulator in Alias Maya 3.0. This has a number of problems in its design. How many can you find?

Design: Build a better deforming manipulator

Implementing Suites of 3D UI Tools

Paul Isaacs
pauli@pauliface.com



Implementing Suites of 3D UI Tools

The previous module of this course presented techniques for the design of individual manipulators. This module presents approaches for designing and implementing suites of 3D tools to be used together in an application setting.

Limitations and context

Device limitations

As in the previous module of this course, the techniques are targeted to a system with just a keyboard and a mouse with one button. Moreover, the only keyboard keys used are the control and shift modifiers. This frees all other keys for use by other parts of the application.

Task set

The types of 3D manipulation discussed herein are restricted to translations, rotations, and scales. The operations are applicable to all kinds of objects, such as groups, whole shapes, sets of points, cameras, and lights. Other types of tasks are not addressed (e.g., shape deformation, color editing, animation paths). The techniques discussed here can be extended to apply to these other kinds of tasks.

CosmoWorlds

The tools presented in this module are from a program called CosmoWorlds. CosmoWorlds is a web−oriented authoring tool, and the modeling tools are specifically oriented towards the creation of low−polygon scenes. CosmoWorlds was created by Cosmo Software, a now defunct subsidiary of SGI. CosmoWorlds is currently owned by Computer Associates and is no longer commercially available. However, the techniques and principles of its design and implementation serve as a useful example for anyone creating a 3D user interface.

Overview and goals

In an application, you'll want to move many different types of objects in a variety of ways. The ideal suite of tools allows you to perform these tasks in a consistent way. This talk is organized in four sections, corresponding to goals that move toward that end:

- **Leverage scene geometry**
  Geometric objects can serve as guides for motion, placement, or alignment. A 3D scene may contain multiple lines, planes, directions and coordinate systems. Multiple views present an even greater range of options. Make the tools leverage these spatial landmarks so that users can turn the scene itself into a tool.
- **Be consistent**
  Different tools don't necessarily require different paradigms. They should share consistent behavior and a consistent visual language. Also, different types of objects don't necessarily require different tools. A consistent underlying software architecture makes your application easier to write and easier to use.
- **Provide a variety of tools**
  Different tasks require different tools. Provide a suite of tools instead of forcing all functionality into just one or two.
- **Avoid temporal modes**
  An individual manipulator helps avoid temporal modes because multiple handles simultaneously present a variety of functions. Similarly, a set of tools, simultaneously visible, provides access to the full variety of tasks without resorting to temporal modes.

Leveraging scene geometry

This section shows how geometry in the scene can be used to enhance the functionality of 3D tools. The information is presented in three parts:

- Introduction to landmarks
- Implementing landmarks
- Follow−the−cursor

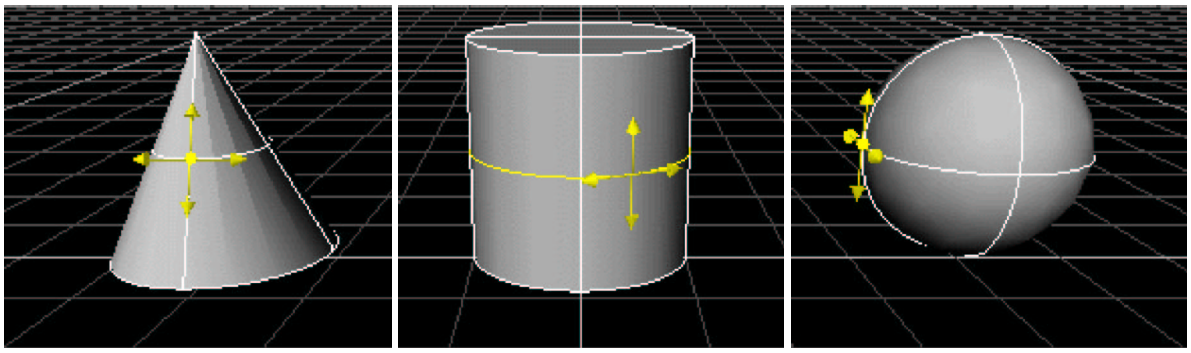
Leveraging scene geometry 1:

Introduction to landmarks

Selection−aligned manipulators: The previous module discussed how handles can be arranged into manipulators that center about an object. These enable tasks that relate to the coordinate frame and bounding box of the object itself. The papers that are included in the final section of the course notes elaborate further on how to create such manipulators.

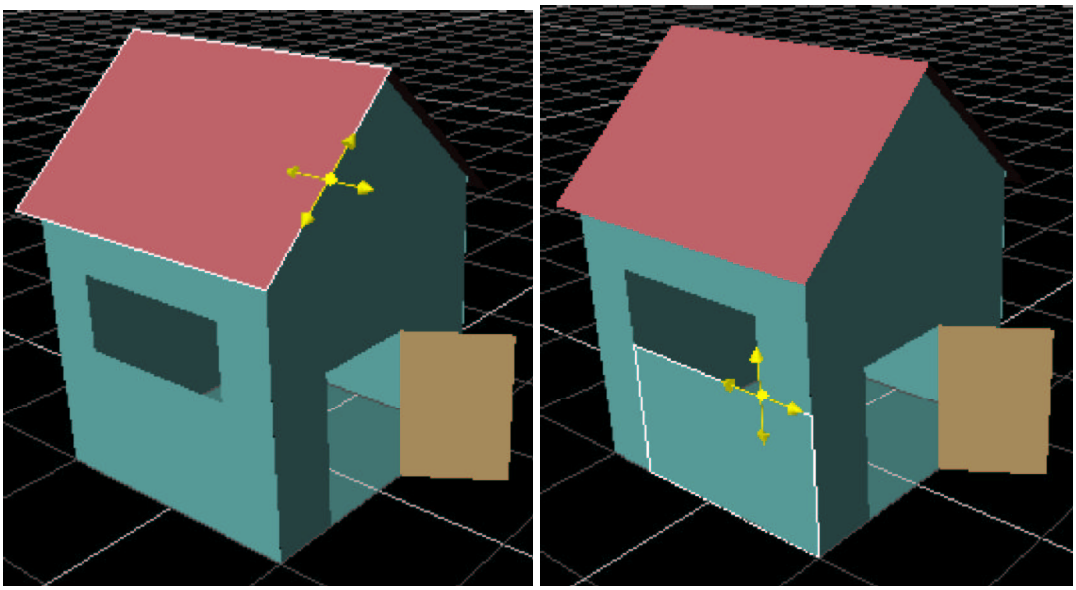
Grids and spaces: Some modeling applications provide tools that allow you to use grids for snapping and alignment; and some allow you to change the alignment of manipulators so that you are working in a variety of spaces, such as local space, world space, and the parent space of an object.

Introducing landmarks: Tools that let you work directly with features and landmarks in a 3D scene are less common. Each object in a scene may have several points, lines, curves, and planes of interest. In addition, each of these features has a natural set of related directions. When these features and directions are accessible through the 3D tools, new opportunities arise for placing and moving objects.



landmarks on primitives

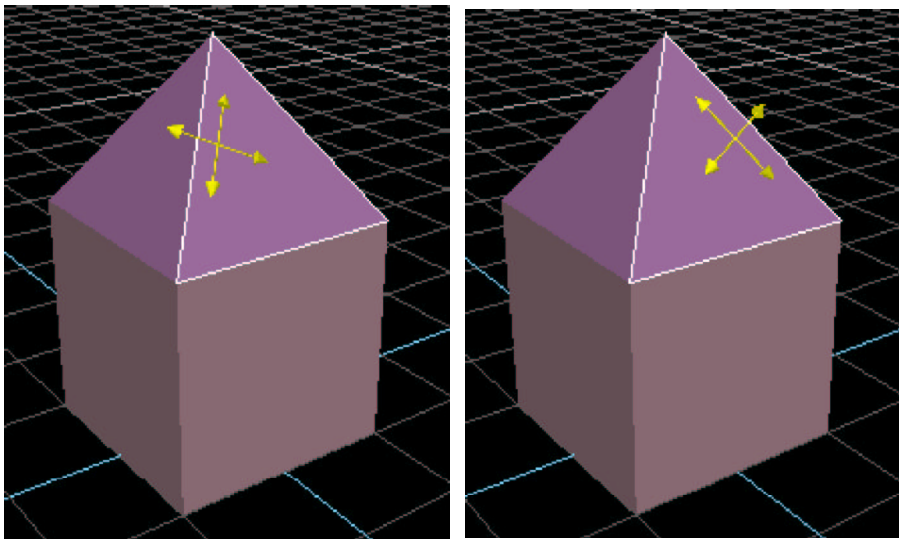
Landmarks on primitives: Each type of primitive object has a natural set of landmarks. For example, a cone has a ring at the base. It also has a mid−level ring and lines that represent the intersection of the cone with its xy, xz, and yz planes. The intersections of these lines and circles, as well as the center of the base, comprise a set of points that also serve as landmarks.



landmarks on polygonal objects

Landmarks on polygonal objects: Every polygon in a polygonal object has a set of landmarks. These are the vertices, the edges, and the midpoints of the edges. Together they present landmarks over the entire topology of the object.

Landmarks on other types of objects: CosmoWorlds does not have a way to model patches or spline surfaces; but these, too, have natural landmarks. Every time you create a new type of object, there is an opportunity to present a new set of landmarks.



directions align to the nearest edge

3 related directions: The useful directions at any landmark relate to the model itself. One direction is the normal to the model at that point. The other two are perpendicular to each

other and lie in the normal plane. Of these two, one aligns with the line or circle closest to the point in question. The last direction is orthogonal to the first two.

A coordinate frame at each landmark: The location of a landmark, together with the 3 orthogonal directions, forms a coordinate frame. These coordinate frames form the basis of landmark based interaction.

Leveraging scene geometry 2:

Implementing landmarks

An object oriented approach: The implementation of landmark finding involves two tasks: finding the landmarks and presenting good feedback to the user. In an object oriented application, this burden is shouldered by each type of object individually. The assumption is made here that each type of selectable object stems from a base class of object upon which virtual methods may be defined. Thus, a CubeObject and a PolygonalObject have independent implementations, but the required methods can be invoked on either object (cast to the base class), without knowing which type of object it is.

Finding landmarks

Finding the landmarks with getLandmark(): Each type of object needs to implement a getLandmark() method. Typically, a picking utility is invoked first to determine what object and location were beneath the cursor. Next, the selected object's getLandmark() method is called to get the details about the landmark. In order to allow users to disable or enable snapping to the various points and lines on the object, an isSnapping flag is passed into the method. The selected landmark location and directions are passed back as a result.

Snapping to features: Each type of object knows where the key features lie and how they are oriented. For example, the cone knows the locations of the key points, lines, and rings. If snapping has been requested, the object needs to determine whether or not the picked point is within a certain threshold of a key feature. These calculations are best done in screen space, with the threshold expressed in pixels. Otherwise, features that are farther from the camera are harder to pick.

Priorities in picking: Points should be easier to find than the lines on which they lie, so they should be given preference and checked first. If the cursor is not close enough to any of the key points, then the lines and circles should be checked. Failing this, the unsnapped location below the cursor should be used. In practice, it is best to use a larger pixel threshold for the points than the lines and circles.

When snapping is disabled: When snapping is disabled, getLandmark() skips the snapping step and just finds the correct directions for the point that is directly below the cursor.

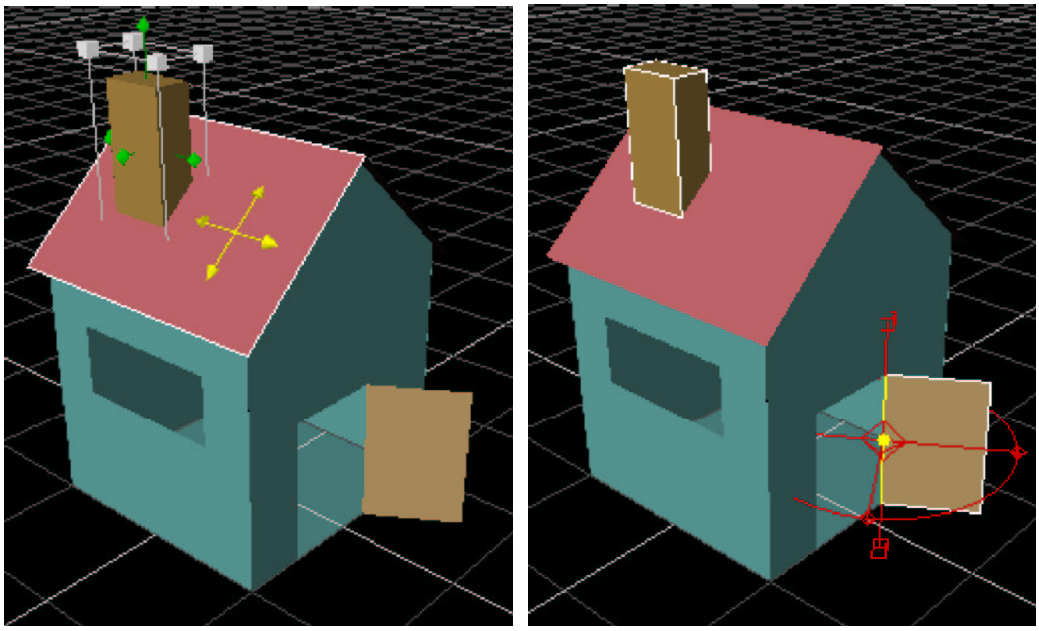
Displaying feedback

Feature feedback versus landmark feedback: There are two parts to the feedback information. The first is a display of the features on the object. The second is a display of the

landmark's location and directions. The first is handled by the selected object, the second by the 3D UI tool.

Displaying feature feedback with showFeatureFeedback(): Each type of object must implement showFeatureFeedback(). This method takes information about the selected landmark as input. In response, it displays feedback that reveals its key features. Once again, the object itself has the best information; it knows how to draw its own feature feedback. In general, the object displays all the key features in one color (white in CosmoWorlds) and displays any snapped feature in a different color (yellow in CosmoWorlds). The object should **not** display the landmark feedback (location and directions). This is left to the 3D UI tool, since different tools may wish to represent this information in different ways.

Keep showFeatureFeedback() separate from getLandmark(): It's best not to just show feedback as a side effect of getLandmark(). This leaves it up to the 3D UI tool that's in use to decide whether or not the feature feedback should be displayed.



landmark feedback with the Plane and Wheel Tools

Displaying landmark feedback: The display of the landmark location and directions is left to the 3D UI tool. Two examples are illustrated above. In the first, a Plane Tool is being used to determine a plane of motion. It calls getFeedback(), showFeatureFeedback(), and then draws a lightweight indication of that plane. In the second, the Wheel Tool is being moved so that it aligns with a feature in the scene. Its feedback is the Wheel Tool itself, which moves and orients to indicate the new position.

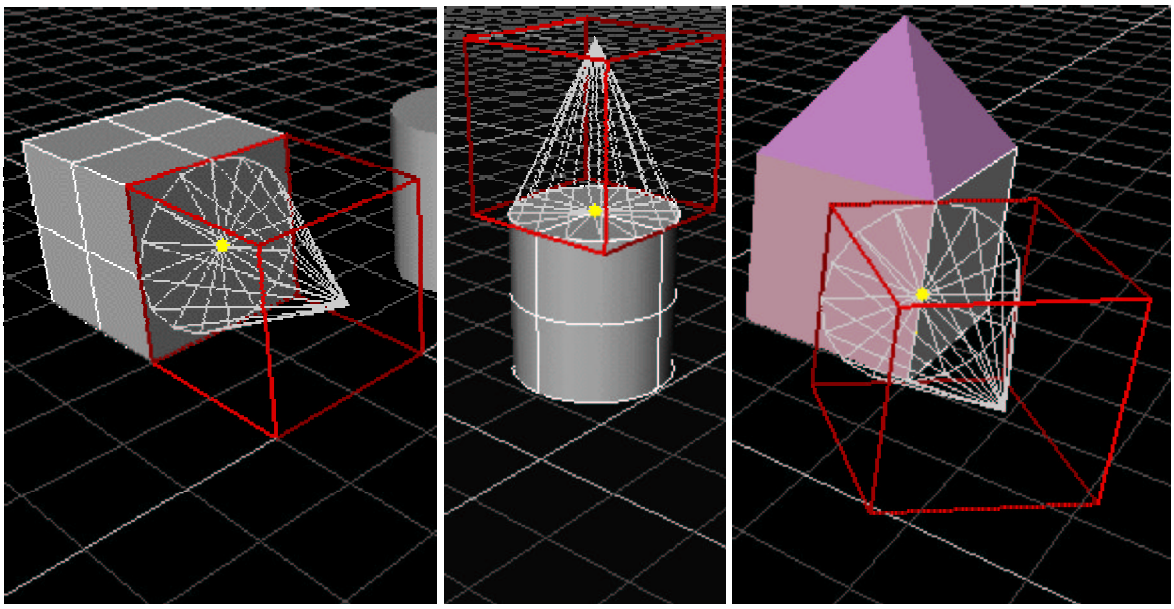
Leveraging scene geometry 3:

Follow−the−cursor

Reaping the benefits: The above infrastructure is a lot of work. But it lays the groundwork for an application to implement numerous new tools. Since these tools are all based on the same infrastructure, they share a common look and feel throughout the application.

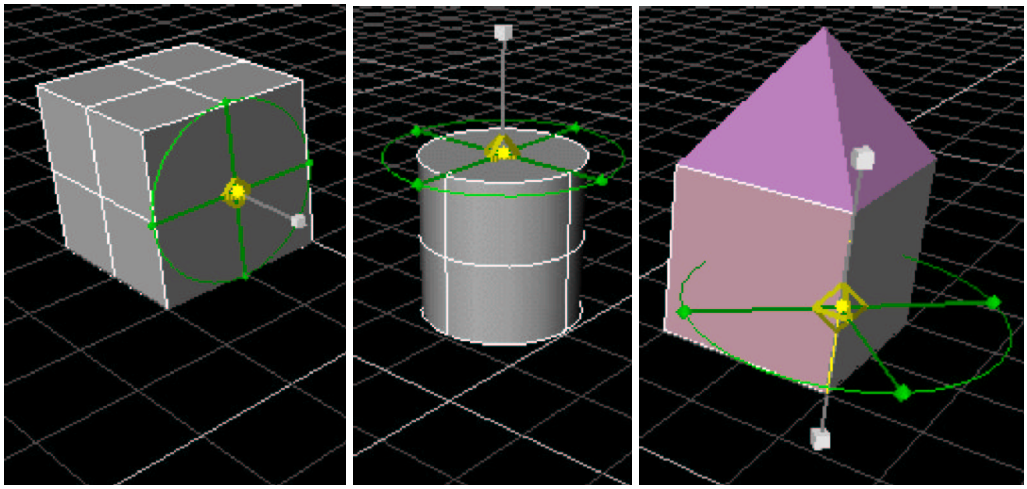
Follow−the−cursor defined: ***Follow−the−cursor*** is a term used to describe the act of finding the landmarks below the cursor as it is moved across the scene. It's done by calling getLandmark() and showFeatureFeedback() each time the cursor is moved to a new location.

Placing new objects: The operations of import, paste, and shape creation all introduce new objects into a scene. Sometimes it's useful to place new objects at the origin. When pasting, it might be desirable to paste over the location of the original object that was copied. And when importing, there might be a location already in the file that should remain unchanged.



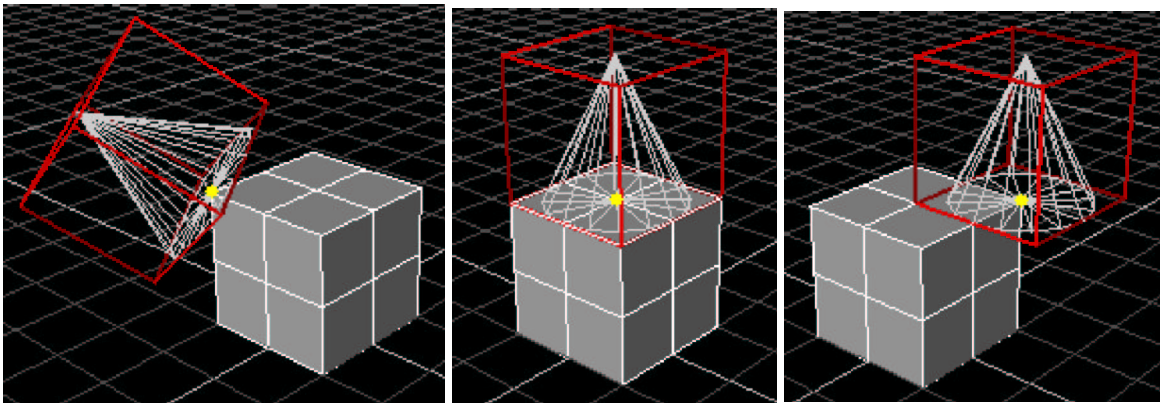
placing a new cone with follow−the−cursor

Using follow−the−cursor to place new objects: Follow−the−cursor provides another way to introduce objects into the scene. Landmarks provide convenient locations and orientations for placing new objects. Using follow−the−cursor, a new object can be moved across the surface of objects in the scene, prior to committing to a location for the object.



relocating a Wheel Tool with follow−the−cursor

Using follow−the−cursor to relocate existing objects: Some tools in CosmoWorlds have white octahedral handles. Throughout the application, this type of handle may be click−dragged to move an object in follow−the−cursor style. Once this handle is clicked, and for as long as it is dragged, the tool relocates and reorients the object to align with the selected landmark.



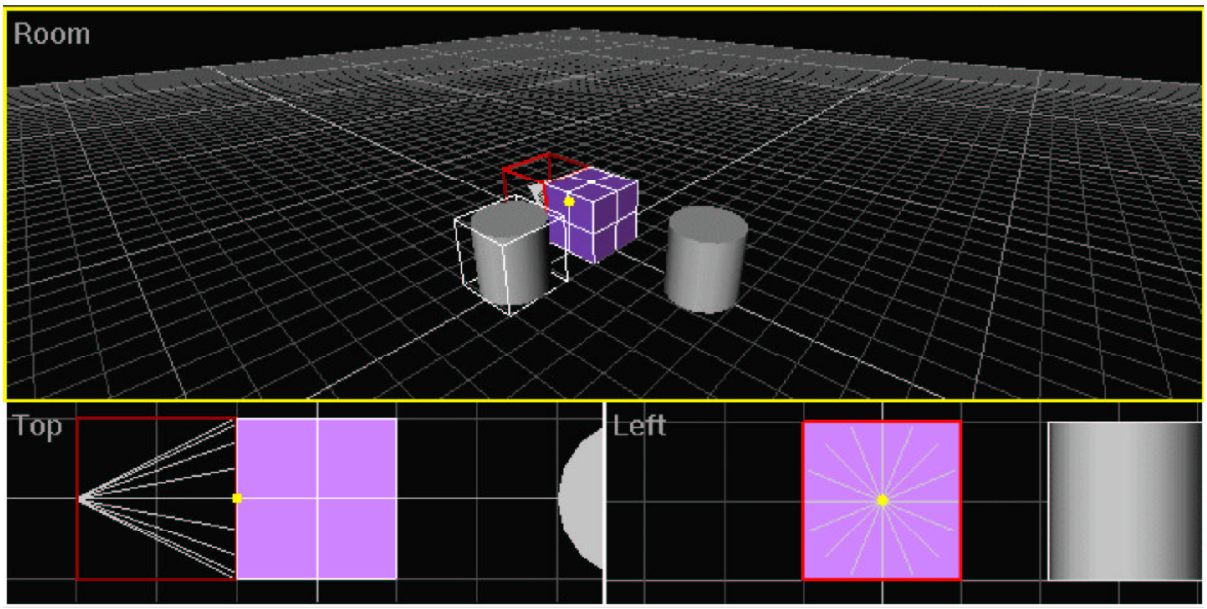
(a) *(b)* *(c)*

freezing the orientation during follow−the−cursor

Freezing the orientation: The landmark directions at a desired location don't always match the desired orientation. For example, in figure (a) above, the orientation at the corner of a box is directed normal to the vertex. It is a diagonally outward direction and is not parallel to any side of the box. But what if you want to position an object at a corner of the box, facing upward? Instead of doing this in two operations (place the object, then align it some other way), tools can freeze the orientation and save it for later, all during one motion of the mouse. This is achieved using a clutched mode as discussed in the prior module. When the user presses the control key, the tools can save the current orientation and reuse it as long as the key is depressed. So, the selection can be aligned by moving it over the top of the box, as in figure (b), freezing the orientation with the control key, and then moving it until it snaps

into position at the corner, as in figure (c). When placed, it will have a location at the corner, but an orientation that aligns with the top of the box.

Using follow−the−cursor for translation tasks: In CosmoWorlds, a Plane Tool allows a selected object to be moved parallel to any plane in the scene. This permits motion in directions that do not align with the selection's bounding box. The desired plane is selected with follow−the−cursor; prior to clicking the mouse button, the Plane Tool displays axes that align with the feature below the cursor. Once that feedback indicates a desired plane of motion, the user can click−drag to move the selection parallel to that plane. Later in this presentation, we'll show how the Plane Tool uses clutched modes to permit motion normal to the plane or parallel to an edge.



placing a cone on a surface that is occluded in the primary view

Multiple views show you more: Different landmarks may be hidden and visible in different views of a multi−view application. Since follow−the−cursor allows you to find landmarks in any view, you can sometimes avoid changing your perspective camera's view (to reveal a hidden landmark) by placing the object in one of the other windows.

Being consistent

This section shows how consistency is achieved both through a uniform look and feel, and a well designed application architecture. Techniques are presented that break down the implementation of 3D user interfaces into simpler parts; parts that interact to form a complex and rich set of tools. The information is presented in the 9 parts:

- Look and feel
- Underlying architecture
- Command−based architectures
- Dragger + Commands + Selection = Manipulator
- One set of commands for all draggers
- More on motion commands
- One set of commands for all selections
- Motion commands, snapping and constraints
- Motion commands make it easier to write new tools

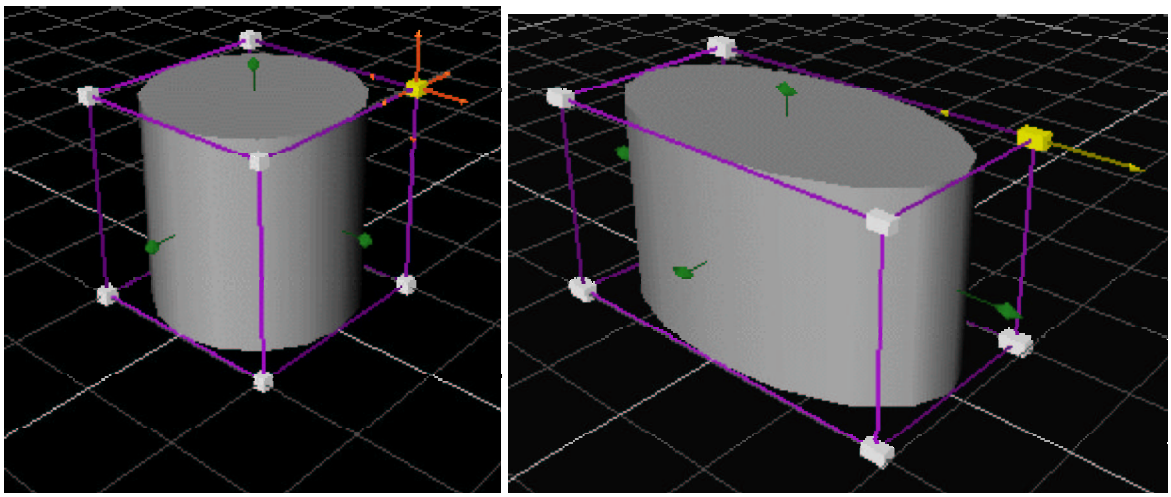
Being consistent 1:

Look and feel

The first module of the course stressed the importance of a consistent graphical language, as well as consistent use of clutched modes. The tools presented here share this goal.

Graphical language: Throughout the tool set, white cubes indicate scale handles, green spheres are rotation handles, and white octahedra are follow−the−cursor handles. Translation is generally indicated by planes whose perimeters highlight and/or a pair of yellow axes. You can of course choose your own graphical language; just be consistent.

Clutch modes: Shift−dragging (holding the shift key while dragging) is used to toggle between constrained and unconstrained modes. This is such a widespread convention that I don't recommend straying from it. Control−dragging (holding the control key while dragging), on the other hand, has no universal meaning. In CosmoWorlds, it can serve a variety of functions depending on the tool. Beyond this, if you feel you've run out of available modes and want to use another modifier key, chances are you're trying to pack too much functionality into one tool; you should consider breaking it down into two separate tools.



choosing a 1D scale direction with gesture−selection

Gesture−selection: Sometimes there is more than one "good" direction a tool might want to use. For example, shift−dragging a scale handle of the Box Tool initiates 1D scaling. But which direction is best? One approach is to provide separate handles for each direction, but this could crowd the manipulator or limit the number of available directions for a given point of view. An alternative is ***gesture−selection***. When the handle is shift−clicked, three axes are displayed simultaneously. By convention, the color orange is used to indicate a choice for

gesture−selection. As soon as the user moves the cursor, the direction of cursor motion is noted and compared, in screen space, to the available directions. Whichever one most closely matches the gesture becomes the direction of scale. This same convention is used consistently throughout the tool set, so if the user sees these orange arrows, he knows that it is time to make a gesture selection.

Being consistent 2:

Underlying architecture

There is more to being consistent than just the outer look and feel of your 3D tools. The underlying architecture of your application has a huge effect on both how easy it is to program and how the result feels to your users. A well−designed architecture is object oriented and shares code in a consistent way, and this ultimately is reflected in the usability of the application.

Follow−the−cursor: We've already seen how one pair of methods, getLandmark() and showFeatureFeedback(), can be implemented in an object oriented way. These methods then enable a multitude of tasks to rely on follow−the−cursor. The result is shared code and consistent look and feel.

Inside the manipulators: Different 3D tools share code to achieve many of the design benefits described in the previous module of this course. Each bullet below describes a common technique and the classes that are used to encapsulate them.

- Different styles of motion mapping (i.e., mapping 2D cursor motion to translations, scales, and rotations) are used repeatedly in the toolset. A set of classes called ***Projectors*** encapsulates these different mappings.
- Round manipulators and round feedback appear squished when rendered within a nonuniformly scaled or sheared space. A class called the ***AntiSquish node*** is used to make such objects round again. The AntiSquish node analyzes the current transformation matrix and appends a new matrix that undoes the squishing, while leaving the translation and rotation intact.
- Some tools need to surround a selection. These are always initially scaled as a canonical 2x2x2 unit cube. The ***SurroundScale node*** calculates the bounding box of the selection. It then appends a matrix to the current transformation matrix that translates and scales the tool to fit around the selection.
- Handles need to be kept within a reasonable range of sizes. The ***MinSizeMaxSize node*** contains parameters, expressed in pixels, that specify the largest and smallest permissible size on screen . It analyzes the current transformation matrix to determine the size of the handle in screen space, then appends a scaling matrix that keeps it within the acceptable range of sizes.
- Some tools need to appear different in different viewports. For example, you might show certain feedback only in the view that currently has the user's focus. Rather than have different scene graphs on hand to display in the different views, tools employ a ***ViewportSwitch node***. This node has different subgraphs to display depending on which view is being rendered.

Resources for "Inside the manipulators": The first three techniques are described in more detail in the following resources. The others are not covered, but should be relatively straightforward to implement.

- The three papers included at the end of these course notes.
- Open Inventor source code, which is available at the following locations:
  http://oss.sgi.com/projects/inventor/
  http://www.studierstube.org/openinventor/
- Any new information I find or create after these course notes go to press will be at:
  http://www.pauliface.com/Sigg02/index.html
- Me! I'll help you out, within reason. pauli@pauliface.com

Being consistent 3:

Command−based architectures

Use a command based architecture: Using a command based architecture in your application is the one single thing you should do to keep it consistent, clean, and bug−free. We'll see how commands help for 3D UI in the next part of this section, but first I'll describe a command architecture in general terms.

Commands: A ***Command*** is a base class with two methods, do() and undo(). When do() is invoked, the command is performed. When undo() is invoked, the command returns things to how they were before the command was performed.

Command parameters: Each subclass of Command contains parameters to assist the command in performing do(). For example, an ***AddCone*** command has translation, rotation, and scale parameters to indicate where the cone object should go. It also has a parent parameter which specifies the group node under which it is placed. When you call do() on the AddCone command, it inserts the cone object in the scene graph, below the parent, then sets the cone object's transform fields accordingly. When you call undo(), it removes the cone from the scene.

CommandManager: The CommandManager is a separate class that takes care of executing commands. The application has just one instance of CommandManager. When a user performs an action, a command is created and submitted to the CommandManager. Later, at the appropriate time (usually just before the next render occurs), the application tells the CommandManager to perform all pending commands.

Every action must occur through a command: No matter what the user does, whether through a button, key, or mouse, the action must be performed by using a command.

Benefits: The above architecture yields the following benefits:

- **Consistency:** The same type of command can be called through different means. For example, a keyboard shortcut, a 3D tool, or an icon button can all execute the same kind of command. The code for the command gets reused and the result of the command is consistent regardless of how it was initiated.
- **Infinite undo/redo:** The CommandManager can keep a list of all commands that have been executed once the application starts up. Infinite undo is achieved by calling undo() on each command in the list, in reverse order. Redo() is achieved by calling do() once again, in forward order.
- **Logs enable debugging:** The CommandManager can write the commands and their parameters to a file and update that file each time a command is performed. This produces a log of all user actions. If the program crashes, the user can submit the log file with his bug report. The log file then makes it a simple matter to reproduce the

bug and determine the reason for the crash. Each command in the log is fed in turn to the CommandManager, and at the last command the error is reproduced.

- **Regression testing:** By saving all the error logs from past bug reports and running all of them when the code base changes, engineers can make determine whether old bugs have crept back into the code.
- **Monkey testing:** This is a term that invokes the image of a monkey banging at the keyboard. You can write a utility that randomly generates commands and parameters for those commands, then hands them to the CommandManager one after the other. This will throw a whole variety of unlikely combinations at the application and reveal problems that might not turn up in ordinary user testing.

Being consistent 4:

Dragger+Commands+Selection = Manipulator

In the previous module, a manipulator was defined as follows:

A manipulator is a visible graphic representation of an operation on, or state of, an object, that is displayed together with that object. This state can be controlled by clicking and dragging on the graphic elements (handles) of the manipulator.

This overall description can be broken down further within the context of a command based architecture.

Dragger: A dragger is a visible graphic representation that can be clicked and dragged to generate one or more types of ***MotionCommand***.

MotionCommand: A MotionCommand is a subclass of Command that specifies a rotation, translation, or scale.

Selection: A ***selection*** is the entity in the application that is currently selected. It can be a primitive, polygonal object, group, light, etc. It can also be a ***subselection***, such as a set of points, lines, or polygons.

Selections watch for execution of commands: When a selection is established, it can register with the CommandManager to be notified via callback when a new command is executed. The selection is free to respond however it wishes.

Putting it all together: The dragger is a visible graphic representation. When you click−drag the dragger, it generates a command. When the command is executed, the selection responds with a change of state. Taken as a whole, this forms a manipulator.

Being consistent 5:

One set of commands for all draggers

Isolating the dragger's tasks: Decoupling the commands and the selection from the dragger saves work, shares code, and makes things consistent. The MotionCommand classes only need to be written once. All a dragger needs to know how to do is generate them. Granted, this is a tough task, but at least the task has been isolated. The dragger does not even need to know what kind of selection is listening. Different draggers invoke commands based on different handles and different mouse mappings, but the result is always one of these commands.

An exhaustive list: The 3D UI tools presented in this module are all implemented with the same eight MotionCommand classes:

- translateInLine
- translateInPlane
- scale1D
- scale2D
- scaleUniform
- rotate1D
- rotate3D
- rotateAndTranslate

Parameters: The motion command parameters describe the motion to be applied to the current selection. For example, TranslateInLine has parameters describing the line origin and direction, as well as the distance traveled along the line. The full description of the line is used instead of just a simple translation triplet, because snapping can be implemented more effectively this way. Advanced snapping techniques are out of scope for today's discussion, but for now keep in mind that a snap tool can only draw feedback of the line of travel if a full description of the line is available.

Coordinate frame parameter: Each motion command has a parameter describing the coordinate frame in which the command is expressed. In this way, a selection that responds to the command knows whether or not it needs to transform the command into its own local space before applying the command.

Being consistent 6:

More on motion commands

Every motion command contains information in addition to what's available in the base Command class.

Start, Move, Finish: Since motion commands are based on a click−drag−release action, they contain extra information indicating which stage of motion the command represents. Selections that listen for motion commands typically save their current state upon ***Start.*** When responding to ***Move*** and ***Finish***, they change their state based on the given motion, relative to the starting state.

Matrices: Motion commands contain methods that return matrices calculated based on their parameters. The ***getMotionMatrix()*** method returns a matrix representing the command as expressed in the command's local space. For example, the TranslateInLine command returns a translation matrix. The ***getLocalToWorldMatrix()*** method returns a matrix that transforms between the command's local space (as given by its coordinate frame parameter) and world space. The ***getWorldToLocalMatrix()*** returns the inverse of the localToWorld matrix.

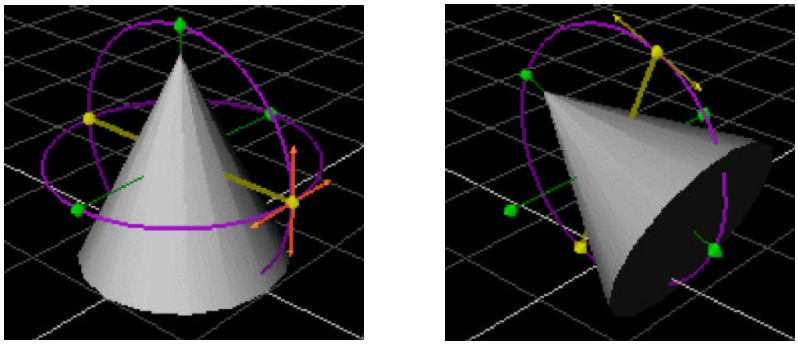
Being consistent 7:

One set of commands for all selections

Isolating the selection's tasks: Each type of selection needs to implement an appropriate response to each of the eight types of motion command. Again, this is significant work. But these eight responses enable a selection to be controlled by any type of dragger. The selection does not need to know what kind of dragger initiated the command, nor is any new code required when new types of draggers are added to the application.

Transforming the **mo**tion **c**ommand to **l**ocal **s**pace: Each type of selection needs to transform the motion command's motionMatrix into its own local space in order to process it correctly. To do this, the selection needs the motion command's three matrices, as well as its own localToWorld and worldToLocal matrices. The equation for transforming the motion matrix is as follows:

[localMotionMatrix] = [selectionLocalToWorld]*[commandWorldToLocal]*
[motionMatrix]*
[commandLocalToWorld]*[selectionWorldToLocal]

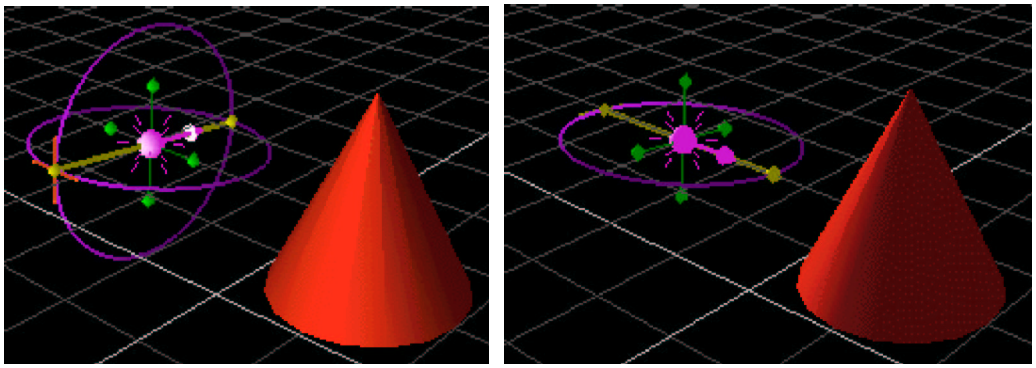


rotating a TransformObject with the Box Tool

How **t**ransforms **r**espond: A ***TransformObject*** is used to contain primitive, polygonal, and group objects. It interprets motion commands by updating its rotation, translation, scale, scaleOrientation and center fields. TransformObjects respond to motion commands by adjusting their fields to represent the incremental change represented by the localMotionMatrix:

- Calculate the localMotionMatrix (as above)
- Append localMotionMatrix to a matrix representing the field values saved from when the command was initiated.
- Decompose the combined matrix into the five fields to get new field values

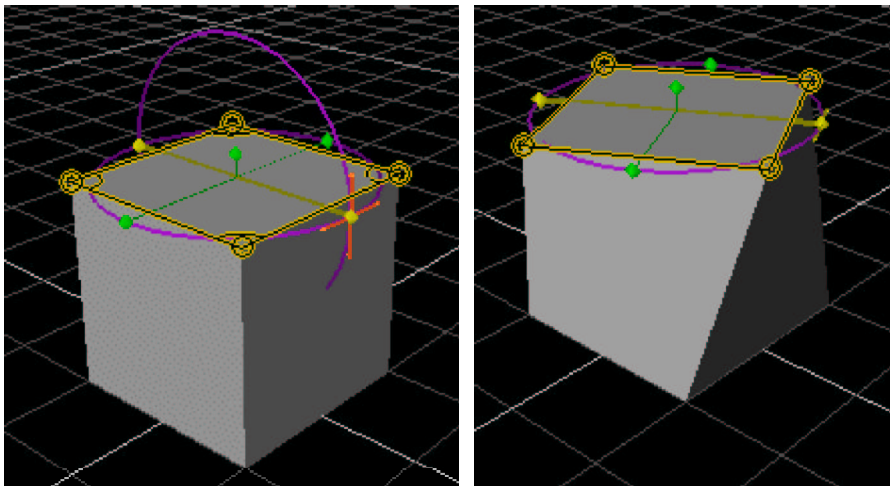
Decomposing the matrix is tricky, but the Open Inventor source code contains a method to do this for you. The method to look for is in SbMatrix.c++, and it's called SbMatrix::factor().



rotating a LightObject with the Box Tool

How lights and cameras respond: Lights and cameras have their own object types. When they are selected, the response is similar to a TransformObject. The differences are as follows:

- Neither lights nor cameras contain a scale field. So this component is ignored when the final matrix is decomposed.
- Point lights contain no rotation, so they must throw out any rotation component.
- The parameters have different names. Position instead of translation, direction instead of rotation, etc.



rotating a set of points with the Box Tool

How selected sets of points respond: When a polygonal object is being edited to alter its shape, the selection is specified as a set of points. For such selections, the response to a motion command is not to edit the fields of the Transform that contains the object, because that would affect all points equally. The object instead responds thus:

- Calculate the localMotionMatrix (as above)
- Transform only the selected points by that matrix
- This has the net effect of moving the selected points relative to the other points in the shape. Even when the matrix contains rotations or scales, this is the correct behavior. The points move to positions that reflect the applied rotation or scale.

Being consistent 8:

Motion commands, snapping and constraints

Snapping and constraints are used to modify a transformation subject to certain criteria. This is a big topic; I touch upon it briefly here to show how it fits into a command based architecture..

Constraint classes register with CommandManager: Just as selections register to be notified when new commands are executed, so do any classes that wish to constrain motion.

Constraints are notified first: When a motion command is executed, the CommandManager notifies constraint classes before it notifies selections.

Constraints can modify the motion command: Constraints modify motion commands by examining their parameters, performing calculations, and changing the command's parameters if needed. For example, a grid constraint looks at a translateInLine command and attempts to snap the selection so it aligns with the grid. If a translation by the given distance (parameter) along the line brings the selection within a snappable neighborhood of one of the lines of the grid, the grid constraint changes the distance parameter to a value that produces a perfect snap.

Selections respond to the modified commands: Constraints occur before selections respond to them. Selections respond in the same manner as when the command is not constrained. If the command has been modified by a constraint, the selection moves by the constrained amount, as desired.

Draggers respond to the modified commands: If constraints are not used, draggers can move themselves and send out motion commands to influence the current selection. But when constraints are used, draggers should not move themselves based on the command they generate, since that command is likely to be modified. The result would be a dragger that moves out of sync with the selection. Instead, draggers register with the CommandManager the same way that selections do. After generating motion commands, they wait to be notified by the CommandManager, and move themselves to match the parameters in the motion command. This way, the draggers also base their motion on the constrained command.

Being consistent 9:

Motion commands make it easier to write new tools

We've seen how splitting up manipulators into draggers, motion commands, and selections decouples the constituent parts of a manipulator. The net result of all this is that it's easier to extend the application to do new things.

Creating new draggers: Under this command based architecture, new tools are created by implementing new draggers. The draggers need to be able to generate motion commands, but they do not need to know how selections will respond.

Creating new types of selection: Similarly, a new type of selection can be added to the system by implementing responses to motion commands. The selections do not need to know about the draggers that create those commands.

Providing variety

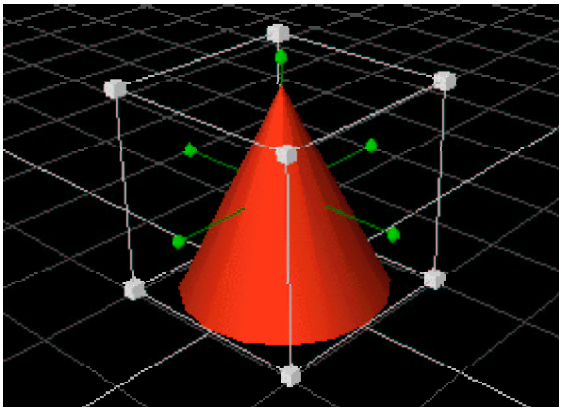
Today's presentation is limited to rotation, translation, and scaling. Even within that limited context, there are different ways to think about the problem. This section describes four tools from CosmoWorlds. Each performs a different type of interactive task. In each description, it is shown how the implementation relates to the techniques that have been presented.

The four tools are:

- **A Box Tool:**
  for operations that relate to the selection's coordinate frame
- **A Wheel Tool:**
  for scaling and rotating about locations and directions that do not relate to the selection's coordinate frame
- **A Plane Tool:**
  for translating parallel to directions that do not relate to the selection's coordinate frame
- **A Dot Tool:**
  for carrying objects to sit upon and align with landmarks in the scene

Providing variety 1:

A Box Tool



a Box Tool

A Box Tool enables operations that align with the selection's coordinate frame

This box−style tool surrounds a selected object. It lets you perform tasks that relate to the coordinate frame of the selection itself. The various handles rotate, translate, or scale the object. The modifier keys allow you to vary the way you perform these tasks as you drag.

Implementation overview: The Box Tool borrows from the infrastructure we've described as follows:

- Cursor mapping techniques, like the ones discussed in the first module of this class, are used to generate motion commands. Projector classes perform these mappings.
- SurroundScale and MinSizeMaxSize nodes scale and place the box around the selection.
- AntiSquish nodes keep the circular feedback round regardless of context.
- MinSizeMaxSize nodes keep the handles to a workable size.
- Shift−dragging and control−dragging provide alternate ways of performing the tasks.
- Gesture−selection aids in selecting directions for 1D scaling, translation, and rotation.

Translation: All translation operations occur relative to the planes of the bounding box. The available operations are:

- Click−drag a side of the box to perform 2D translation freely within a plane parallel to that side of the box.
- Shift−drag to translate along a 1D line within that plane. Gesture−selection is used to pick between the two directions in the plane.

- Control−drag to translate normal to the selected plane.
- The above three operations combine to allow translation in all three directions, regardless of the plane that is used. This means you don't need to rotate the camera to get at a handle for the direction you need.

Rotation: Round green handles let you rotate the object. Rotations occur about the 3 axes of the object's local space. The center of rotation is located at the intersection of the green posts containing the rotation handles. Each post represents a rotation axis.

- Click−drag a round handle to rotate. When you click, two orange arrows appear. These represent choices between two 1D rotations; one about each of two posts you did not click. Gesture−select to initiate 1D rotation about one of these two axes.
- Shift−drag to perform 3D rotation about the center, using an ArcBall.

Scaling: The white box−shaped handles let you scale the object. The center of scaling is the same as the center of rotation. The directions for 1D scaling are parallel to the edges of the box.

- Click−drag to perform uniform scale.
- Shift−drag to perform 1D scale. Gesture−selection lets you pick between all three scale directions.
- Control−drag to perform uniform scale using the opposite corner as the center of scaling. This lets you push or pull the handle toward the opposite corner, leaving the opposite corner fixed.
- Control−Shift−drag (both together) to perform 1D scale about the opposite side. This lets you stretch an object, leaving the opposite side frozen in place.

Changing the center of rotation and scale: The rotation and scale operations occur about the center of the rotation assembly. By default, this is located at the middle of the object's bounding box. The rotation assembly, and hence the center of rotation and scaling, is adjustable; you can translate it (parallel to the sides of the box) by control−dragging the rotation handles.

- Control−drag a round handle to translate the rotation assembly, and hence the center of rotation and scaling. The motion is a 2D translation
- Control−Shift drag a round handle to move the rotation assembly via a 1D translation. Gesture−select to choose between the two available directions.

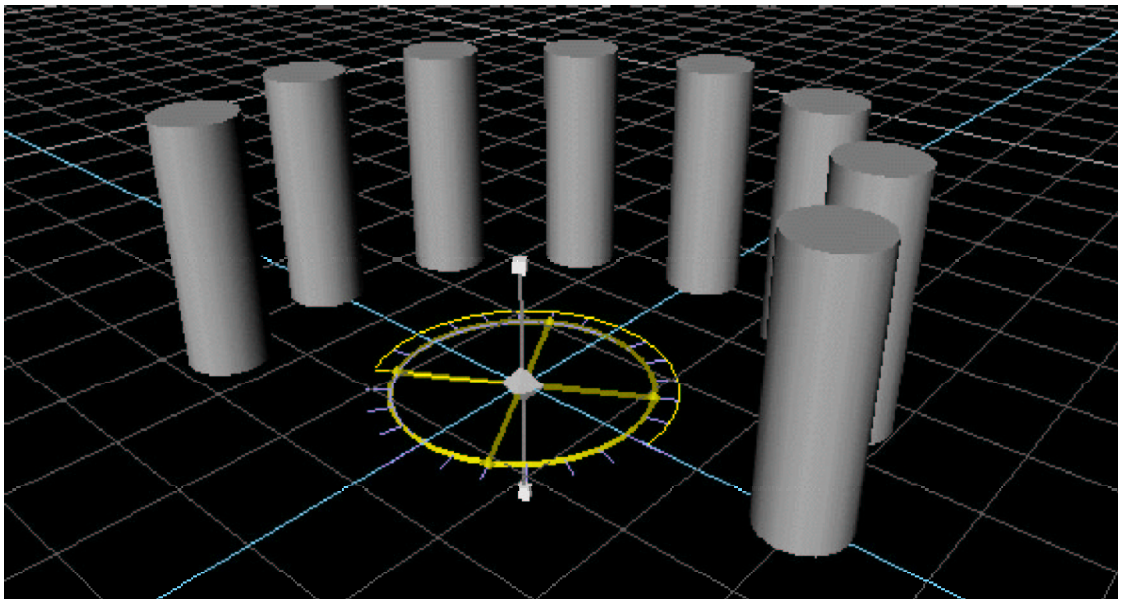
Providing variety 2:

A Wheel Tool

A Wheel Tool enables rotation and scale about landmarks in the scene

This wheel−style tool lets you rotate and scale about arbitrary points and with respect to arbitrary directions. You can position the Wheel, follow−the−cursor style, to align with landmarks in the scene. After this, rotation and scale operations occur relative to the axes of the Wheel.

The Wheel is useful in cases where the object's bounding box doesn't contain the point you want to transform about, or where the axis you want does not align with the object's bounding box.



laying out a ring of columns with the Wheel Tool

Example: Creating a ring of columns
Let's say you are arranging columns in a circle. An easy way to arrange them is to plop a bunch of them down in one spot, and then rotate them one by one about the center of the ring. You can't do this with the Box tool unless you move the Box's rotation assembly (and hence its center of rotation) out of the bounding box, which is kind of unnatural. A column's natural rotation center is really inside the column. The Wheel tool makes such a task simple. Start by placing the Wheel at the center of rotation. Then select and rotate each column in turn, using the Wheel Tool.

Implementation overview: The Wheel Tool borrows from the infrastructure we've described as follows:

- Follow−the−cursor is used to position and align the Wheel.
- Cursor mapping techniques generate the scale and rotation commands. Projector classes perform these mappings.
- AntiSquish nodes keep the circular feedback round regardless of context.
- MinSizeMaxSize nodes keep the handles to a workable size.
- Shift−dragging and control−dragging provide alternate ways of performing the tasks.

Placement: Place the Wheel by click−dragging the white octahedron at the center of the Wheel.

- Click−drag to move and align the Wheel using follow−the−cursor.
- Shift−drag to disable snapping to key features.
- Control−drag to freeze the orientation while moving.

Rotation: Rotation is just like in the Box Tool, except that only one direction is available for 1D rotation. This rotation is in the direction of the visible green ring.

- Click−drag a round handle to perform 1D rotation.
- Shift−drag to perform 3D rotation about the center, using an ArcBall.

Scaling: The box−shaped handles let you scale the object. The center of scaling is the center of the Wheel.

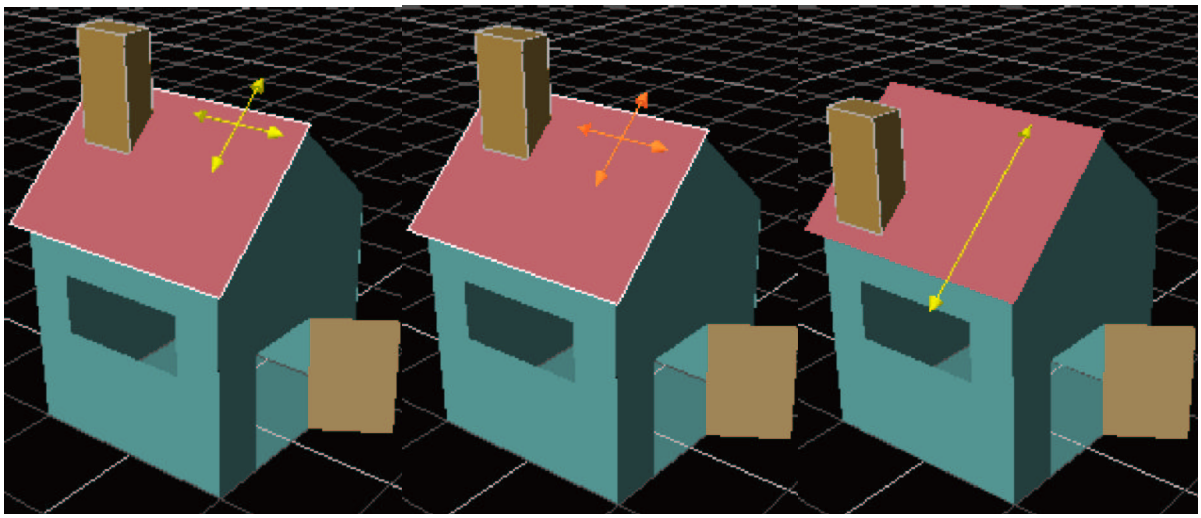
- Click−drag to perform uniform scale.
- Shift−drag to perform 1D scale. The direction will be parallel to the white axis connecting the two scale handles.
- Control−drag to perform 2D scale. This scales evenly in the plane defined by the ring, but does not stretch along the white axis.

Providing variety 3:

A Plane Tool

A Plane Tool enables translation parallel to any edge or plane in the scene

This plane−style tool lets you translate parallel to planes and edges that are not aligned with the bounding box of your selection.



(a) *(b)* *(c)*

Translating a chimney parallel to a roof's slope with the Plane Tool
(a) selecting the plane
(b) selecting the direction
(c) translating parallel to the slope

Example: You might have a chimney on the sloped roof of a house. The slope of the house does not align with any plane of the chimney's bounding box. The Plane Tool lets you relocate the chimney by translating it parallel to the surface of the roof.

Implementation overview: The Plane Tool borrows from the infrastructure we've described as follows:

- Follow−the−cursor is used to select the directions of translation.
- Cursor mapping techniques generate the translation commands once the directions are established. Projector classes perform these mappings.
- A MinSizeMaxSize node keeps the feedback to a workable size.
- Shift−dragging and control−dragging provide alternate ways of performing the tasks.

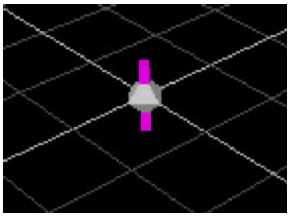
- Gesture−selection aids in selecting directions for 1D translation.

Translation: With this tool, the world is your handle. Any time you move the mouse, follow−the−cursor is used to find a landmark and its directions. The exception is when the mouse is over the handle of another tool, in which case the Plane Tool defers and lets the other tool respond to the mouse. The two directions within the landmark's plane are displayed with yellow arrows. As you move the cursor, the arrows skitter and play over the surfaces in the scene.

- Click−drag to perform 2D translation parallel to the plane you've selected.
- Shift−drag and the yellow arrows turn orange. Gesture−select to perform 1D translation parallel to the selected direction. This technique allows you to move parallel to edges in the scene, as opposed to planes.
- Control−drag to perform a 1D translation in a direction normal to the selected plane.

Providing variety 4:

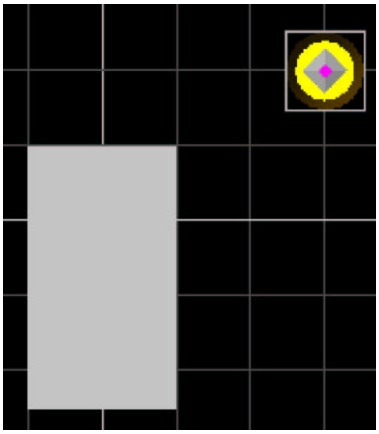
A Dot Tool



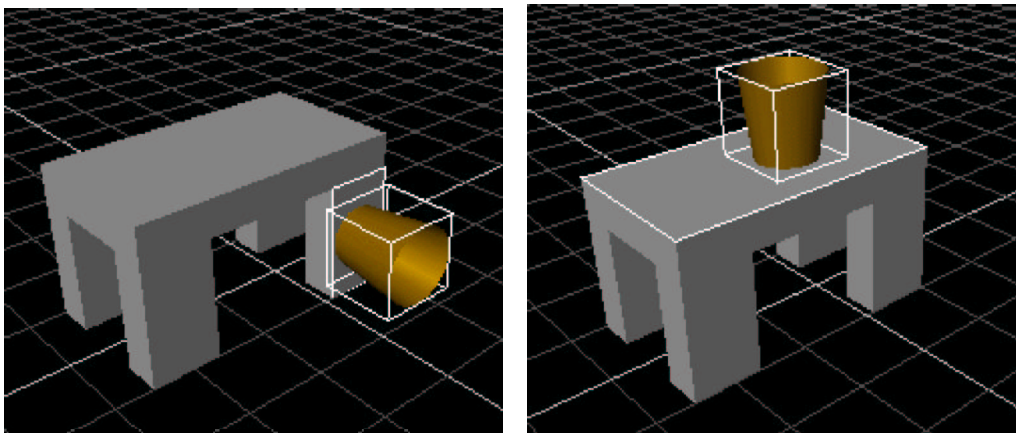
a Dot Tool

A Dot Tool enables you to carry the selection to sit atop a landmark in the scene

This dot−style tool lets you rotate and translate an object simultaneously to sit atop and align with another surface in the scene. Placement of the Dot Tool on the selection marks a point on the selection. When the Dot Tool and the selection are carried together, that marked point serves as the point of connection by which you carry and place the selection.



positioning the Dot Tool on a glass in the top view



moving the glass up the leg and onto the top of a table

Example: Let's say you want to move a glass so it sits atop a table. First, you click−drag the white octahedron to place the Dot at the center of the bottom of the glass. Then, you click−drag the pink post to move the Dot and the glass together onto the surface of the table.

Implementation overview: The Dot Tool borrows from the infrastructure we've described as follows:

- Follow−the−cursor is used to both for placing the dot on the object and carrying the object to a landmark.
- A MinSizeMaxSize node keeps the handles to a workable size.
- Shift−dragging and control−dragging provide alternate ways of performing the tasks.

Placing the Dot Tool: The white octahedron is used to place the Dot Tool. This marks a spot on the selection.

- Click−drag to move and align the Dot using follow−the−cursor.
- Shift−drag to disable snapping to key features.

- Control−drag to freeze the orientation while moving.

Carrying the selection: The pink post is used to carry the selection. Click−drag it and the Dot Tool moves together with the selection. As the pair of objects follows the cursor, the marked location, which is also the center of the Dot, is moved to coincide with the feature below the cursor. The pair of objects also rotates to aligns so that the post is normal to the surface of the landmark.

- Click−drag to move and align both the Dot Tool and the selection using follow−the−cursor.
- Shift−drag to disable snapping to key features.
- Control−drag to freeze the orientation while moving.

Avoiding modes

Peaceful coexistence: The previous module of this class explained why temporal modes are a bad thing. The set of tools presented here do not cause a problem in this regard. This is because each tool occupies a place in space, and the handles may all be selected independently from one and other. Hence, all the tools can peacefully coexist in the scene at the same time and still be accessed. The tool you click determines the tool you want to use.

Toggling the tools on and off: There are cases where one tool may occlude another. And you might not always want to have all the tools visible; they create unnecessary clutter if you are not using them. For these reasons, the tools may be toggled on and off independently. Thus, you can use them in any combination you choose. The net result is much like a real life workbench. You can choose to leave a lot of tools strewn about, or you can put them away.

How the Plane Tool combines with the others: One tool that might seem to cause a problem is the Plane Tool. Since it presents feedback each time you move the cursor, you might think that it would interfere with the other tools, which have literal handles. Fortunately, this is not the case. Event handling occurs (in most systems) in an order as the scene is traversed. By placing the Plane Tool at the end of the scene, you can insure that it is traversed last. Hence, when the cursor is over the handle of a different tool, the cursor motion will elicit a response from that tool. If the cursor is over no other tool, then the Plane Tool will get a chance to pick into the scene and display feedback.

Conclusion

A good underlying architecture supports the implementation of a suite of 3D UI tools in a consistent way. Standard cursor mapping techniques, landmark−finding techniques, and a well−designed set of motion commands provide such a framework.

Within this framework, tools may be constructed to address a variety of tasks. Some have been presented here and related to the implementation framework.

The 3D tools presented have been restricted to a certain set of tasks relating to translation, rotation and scale. Others types of tasks and means of manipulation remain to be designed and implemented. Hopefully, the framework presented here will support the creation of these new 3D tools as well.

An Architecture for Direct Manipulation of 3D Objects

Paul Isaacs
Rikk Carey
Howard Look
David Mott

Silicon Graphics Computer Systems
2011 North Shoreline Blvd.
Mountain View, CA 94039-7311

{ pauli | rikk | howardl | mott }@sgi.com

1 Abstract

This paper presents a comprehensive system architecture for the direct manipulation of 3D objects. Previous work in the area of 3D interaction has concentrated on specific 3D interaction techniques. Our research has been aimed at developing an underlying architecture that accommodates these techniques and facilitates the development of new, reusable 3D interaction objects. The architecture presented in this paper defines an extensible, object-oriented framework for constructing a variety of object classes called *manipulators*. Manipulators are 3D objects that reside within a scene and respond to user input in an intuitive manner. In our architecture manipulators merge behavior directly with geometry. The persistent behavior of these objects permits manipulators to work within applications that have no knowledge of the interaction techniques that the manipulators employ. Our architecture defines a 3D composition model for constructing aggregate manipulators out of simpler ones, paving the way for self-editing objects with embedded controls. Means are provided for the end-user or programmer to easily customize the shape or look of a manipulator according to personal tastes or needs. The architecture has been implemented as part of the IRIS Inventor ™ Toolkit.

2 Introduction

2.1 Motivation

The value of 3D user interface has been clearly demonstrated in recent years[1] [2] [3] [4][5] [8] [9][10] [11][12]. The advantage of direct manipulation is apparent as soon as we see the difference between rotating an object indirectly, using a set of sliders or command line input, versus rotating that same object with a virtual trackball surrounding the object. Immediately, the abstract notion of correlating numbers to orientations vanishes; working with objects within the scene provides a context that is far more intuitive. Furthermore, 3D interaction invites developers to explore much more realistic and natural human interfaces, and encourages creative and expressive solutions that are not possible with traditional 2D interfaces. Interfaces composed of three dimensional objects add new possibilities for realism, directions of motion, shape, and rendering style. Interface objects can look and behave like their real world counterparts, instead of 2D reductions of those objects; this alone presents many new avenues for user interface design. And, of course, traditional 2D widgets can be implemented in 3D. The result is an expanded suite of tools without any loss of old functionality.

Note however that it is still extremely difficult to incorporate 3D interaction techniques into applications. This is because the mathematical algorithms and event distribution paradigms are varied and complex; there is no common mechanism for working with all of them. Furthermore, the creation of new interactive 3D objects should not require the developer to understand the underlying algorithms.

Our primary objectives in developing the architecture were to:

- Provide a framework for creating manipulators.
- Encapsulate geometry and interaction within manipulators so that behavior is defined by the manipulators, not the application.
- Enable manipulator behavior to be persistent between applications.
- Provide an extensible event model that allows 3D objects to respond to user input.
- Encapsulate a set of mathematical techniques that project 2D user input into 3D motion.
- Provide a way to customize the look of manipulators.
- Define a mechanism which allows manipulators to be combined into compound objects that have more complex functionality.

2.2 Related Work

Most of the previously published works on direct manipulation of 3D objects have concentrated on exploring new user interaction techniques [1] [2] [3] [4] [5] [8] [9] [10][11]. These works, while valuable, do not address the problem of building an underlying, extensible architecture for creating 3D interactive interfaces. Our work has concentrated on the development of an easy-to-use framework for 3D interaction which integrates with a typical 3D graphics application. This work is critical in order to encourage experimentation with 3D interaction techniques, and for these techniques to find their way into mainstream applications.

There have been a few published works addressing system issues. Tarlton and Tarlton's framework presented a well developed object-oriented 3D programming model, but separated the behavior of objects from the 3D objects themselves [13]. The Interviews toolkit [7] presents a solid object-oriented architecture for direct manipulation interfaces, however it does not address 3D issues. The *3D widgets* of Conner, et. al., are similar in concept to our manipulators, but take a different approach [3] [11]. Their papers provide compelling arguments for the use of 3D widgets and motivate the desire for a supporting architecture. They provide details about their interaction model, but architectural details of the system are not presented.

2.3 Overview

In this paper we describe an object-oriented framework for developing 3D direct manipulation that is easy to use, efficient and extensible. This system is implemented in C++ and is based on our earlier work, the IRIS Inventor ™ Toolkit presented in [6][12].

The next section introduces *manipulators*. In section 4, we describe how manipulators handle events. In section 5, we present *projectors,* the engines that perform geometric calculations for manipulators. Section 6 describes *parts*, the manifestations of projectors as visible geometry in 3-space, and how they can be customized by users. *Simple and compound manipulators* are introduced in sections 7 and 8. A few interesting examples of higher order manipulators are in section 9.

3 Introduction to Manipulators

Manipulators are 3D objects that reside within a scene and respond to user input in an intuitive manner. Typically, manipulators are employed to edit some aspect of the user's data, such as geometry, layout, or appearance. For example, our system includes a `TrackBall` manipulator that is used to rotate and scale other geometric objects in a scene (see Figure 1[b]).

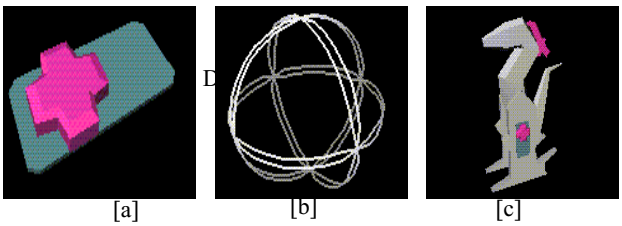


Figure 1. [a] Simple Manipulator, [b] Compound Manipulator and [c] Contraption

Manipulators are composed of one or more *parts*, each of which is a specific geometric entity with a built-in interaction behavior and purpose. Each part may be picked with a mouse-down event. For example, the geometry for each stripe of the `TrackBall` manipulator is a separate part. The geometry can be changed programmatically or through a run-time resource mechanism. Although you can change the geometry of any part, it will always move according to the same paradigm. A real-world analogy is a dresser drawer; drawers come in many different shapes and sizes, but they all slide in and out.

Once a pick initiates manipulation, each subsequent mouse-motion event is mapped into 3D motion in the scene. There are, of course, a variety of ways to map 2D locations into 3D motion. Manipulators employ *projectors* to help in that mapping. In the `TrackBall`, projectors utilize virtual spheres and cylinders to produce rotations.

Each manipulator has *fields*, variables that users can examine to determine its current state. For example, a `Slider` manipulator (see Figure 1[a]) responds to events by moving along the X-axis of its local coordinate space. It contains a field, `translation`, which always reflects its current position along that line. Field values can be used to drive other changes in the scene. So, a `Slider` manipulator's translation can be connected to the position of a second object; the result is a 3D slider that moves another object through 3-space.

Manipulators allow programmers to register *callbacks*, which are executed at key points in its operation, such as when the user starts or finishes manipulation.

Manipulators can be built out of other manipulators. A *simple manipulator* contains only one moving part. By grouping them into *compound manipulators,* a single interactive object with many moving parts can be constructed. Compound manipulators may also contain other compound manipulators. The `Slider` manipulator is a simple manipulator, while the `Trackball` is a compound manipulator.

Contraptions take this process a step further. A contraption is a self-contained object that includes manipulators and other kinds of objects. The manipulators are connected internally to edit parts of the contraption itself. For example, the "walking dinosaur" contraption shown in Figure 1[c] has a `Slider` on its side that makes its legs get longer or shorter. But the `Slider` is also a part of the dinosaur itself, and is not considered an external interface.

4 Event Handling

Inventor's event model is straightforward and is described in [6] and [12]. We give a brief summary: In Inventor, 3D scenes are stored as directed acyclic graphs of objects called nodes, which represent geometry, materials, lights, etc. Actions are operations which can be applied to scene graphs. Event handling is initiated in response to a user or system event, for instance a mouse button press, by passing a message to the root object of the scene graph through the `HandleEventAction`. The toolkit provides event classes so that this mechanism works in any window system. Nodes in the database are traversed in order until some node handles the event, or all nodes have been visited. Any node may handle an event. A node can also grab events; all future events will be sent directly to the node without a database traversal, until the node releases the grab. In Inventor, most nodes ignore events completely, but manipulators, which are composed of nodes, respond to events.

A typical manipulator will handle an event if the event type is of interest and the manipulator has input focus, that is., lies beneath the locator. A manipulator can see if it lies beneath the locator by querying the `HandleEventAction`. The action performs a pick on the scene database using the locator position as the pick point, producing information about the object picked. This information is cached so that subsequent requests do not initiate another pick operation. In Inventor, this information is stored in a `Detail,` which contains a path to the picked object, the intersection point, the surface normal, and other items.

As discussed earlier, manipulators use projectors to map mouse locations into 3D motion. In response to a mouse-button press, a manipulator will set up a projector. In response to mouse-motion events, a manipulator will use that projector to generate 3D motion.

5 Projectors

Projectors are classes that project 2D locations onto an abstract mathematical 3D shape. They interpret successive intersections

with that surface as translations, scales or rotations. Each class of projector performs these projections onto a different kind of shape. The way the intersections with this shape are interpreted also varies by class.

Table 1 shows the currently implemented classes of projectors, the type of manipulation operation they are used for, the shape they project onto, and the way the mouse position is interpreted.

5.1 Projecting the Mouse Position

In order to convert a mouse position into a ray, a projector must be initialized with a view volume. In Inventor, a view volume can be retrieved from any `Camera` node.

By default, a projector's shape is defined in world space. A workspace matrix may be specified that places the projection shape within another 3D space. For manipulators, the *workspace* is the matrix that transforms from the manipulator's local space into world space. The mouse position ray is transformed by this workspace matrix before intersecting it with the projector's shape.

When its `project()` method is called, a projector intersects the transformed ray with the surface using common methods for ray intersection, and returns the result.

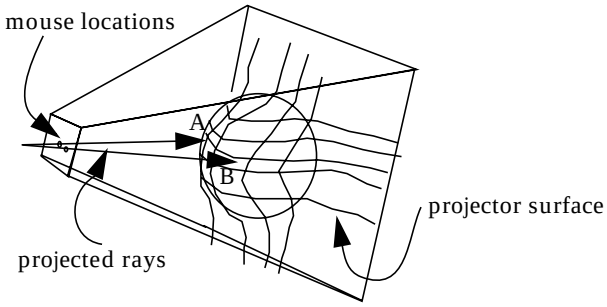


Figure 2. SphereSheetProjector with Two Projected Rays

Figure 2 illustrates a `SphereSheetProjector` with two mouse positions being projected onto it. Refer to Appendix B for sample Inventor code that sets up and performs these projections.

5.2 Interpreting Mouse Motion

Projectors are used by manipulators to determine motion based on successive mouse positions. In the case of a `PlaneProjector`, the code fragment:

```
Vec3f motion
    = myProj->getVector(mousePt1,mousePt2);
```

gives the 3D translation within the plane. The `getVector()` method simply calls the `project()` method twice, returning the difference between the two results.

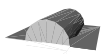
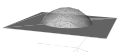
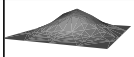
Other projectors may perform more complicated calculations. For example, the `SphereSheetProjector` class performs intersections with a hyperbolic sheet draped over a sphere. This surface is useful for rotations because it is a continuous function; there are no discontinuities when the mouse crosses the sphere's edge (a technique similar to those described in [2] and [10]). The `SphereSheetProjector` has a method `getRotation()`, which takes two mouse position arguments and returns a `Rotation`. `GetRotation()` computes the vector from the two intersections to the center of the virtual sphere and returns a rotation equivalent to the angle between them. For example, to calculate the rotation of an object when mouse moves from A to B (as in Figure 2):

```
Rotation rot = myProj->getRotation(
                        mousePtA, mousePtB);
```

Other rotational projectors project onto different geometry, so they may determine a different rotation based on the same two mouse positions. For example, a `SphereSectionProjector` has discontinuities at the edge of the sphere, unlike the `SphereSheetProjector`. Each has its advantages in different situations.

6 Manipulator Parts

As discussed earlier, a manipulator has a collection of parts. Each moving part has a characteristic interaction behavior. The geometry of parts is customizable by the user or programmer. Active parts can be changed to perform highlighting, providing better feedback

Class	Use	Geometry	Behavior
`LineProjector`	1D translation 1D scale		Projects onto closest point on line.
`PlaneProjector`	2D translation 2D scale 1-axis rotation		Projects onto plane.
`CylinderPlaneProjector`	1-axis rotation		Projects onto infinite cylinder or plane.
`CylinderSectionProjector`	1-axis rotation		Projects onto closest point on section of the cylinder.
`CylinderSheetProjector`	1-axis rotation		Projects onto hyperbolic sheet draped over an infinite cylinder.
`SpherePlaneProjector`	3D rotation		Projects onto sphere or plane.
`SphereSectionProjector`	3D rotation		Projects onto closest point on section of sphere.
`SphereSheetProjector`	3D rotation		Projects onto hyperbolic sheet draped over a sphere.

to the user.

6.1 Behavior of Parts

The behavior of a part is determined by associating it with a projector. In Figure 3, the white shapes are the part geometries. When manipulated, the shapes will move along lines (`Translate1Manips` use `LineProjectors`), within planes (`Translate2Manips` use `PlaneProjectors`), or around an axis (`RotateCylindricalManips` use `CylinderSheetProjectors`). The long arrows are the projector lines and planes; the shaded cylinders are the projector cylinders, centered about the axis of rotation.

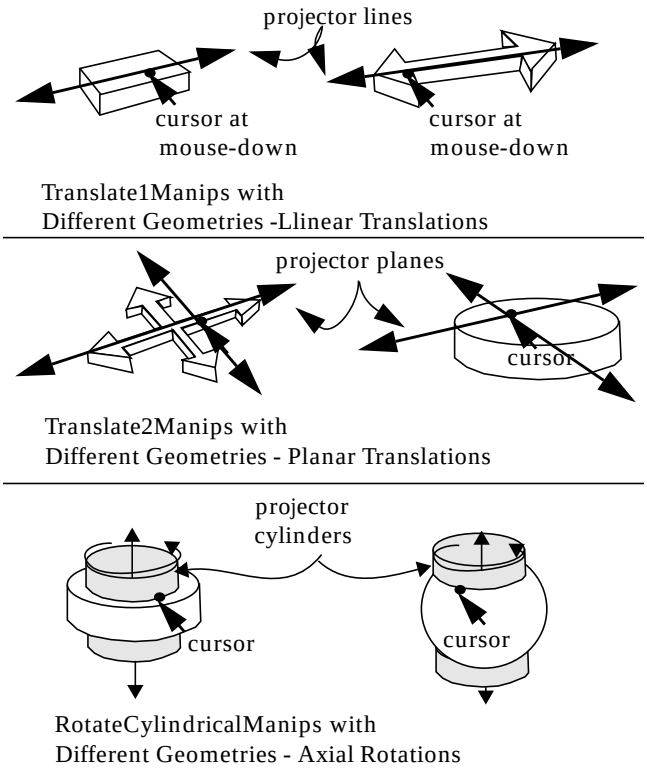


Figure 3. Different Geometries, same Projectors

When the mouse button goes down over a part and manipulation begins, the projector is initialized. The *detail* of the `HandleEventAction` (section 4) contains the *pickPoint*, or initial point of intersection with the geometry. The pickPoint is used to set the projector's shape parameters. For example, the radii of the projection cylinders in Figure 3 are set to be the distance between the pickPoint and the central axis of rotation. The further from the center the pickPoint is, the larger the projection cylinder will be.

Once a projector has been initialized, and for the duration of the manipulation, the shape of the part geometry makes no difference; only the shape of the projector affects the motion. As the mouse is moved, its motion is interpreted by the projector as discussed in section 5.2. For example, the `Translate2Manip` repeatedly calls `getVector()` and uses the results to position itself.

6.2 Customizing Parts

Each manipulator class has a published list of named parts that users can change. For example, to change the "`scalePart`" of the `ScaleUniformManip` into a cone:

```
myScaleManip->setPart(
                "scalePart", new Cone);
```

In Inventor, this modification to the manipulator can be described in a file as follows:

```
#Inventor V2.0 ascii
ScaleUniformManip {
    DEF scalePart Cone {}
}
```

6.3 Highlighting Parts

The simplest kind of part stays the same at all times. But what if you want to highlight (e.g., change the shape or color of) the part while it is in motion? The designer of a manipulator class can build this sort of change into its structure.

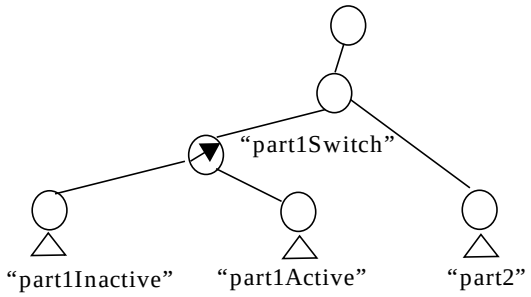


Figure 4. Manipulator Structure With Two-State and One-State Parts (This and subsequent figures use Inventor scene graph diagrams as defined in [6][12])

In Figure 4, "part1" has two states and "part2" has one state. The customizable parts are "part1Inactive," "part1Active," and "part2." The other parts are private, and therefore not customizable.

"Part1Switch" is a switch used to select between "part1Inactive" and "part1Active." When "part1" is not in motion, the switch is set to display the inactive version. When manipulation begins, the manipulator flips the switch to display "part1Active." When manipulation ends, it flips the switch back.

"Part2" will always stay the same. In most of our simple manipulators, there is one part which is pickable and moves, and one part which is unpickable and serves as a visual reference for feedback (such as an axis of rotation or center of scaling). The moveable part has two states for highlighting while the feedback part does not change.

Of course, there are other possibilities, such as three-state parts, parts that animate autonomously when they are picked, and so on.

7 Simple Manipulators

As stated earlier, a simple manipulator is a manipulator with only one moving part. `DragManip` is the base class for simple manipulators that move in 3-space in response to click-drag-release mouse events. The base class has no parts or projectors. Each subclass adds one projector, one moving part, a field reflecting the state of the moving part, and perhaps a feedback part.

The base class `DragManip` provides the "motionMatrix," which contains the cumulative transformation. When the manipulator calculates motion based on the projector, it moves itself by updating the "motionMatrix." `DragManip` also allows programmers to reg-

ister callbacks to be activated after click, drag, and release events.

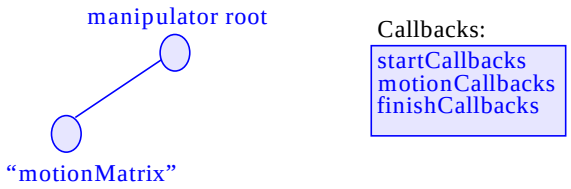


Figure 5. DragManip

7.1 Example: Rotating Disc Manipulator

This manipulator is used to rotate about an axis. It contains two parts. “RotatePart” is the geometry that the user picks on to rotate the manipulator. “FeedbackPart” is unpickable feedback geometry (for instance, an arrow along the Y axis). The manipulator updates the rotation field and “motionMatrix” whenever it moves.

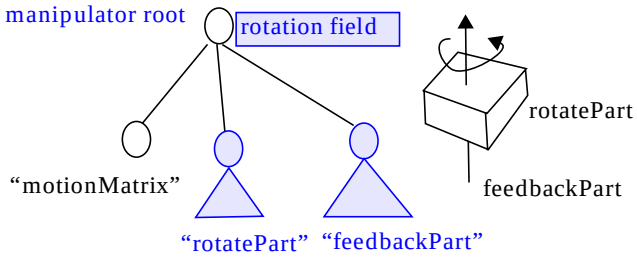


Figure 6. RotateDiscManip

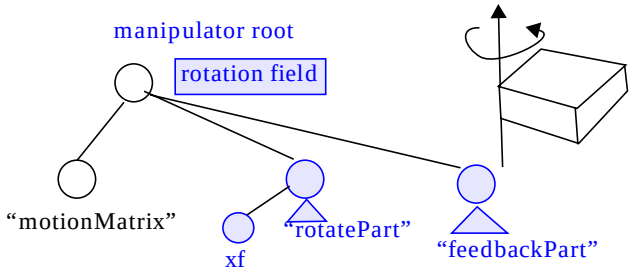


Figure 7. Rotating About a Corner of a Part - Just add a transform to the “rotatePart.”.

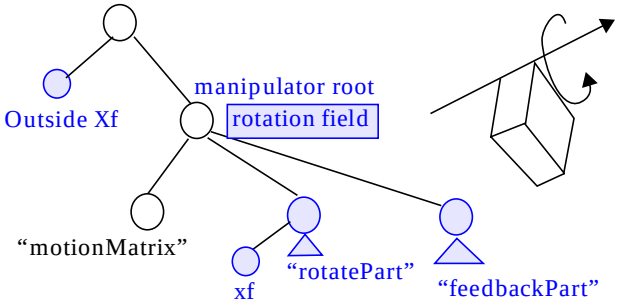


Figure 8. Rotating About an Arbitrary Axis - Use a transform to re-orient the entire manipulator.

Figure 6 shows the ingredients of this manipulator, while Figure 7 and Figure 8 show how it can be used in different situations (In reality, the “rotatePart” is a two-state part as per section 6.3. For brevity, the Figures depict it as a one-state part.)

7.2 Example: PushButton Manipulator

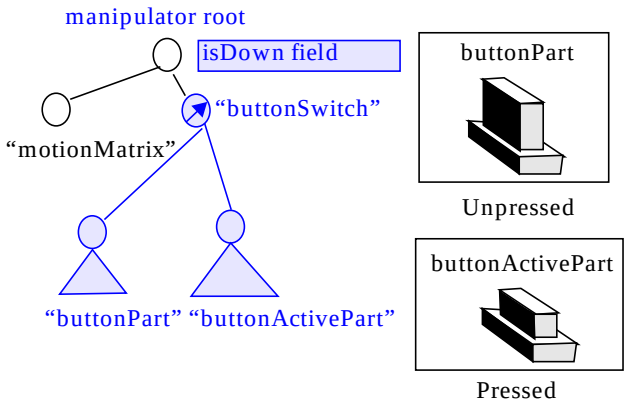


Figure 9. PushButton Manipulator

This simple manipulator shows how traditional 2D widgets can easily be brought into the 3D realm. By default, the *isDown* field of the `PushButtonManip` is FALSE and the “buttonPart” is displayed. When the manipulator is selected, the isDown field is changed to TRUE and the manipulator switches to display the “buttonActivePart.” Nothing happens while the mouse is dragged over the screen. When the button is released, the isDown field returns to FALSE and the “buttonPart” is displayed once again. (The “motionMatrix” is unused.)

8 Compound Manipulators

As stated earlier, compound manipulators group other manipulators together into a single aggregate. `CompoundManip` is the base class for all compound manipulators. Its layout and available callbacks are the same as for `DragManip`.

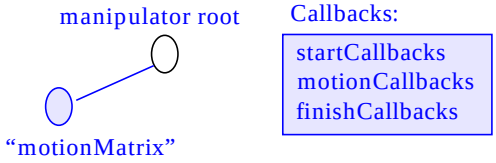


Figure 10. Compound Manipulator

`CompoundManip` includes support to make its child manipulators move in a synchronized way. For instance, when you rotate one stripe of the `Trackball`, all stripes rotate. This is achieved by the following simple mechanism: the `CompoundManip` monitors the fields of its child manipulators. When one of these fields changes, the `CompoundManip` updates its “motionMatrix” based on that of the child manipulator; it then sets the child’s “motionMatrix” to identity. So the motion created by manipulating the child is transferred to a place where it affects the whole group. (Note that not all child manipulators of a compoundManip must be synchronized in this way. Unsynchronized children will simply move relative to the others.)

Unlike `DragManip`, `CompoundManip` does not handle events by default, but passes them on to its children. Subclasses of `Com-`

poundManip may choose to handle events. For example, they may wish to have different child manipulators receive events, depending on which modifier keys are held down. In this case the compound manipulator would route the event to a specific child.

Subclasses typically provide output fields which reflect the state of the manipulator. Often, this requires synthesizing information from several child manipulators into one meaningful quantity or set of quantities, such as a cumulative transform.

8.1 Example: DragBox Manipulator

The DragBox manipulator (see Figure 11) is derived from CompoundManip and coordinates the motion of 6 Translate2Manips to move together as a single box. Transform nodes (xf in the diagram) would place them relative to each other, so that they form a cube.

The translation field of the compoundManip contains the location of the manipulator as a whole.

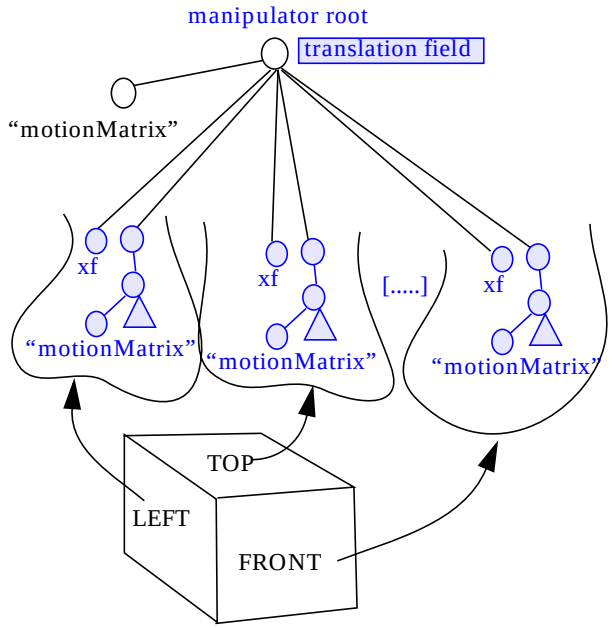


Figure 11. DragBoxManip

9 Further Examples

9.1 Example: HandleBox Manipulator

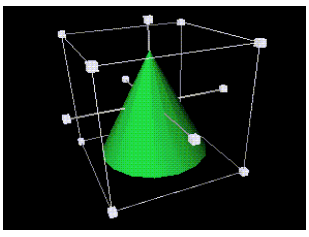


Figure 12. A HandleBoxManip

The HandleBoxManip is a compound manipulator for editing scales and translations. The 8 corners of the box perform uniform 3D scaling, using a ScaleUniformManip. Each of the 3 axial "dumbbells" (lines with a cube at either end) performs 1D scaling along a primary axis with a Scale1Manip.

Each of the 6 faces is a DragPlaneManip, a compound manipulator that translates within a plane when no control keys are held down, but which translates perpendicular to that plane when the ALT key is held down. To do this, the DragPlaneManip employs a Translate2Manip and a Translate1Manip. To route control to the appropriate translation manipulator, the DragPlaneManip handles events as described in section 8.

9.2 Example: A Color Editor Manipulator

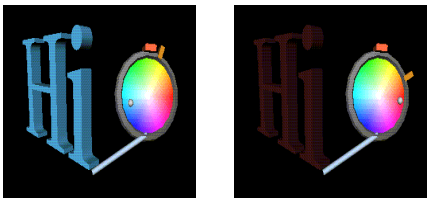


Figure 13. A Color Editor

Figure 13 shows a color editor written as a compound manipulator. It is a composite of three simple manipulators. The planar motion of the small sphere on the front face is interpreted as hue/saturation of the selected color. The small circle is moved with a Translate2Manip. The value of the color is controlled be the lever that runs around the edge, implemented as a RotateDiscManipulator. The button at the top is a PushButtonManip, and when it is depressed the editor aligns itself with the screen. Note that the children of this compound manipulator are unsynchronized.

9.3 Example: A Walking Elephant

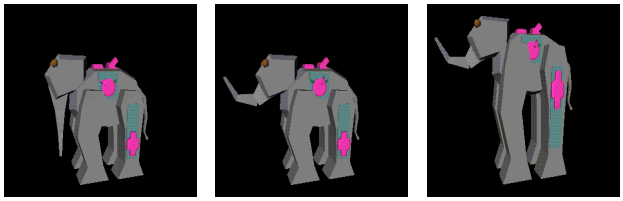


Figure 14. A Walking Elephant

This elephant is an example of what we call a *contraption* (see section 3). It is a persistent object with its own built-in interface, a conglomerate of manipulators and other objects.

The grey parts of the elephant are non-interactive shapes. Along the back of the elephant are two buttons, an on/off switch, created with a ToggleButtonManip, and a reset button, made using a PushButtonManip. The knob on the side of the elephant is a DialManip: it has fields for a minimum, maximum, and current rotation, and it raises and lowers the elephant's trunk. The slider on the leg of the elephant controls the length of its legs and height off the ground. This slider is a SliderManip, and has minimum and maximum fields as well. The elephant can be imported into any application which uses the toolkit and the interaction behavior comes for free.

9.4 Other Assorted Manipulators

The images on the color plates show other types of manipulators and how we have used them.

10 Conclusions and Future Work

We have presented an architecture for designing and implementing 3D interactive objects called manipulators. The architecture accommodates a variety of interaction techniques and has been successful in providing a system for creating a rich set of interaction tools. By allowing simple objects to be combined and modified, we have been able to produce a suite of tools with a wide range of functionality. These objects have encapsulated, persistent behavior whose implementation is hidden from the application.

Directions for future research include creating an interactive program for assembling compound manipulators and completing a 3D suite of tools which replace common 2D widgets. The area of constrained motion also offers great opportunities and challenges. Contraptions have great potential as poseable objects for animation systems, as "working machinery" for use in a virtual modelling workshop, and for creating 3D analogs to the 2D tools now found commonly on the computer desktop.

11 Acknowledgments

The other members of the team that helped design and document the architecture presented in this paper are Ronen Barzel, Gavin Bell, Alain Dumesny, Dave Immel, Paul Strauss and Josie Wernecke. Thanks to Mike Mohageg and Delle Maxwell for assistance with usability testing and esthetics.

Appendix A: Inventor Manipulator Classes

```
Manipulator
    DragManip
        RotateCylindricalManip
        RotateDiscManip
        RotateSphericalManip
        Scale1Manip
        Scale2Manip
        ScaleUniformManip
        Translate1Manip
            SliderManip
        Translate2Manip
        Translate3Manip
        PushButtonManip
        ToggleButtonManip
    CompoundManip
        DialManip
        DragPlaneManip
        TransformEditManip
            HandleBoxManip
            JackManip
            TrackballManip
            TransformBoxManip
        DragCoordinate3Manip
        DirectionalLightManip
        PointLightManip
        SpotLightManip
```

Appendix B: Sample Code

The following is a pseudo-code version of Inventor's handleEvent method and associated routines for the Translate1Manip:

```
void
Translate1Manip::handleEvent(
 HandleEventAction *ha)
{
 // get event from the action
 event = ha->getEvent();

 if (MOUSE_PRESS_EVENT(event, BUTTON1)) {
     detail = ha->getDetail();
     if ( detail != NULL &&
     detail->getPath()->containsNode(this)){
         manipulateStart();
         ha->setGrabber(this);
         ha->setHandled();
     }
 }
 else if ( MOUSE_MOTION_EVENT(event) &&
 ha->getGrabber() == this ) {
     manipulate();
     ha->setHandled();
 }
 else if (MOUSE_RELEASE_EVENT(event, BUTTON1) &&
 ha->getGrabber() == this ) {
     manipulateFinish();
     ha->releaseGrabber();
     ha->setHandled();
 }
}

void
Translate1Manip::manipulateStart()
{
 // Set the two-state part to be active...
 switchPart->whichChild = ACTIVE;

 // Establish the projector line to translate
 // along, an X axis with its origin at the
 // point that was hit.
 line = Line( hitPoint(),
     hitPoint()+Vec3f(1,0,0) );
 proj->setLine( line );

 // Remember the initial hit...
 prevMouse = getMousePosition();

 startCallbacks->invokeCallbacks(this);
}

void
Translate1Manip::manipulate()
{
 // Initialize the projector
 proj->setViewVolume( getViewVolume() );
 proj->setWorkingSpace(
     getLocalToWorldMatrix());

 // Convert 2D mouse locations into 3D motion.
 curMouse = getMousePosition();
 motion = proj->getVector(prevMouse, curMouse );

 // append translation matrix to motionMatrix
 Matrix tm;
 tm.setTranslate( motion );
 motionMatrix->multRight( tm );

 // store mouse location for next time.
 prevMouse = curMouse;

 motionCallbacks->invokeCallbacks(this);
}


void
Translate1Manip::manipulateFinish()
{
 // Set the two-state part to be inactive...
 switchPart->whichChild = INACTIVE;

 finishCallbacks->invokeCallbacks(this);
}
```

12 References

[1] Eric Bier, “Skitters and Jacks: Interactive 3D Positioning Tools”, Proceedings 1986 Workshop on Interactive Graphics, ACM, New York, 1987, 151-169.

[2] Michael Chen, S. Joy Mountford, and Abigail Sellen. “A Study in Interactive 3-D Rotation Using 2-D Control Devices”, Computer Graphics (SIGGRAPH 1988 Proceedings), 22(4) pp.121-129 (August 1988).

[3] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam, “Three-Dimensional Widgets” Computer Graphics Special issue on 1992 Symposium on Interactive 3D Graphics, pp. 183-188 (March 1992).

[4] Michael Gleicher and Andrew Witkin, “Through-the-Lens Camera Control”, Computer Graphics (SIGGRAPH 1992 Proceedings), 26(2) pp.331-340 (July 1992).

[5] Stephanie Houde, “Iterative Design of an Interface for Easy 3-D Direct Manipulation”, Human Factors in Computing Systems(CHI ‘92 Proceedings) pp. 135-142

[6] Iris Inventor Programming Guide, Silicon Graphics Computer Systems, Mountain View, CA, 1992.

[7] Mark Linton, Paul Calder, John A. Interrante, Steven Tang, and John M. Vlissides, “Interviews Reference Manual Version 3.1”, Stanford University, July 1992.

[8] Jock D. Mackinlay, Stuart K. Card, and George Robertson, “Rapid Controlled Movement Through a Virtual 3D Workspace”, Computer Graphics (SIGGRAPH 1990 Proceedings) 24(4) pp. 171-176 (July, 1990).

[9] Gregory Nielson and Dan Olsen Jr., “Direct Manipulation Techniques for 3D Objects Using 2D Locator Devices”, Proceedings 1986 Workshop on Interactive Graphics, ACM, New York, 1987, 175-182.

[10] Shoemake, Ken, from “Math For SIGGRAPH”, SIGGRAPH 1991 course notes #2

[11] Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, D. Brookshire Conner and Andries van Dam, “Using Deformations to Explore 3D Widget Design”, Computer Graphics (SIGGRAPH 1992 Proceedings), 26(2) pp. 351-352 (July 1992).

[12] Paul S. Strauss and Rikk Carey, “An Object-Oriented 3D Graphics Toolkit”, Computer Graphics (SIGGRAPH 1992 Proceedings), 26(2) pp.341-347 (July 1992).

[13] Mark Tarlton and P. Nong Tarlton, “A Framework for Dynamic Visual Applications”, Computer Graphics Special issue on 1992 Symposium on Interactive 3D Graphics, pp. 161-164 (March 1992).

Color Plates

Plate 1. Windmill with on/off switch.

Plate 2. Stopwatch with on/off and reset buttons

Plate 3. Three sliders controlling position of a sphere

Plate 4. Directional light editor - arrow changes direction, ball moves icon

Plate 5. Point light editor - ball moves source of light

Plate 6. Spot light editor - ball moves source, arrow changes direction, cone changes beam spread.

Plate 7. RotateBox: within circles rotates about axis, center of circles drags center of rotation. faces translate, green tabs scals.

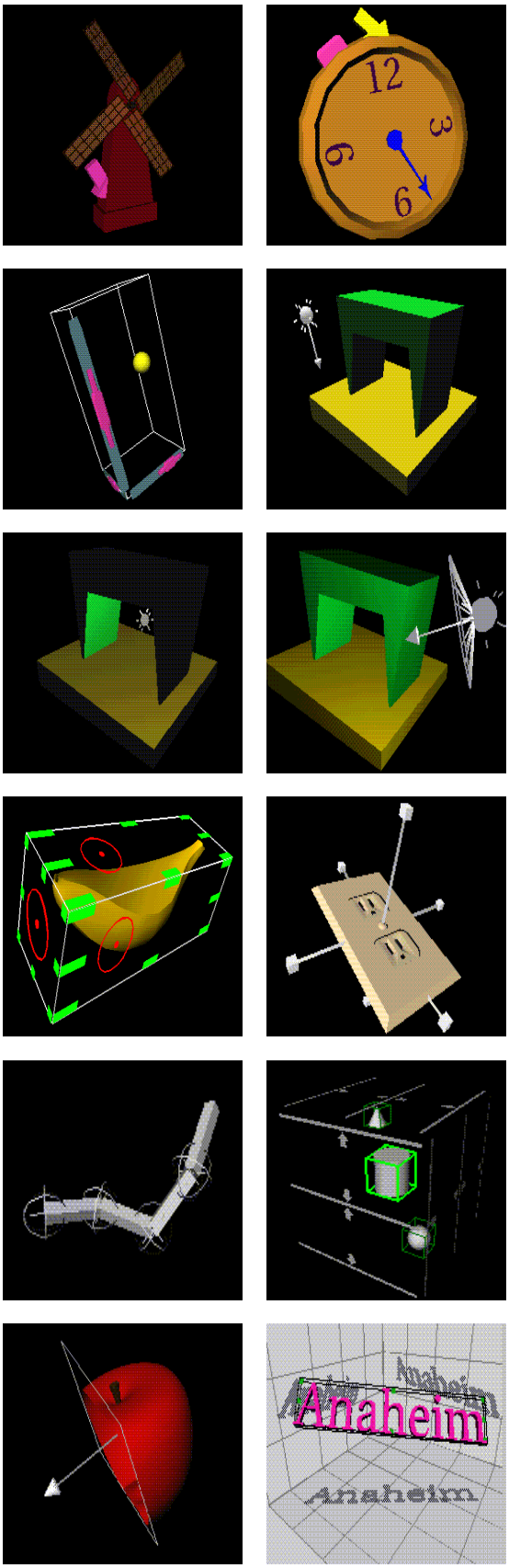
Plate 8. Jack manip: sticks rotate, cubes scale, object translates

Plate 9. Chain: each link can be directly rotated relative to the previous link.

Plate 10. Alignment tool for layout application: picking different arrows aligns the objects inside in different ways.

Plate 11. Clipping plane manip - arrow rotates, square translates

Plate 12. Viewing walls with shadow manipulators



Techniques for Handling Complexity and Robustness in 3D Widgets

Paul Isaacs
Alain Dumesny
Rikk Carey

1 Abstract

As a user "approaches," then "touches" and finally "drags" a virtual tool, he gets closer to the performance of a task. This paper presents techniques to assure that [a] the user's intent is confirmed through increasingly informative feedback, and [b] the user is given early warning and hence may back off before committing to the wrong action.

To clarify the "approach" to a widget, we provide an algorithm for 3D locate highlighting. It is fast, fully rendered, and z-buffered.

For a useful "touch" phase, we discuss design principles for function-revealing feedback. A combination of "take-away" feedback, sub-assemblies, additive feedback and color coding allow widget-makers to imply the purpose and usage of a touched part before it is dragged -- without adding visual weight to the widget.

For coherent feedback during the "dragging" phase, we present new algorithms for mapping 2D gestures into 3-space motion along lines and planes. The algorithms insure that user input has a well-defined effect on the motion for all cursor locations in the view plane, thus avoiding certain degenerate cases that have been previously ignored. We do so by calculating and taking into account the locations of vanishing points and horizon lines.

2 Introduction

2.1 Motivation

In recent years, 3D direct manipulation has come of age. Widgets allow us to grab and move objects to perform spatial tasks that once required us to type numbers or move out-of-the-window sliders.

Research ranges from the presentation of individual widgets [BIER][SNIB][ZHAI] to systems for creating customizable and extensible 3D widget sets [CONN][TARL][WERN1][WERN2]. While some commercial packages only feature indirect manipulation through 1D and 2D sliders [WAVE][SOFT], others have begun to utilize 3D direct manipulation [ALIA] [RADI] [CALI] [SGI1]. Even the line between "model" and "widget" has blurred as researchers incorporate direct manipulation properties into objects such as lights [ISAA1], windows and walls [KENT] and mechanisms [ISAA2][ISAA3].

In this paper we provide solutions in two areas we feel have been insufficiently addressed: complexity and robustness. With multi-functional tools, problems in complexity arise as we try to keep the widget visually lightweight. One option is to make simple widgets and use out-of-scene buttons to change the mode of behavior [ALIA][CALI]. We seek a modeless solution to avoid alternating between button-panel and scene.

Examples exist of well made multi-functional 3D widgets. But try adding rotations to the Open Inventor "HandleBox" [WERN1] or allowing Brown's "Rack" [SNIB] to perform independent twists about each of the three axes. Extra, clearly differentiated parts require more polygons, pixel-space and cognitive mental space on the part of the user.

Eventually we must lower the visual "weight" of the individual parts. Techniques in this paper help make up for this lack of detail. Locate highlighting makes it easier to choose between closely spaced or small objects. Function-revealing feedback adds weight and provides extra information only after a part has received the user's attention.

We also address problems in robustness that affect the way it "feels" to drag a widget. Others have researched the mapping of 2D cursor to 3D object motion [CHEN][GLEI][NIEL][SHOE]. But none of these addresses errors in projecting onto lines and planes that result from failing to consider vanishing points or horizon lines. These errors can cause discontinuous motion when dragging widgets; they are exhibited in all the major software products and systems that employ 3D widgets [ALIA][RADI][CALI][SGI1][WERN1][WERN2][CONN] [LOOK]. We present algorithms to eliminate these errors.

2.2 User Model

We restrict our hardware setup to include mouse-driven widgets rendered as 3D objects on a CRT. Here is our model for how a user gets acquainted with a new multi-part widget (or set of widgets) on the screen:

[1] "The Approach"

The user explores the scene with the cursor. As the cursor passes

over selectable parts of the object, they glow in a way that is consistent across the application. During this exploration, the user becomes acquainted with how the larger objects are broken down into smaller parts, as well as which parts of the object are "hot."

[2] "The Touch"

The user touches a highlighted piece of the widget. If he does not drag it, this is merely an information-gathering operation; it will not cause action to be taken. As each part is selected, new feedback emerges which conveys the function of that part and suggests possible directions for mouse motion. Clear feedback is provided when a choice is to be made between movement in two directions, as well as when a modifier key changes the behavior.

[3] "The Dragging"

The user drags a part. When he does, the widget goes where the user expects. If there is a gesture-based choice between two directions, the user's gesture is interpreted properly. The widget follows any guiding lines or planes without straying or doing anything unexpected. Objects moving along lines can progress toward the vanishing point but may not travel past it. Objects moving in a plane may approach, but not pass beyond, the horizon line for that plane. In either case, no object translates so far away that it becomes too small to select again. When the cursor strays from the guiding feedback for the line or plane, the motion continues to relate to the cursor's position on screen.

2.3 Platform Restrictions and Requirements

Restrictions: We impose hardware restrictions that provide boundary conditions to the user interface problem. We use a single button mouse and a CRT, to comply with the most common hardware configurations.

We use only two keys from the keyboard.We assume that the user associates "Shift" with constraining and "Alt" with a related but alternate behavior. We leave the Control key for use by the application and consider all other keys too "esoteric" for novices.

Requirements: Our locate highlighting algorithm requires z-buffering. There are no additional requirements on hardware or software. We have implemented the work on Silicon Graphics machines using the Open Inventor toolkit, Motif and OpenGL. We present all material in a general manner, with occasional tips directed at developers sharing our software platform.

2.4 The "Transformer"

Much of this research was born from an effort to perfect "the `transformer`," (Figure 1) a do-it-all widget for rotation, translation and scaling of 3D geometries. The `transformer` is so named because it performs affine transformations, but also because it transforms its own shape when different parts are touched. To be truly all-purpose, the `transformer` must perform the following tasks:

- translate - freely within the xz, yz, or xy plane
- translate - independently along the x, y or z axis
- rotate - freely
- rotate - independently about the x, y or z axis
- scale - uniformly
- stretch - independently along the x, y or z axis
- center - be able to change the center of scaling or rotation between various key points on the bounding box of the object.

This is a complex design problem; too many parts are required to permit detailed handles with clear affordances. Our solution is a box-like structure with cubes at the corners and spheres protruding from each face. The `transformer` surrounds the geometry that it affects. A single picture of the `transformer` in its rest state does not reveal how all the parts work. However, interacting with the `transformer` brings out increasingly greater information.

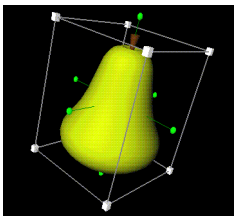


Figure 1. A Transformer in its Rest State, surrounding a Pear

3 The "Approach" Phase

3.1 The Problem

Multi-purpose widgets can have many parts. Each may move independently and perform a different function. Users must be able to find the different parts and feel out their boundaries without committing to an action. We wish to provide feedback as the user explores with the cursor, *before* any part is selected. Clearly, some form of locate highlighting is in order.

This highlighting must be as subtle as possible while still being recognizable, especially during transitions from one part to the next. A single flick of the mouse can carry the cursor across several parts, and should result in minimum "noise." And of course it must be fast. Ideally, rendering the highlight should *not* require a redraw of the entire scene.

These goals (and our solution) are relevant regardless of hardware configuration. They hold with mouse or spaceball, with a 2D bitmapped cursor or a 3D in-the-scene cursor.

3.2 Rejected Solutions

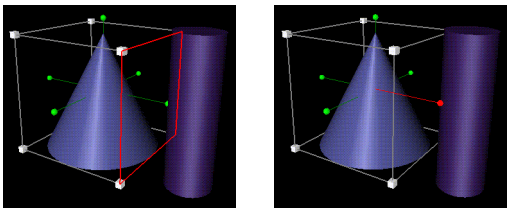


Figure 2. Overlay Plane Highlighting

Overlay Planes:

Speed is crucial in locate highlighting. A natural approach is to use overlay planes, if available, since they do not require a redraw of the scene (See **Figure 2**). We see two disadvantages to this approach. First, the overlay planes pay no attention to depth information. A partially occluded part will render in its entirety in the overlays. This creates visual noise when the cursor moves quickly across the screen; stray lines or surfaces repeatedly appear and disappear. Second, we are restricted to a small number of colors. Feedback in the overlays can not reflect material properties or properties of shading.

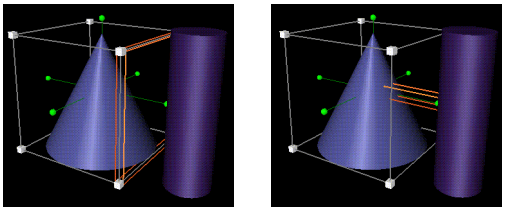


Figure 3. Bounding Box Highlighting

Bounding Boxes:

Another solution is to draw a bounding box around the part (See **Figure 3**). If the box is drawn in the overlay planes, it suffers from the problems outlined above. If drawn within the scene, performance problems arise. When the cursor moves off of an object, how do you "undo" the picture of the old box without redrawing the entire scene? One must keep a list of all objects covered by the box and then redraw them. Also any pixels where the box covered the background must be redrawn. If the box covers many objects it can cause much of the scene to redraw. This solution also ranks poorly in terms of visual noise. Quick cursor motion causes boxes to rapidly pop up and disappear. Finally, this solution can be unclear; when different parts have similar bounding boxes it is hard to tell which is selected.

Text And Sound:

Annotating the selection with text is useful and we recommend it. But it does not take the place of a change to the rendering itself. It is distracting to keep track of out-of-the scene text messages for primary feedback. Sound feedback is useful, but also suffers from the "bombardment of noise" problem when the cursor moves quickly. Both of these techniques are labor intensive as they require unique messages for each selectable object. The labor is increased if language is used and internationalization is required.

3.3 Our Solution -- Z-Buffered Overwriting

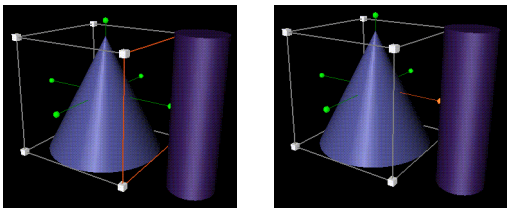


Figure 4. Z-Buffered Overdrawn Highlighting

We propose a z-buffered algorithm for locate highlighting (**Figure 4**). The object in question is overdrawn in a different style. This may be any style the programmer chooses. The only restriction is that the regular and highlighted versions of the object both have the same geometry. Hence when we redraw the object into the current z-buffer we are assured that it will occupy the same exact pixels on screen.

With this restriction in place, the algorithm becomes extremely efficient. Only the highlighted object needs to be redrawn, since no other objects will be overwritten. Until the view changes, the highlighted object may be redrawn into the currently displayed buffer. When the cursor moves off an object we overdraw that object in "regular" style before highlighting any other object. This de-highlighting operation also touches only the relevant pixels of the screen.

In **Figure 4** the locate highlight is rendered as a golden glow -- half emissive color and half diffuse. The feedback is noticeable without being noisy. When the cursor is passed quickly over the scene, each part is emphasized without jarring changes. The transitions are fast because only small amounts are redrawn with each change.

3.4 Algorithm for Z-Buffered Overdrawn Locate Highlighting

The algorithm is the innovation of XXX (Note to Reviewer: Name withheld for blind evaluations).

Objects can render in two ways. Either normally or during an *overdraw pass*. The overdraw pass occurs only when the cursor moves onto or off of a highlightable object. It is a special render pass where only one object is drawn into the front buffer. Objects must know if they are rendering in an overdraw pass because this affects how they set the z-buffer depth comparison function.

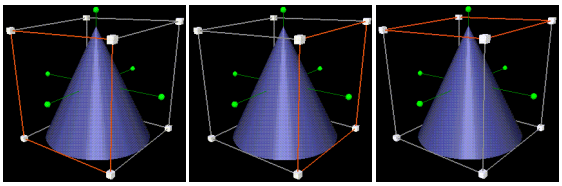


Figure 5. Parts with z-shared Pixels

Sometimes different objects, like the sides of the *transformer*, share pixels with identical z-values(**Figure 5**) . Normal objects use a *lessThan* depth test, so a previously drawn z-shared pixel is not replaced. During the overdraw pass, whether highlighting or de-highlighting, objects use the *lessOrEqual* test and replace every one of their own old pixels. In addition, highlighted objects render using *lessOrEqual* during normal rendering passes. This insures that they will show up in any z-shared pixels.

Each object has two draw routines, regular or highlighted. As stated earlier, both routines must render the same geometry. Each object also has two boolean member variables *highlighted* and *overdrawPass*, stored on a per-object basis.

```
void renderObject( renderState, object ) {
    oldFuncType = getDepthFunction();

     if ( shouldDrawHighlighted(object) or
          object.overdrawPass )
        setDepthFunction( lessOrEqual );

    if ( shouldDrawHighlighted(object) )
        drawHighlighted(renderState,object);
    else
        drawRegular(renderState, object);

    // Restore old z buffer function
    setDepthFunction( oldFuncType );
}

boolean shouldDrawHighlighted( object )
{
    // This gets more complicated with multiple views
    return (object.highlighted);
}
```

drawRegular() and drawHighlighted() are easiest to implement if the software environment provides a way to temporarily override elements of the state. Then drawHighlighted() merely pushes the renderState, sets some property elements in override mode, invokes drawRegular() and finally pops the renderState (Open Inventor programmers have this option). If property override is not available, then drawHighlighted() must change variables used by drawRegular() prior to calling it, or else re-implement drawRegular() with

new properties. The example we saw in **Figure 4** implemented drawHighlighted() by overriding a combination of emissiveColor and diffuseColor material properties.

The handleEvent() routine below relies on fast picking. The system must be able to perform one pick per motion event at interactive speeds. This is not a problem for toolkits like Open Inventor which provide fast picking.

```
void handleEvent( eventState, object )
{
  if ( isMotionEvent( eventState ) ) {
    // Perform pick and see if cursor is over object
    // Toolkit should cache the picked point for each
    // move of the mouse. New picks should be done
    // on a per-event basis, not a per-object basis.
    underCursor = isUnderCursor( eventState, object );

    if ( ! object.highlighted && underCursor ) {
        // Transition into highlighted
        doOverdrawPass(object, eventState, TRUE);
  }
    else if ( object.highlighted && ! underCursor ) {
        // Transition out of highlighted
        doOverdrawPass(object,eventState,FALSE);
    }
  }
  // In an object oriented system, we need to call the
  // base class handle event afterward
  baseClassHandleEvent( eventState, object );
}
```

When the cursor moves onto or off of an object, we invoke doOverdrawPass(). When turning on a highlight, it first checks if "dethroning" a previous highlight. If so, we start by invoking an extra "de-highlight" overdraw pass on the previous highlight object. (We keep track of previous highlights with the global variable prevHighlightObject.)

After that, we make sure we're drawing in the front buffer, set the overdrawPass flag to TRUE and call renderSingleObject() to overdraw just the one object in the appropriate style.

```
void doOverdrawPass( object, eventState, lightOn )
{
    object.highlighted = lightOn;

    // Turn off previous highlight first.
    if ( lightOn && prevHighlightObject != NULL )
        doOverdrawPass( prevHighlightObject,
                        eventState, FALSE );

    // save aloneInfo for single-object drawing.
    if (lightOn)
        aloneInfo = getAloneInfo(eventState, object);

    // This routine is called while handling an event,
    // not while drawing.. With multiple views, we'll
    // need to set the current drawing context before
    // issuing render commands. If single view, this
    // can be commented out.
    setDrawingContext( currentHighlightWidget );

    // render into front buffer
    whichBuffer = getCurrentBuffer();
    if (whichBuffer != FRONT_BUFFER)
        setCurrentBuffer( FRONT_BUFFER );

    overdrawPass = TRUE;
    renderSingleObject( aloneInfo );
    overdrawPass = FALSE;

    // restore the buffering type
    setCurrentBuffer(whichBuffer);

    // get rid of aloneInfo if we just turned off
    if (! lightOn )
        aloneInfo = NULL;
    if ( lightOn )
        prevHighlightObject = object;
    else
        prevHighlightObject = NULL;

}
```

The member variable aloneInfo contains information needed to render the object alone. Extra data in aloneInfo might include transformations and properties that must be set before rendering the object in its local space. (In Open Inventor, aloneInfo would be a path, while object would be a node.)

The renderSingleObject() routine takes aloneInfo as an argument. It collects the extra transforms and properties and sets rendering state in a "set-up" phase prior to calling renderObject(). (In Open Inventor, this is like applying a renderAction to a path as opposed to rendering a single node)

3.5 Special Details For Multiple Views

The algorithm as described above assumes a single view of the scene. For multiple views, we must overdraw the highlight only into the window/widget where the cursor is placed. We monitor the current window by keeping track of when the cursor enters or leaves the various scene-rendering windows. When overlay windows are in use there may be two windows of the same view below the cursor at the same time -- a full-rendering window and an overlay window for drawing on top. Here we must either watch both windows or, if they share a parent window, we can monitor the parent (this is the case with the Open Inventor toolkit's SoXtRenderArea).

When the cursor enters a new window, we set this to be our current highlight window. When the cursor leaves a window and we have drawn a highlight into that window, we first overdraw a regular version of that object into the old window before carrying on. Finally, it is possible that we may be drawing into a window for which we have never received enter notification; as when the cursor is inside that window when the application starts up. In this case we are forced to either use window-system calls to calculate the current cursor window or to simply make an assumption, such as using the first view we drew into.

Once the current highlight window/widget is known, we assume there are two global variables currentHighlightWidget and currentDrawingWidget. Then the doOverdrawPass() must include the call to setCurrentDrawingContext() (described within the pseudo-code) and the code for shouldDrawHighlighted() changes to:

```
boolean shouldDrawHighlighted( object )
{
    if ( object.highlighted &&
(currentHighlightWidget == currentDrawingWidget))
        return TRUE;
    else
        return FALSE;
}
```

4 The "Touch" Phase

4.1 The Problem

Complex 3D widgets do not always have room to render each part with clear affordances. We have a window of opportunity to present feedback during the "touch" phase -- the period of time after the

user has selected a part and before he has moved the cursor. How can we best take advantage of this?

4.2 Function-Revealing Feedback

The following principles comprise an approach for clear, informative feedback. Because the feedback is temporary, we can take somewhat drastic liberties with the widget's form.

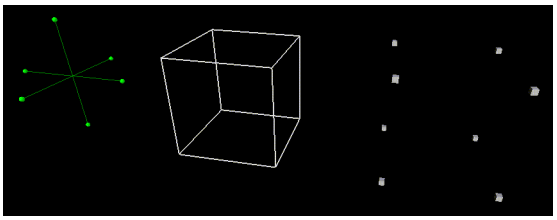


Figure 6. Sub-Assemblies of the Transformer

Sub-Assemblies

Group the widget into two or more sub-assemblies, each with a distinctive set of related operations. We divide the `transformer` into three sub-assemblies for rotating, translating and scale (**Figure 6**).

"Take-Away" Feedback

When a single part is touched, display only the sub-assembly containing that part. Removing the others frees up room to add new feedback without overloading the widget. Continued display of the touched part makes the user feel he retains hold of the object.

Additive Feedback

Add new shapes that clarify the part's function. Guidelines for cursor motion are especially useful. The possiblities here are vast. **Figure 7** illustrates our solutions for the `transformer`.

Color Coding

Consistent color coding provides clear functional clues. Whatever colors you choose, distinguish between the following types of objects. Our color choices, written below, are illustrated in **Figure 7**.

- *Primary* --Object you can touch to perform a task (white or green)
- *Highlight* -- Selected object and/or directions of motion.(yellow)
- *Choice* -- Objects you choose between, as when you press the shift key and may select between two directions of motion (orange)
- *Secondary Hints* --Objects that guide, such as curves or lines you follow with the mouse. Objects that inform, such as bounding boxes for proportional scaling (purple)

The *highlighted* and *choice* colors should stand out. The *secondary hints* color should recede.

4.3 A Note On Buttons

Buttons, when touched and released, effect an immediate reaction. They must be clearly marked as buttons so that users will understand this. Certain shapes, such as a light switch or raised rectangle, can make this clear.

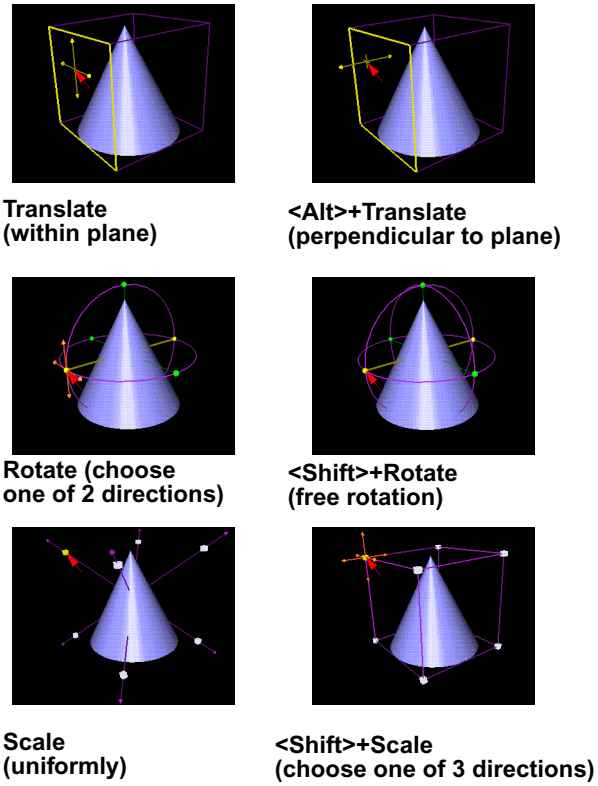


Figure 7. Response of the Transformer to Various "Touches"

5 The "Dragging" Phase

5.1 The Problem

When we touch and then drag an object, we expect the widget to follow the cursor. However all screen locations do not map well to a given line or plane of motion. In each case we must identify these regions and specify coherent behavior when the cursor moves through them.

5.2 Pitfalls In Motion Along Lines

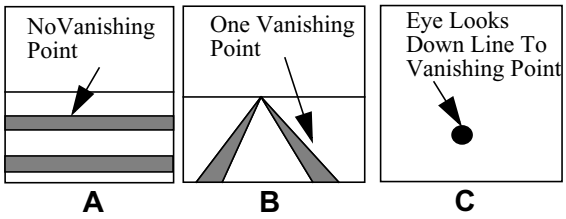


Figure 8. How Infinite Lines Project onto the View Plane

Lines fall into three catagories:

- [A] No vanishing points. The line is parallel to the view plane (**Figure 8A**).
- [B] One vanishing point intersects the view plane (**Figure 8B**).
- [C] Eye looks directly down the line to the vanishing point.. The infinite line appears as a point (**Figure 8C**).

Lines are thin, so most screen locations do not fall directly on a given line. How we find the "closest" point to the cursor influences our

results. The easiest way is to project the cursor location as a line into world space and then calculate the closest point between the two world-space lines. Unfortunately, a disquieting effect occurs. **Figure 9** shows how a single straight line gesture can cause the widget to move in one direction, then switch directions and double back! This behavior occurs because in a perspective view, each pixel points in a different direction from the eye. As we sweep, say, left to right, the points spray a fan of world-space lines. As this fan passes across the location of a vanishing point on screen, the double-back artifact will be witnessed.

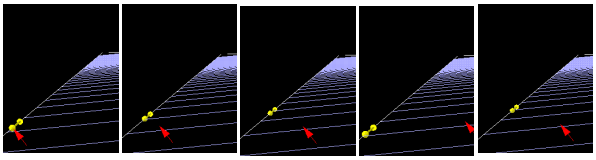


Figure 9. Double-Back Artifact of Finding Line Point in World Space: Widget goes Backward, then Forward as Cursor moves Left to Right.

To fix this, we must calculate the nearest point in screen space. We project the world-space line onto the screen, drop a perpendicular from the cursor to the screen-space line, and then map that screen point back onto the world-space line.

Even when we find the nearest point in screen space, we still get strange motion (discontinuous jumps in depth) in case [B] if we do not first find the vanishing point. As we drop perpendiculars, pixels within the stippled area (**Figure 10**) will be closest to points that do not map back onto the world-space line -- they are on the other side of the vanishing point. Pixels within the first stripe map to locations at infinite or near-infinite distances. We must come up with a "good" width for that first stripe, then map all points in the stippled region and the first stripe to the *clamped vanishing point*. This gives a continuous mapping across the entire screen.

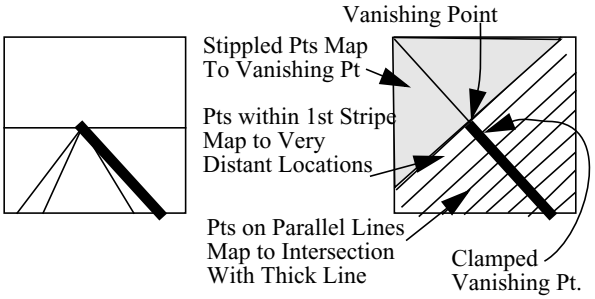


Figure 10. Regions for Mapping Screen Locations onto a Line

5.3 Calculating the Vanishing Point

Given the following transformation matrices:

- *affineMatrix* -- World-space to affine-space. Rotate, translate and scale so eye is at (0,0,0) looking down z axis at world scaled to fit normalized box.
- *projMatrix* -- Affine-space to projection-space, performs perspective distortion. Projection space is a normalized (1x1x1) cube.
- *screenMatrix* -- Projection-space to screen-space. Screen space coords range from (0,0,0) to (pixelWidth, pixelHeight,0). screenMatrix must reflect the aspect ratio of the viewport, via pixelWidth and pixelHeight, or screen-space choices will seem to take place in a distorted space.

This method finds the screen space vanishing point for a line given in world-space, returning the value in *vanish*. If *line* is parallel to the view plane (case [A] of section 5.2), there is no vanishing point and FALSE is returned.

```
boolean getVanishingPoint(line, &vanish)
{
    // Get direction of line in affine space
    affPt1 = line.pt1 * affineMatrix;
    affPt2 = line.pt2 * affineMatrix;
    affineLineDir = affPt2 - affPt1;
    normalize( affineLineDir );

    // Is z-direction 0?
    if ( affineLineDir.Z == 0 )
        return FALSE;

    // Transform affineLineDir to screen
    // space to get vanish (explained below)
    vanish = affineLineDir * projMatrix * screenMatrix;

    return TRUE;
}
```

If the z-component of affineLineDir is 0, then the line is parallel to the view plane and there is no vanishing point.

All parallel world-space lines share the same vanishing point. Consider the one line from this set that passes through the eye point. The eye looks straight down along this line, so all points along it map to the some point in screen-space. This screen point is therefore the vanishing point. In affine-space, the eye is at (0,0,0). The line from (0,0,0) to a point located at affineLineDir must be parallel to the original line, but pass through the eye. All points on the line will map from affine-space into screen-space as the vanishing point. We select *affineLineDir* and multiply by *projMatrix* and *screenMatrix* to find *vanish*.

5.4 Calculating the Clamped Vanishing Point

We can't just pick some number of pixels and move that distance away from the vanishing point. Depending on the line, a fixed number of pixels maps to a wide range of distances. Instead we travel a proportional distance from the vanishing point toward the *appearing point,* defined as the screen space projection of where the line intersects the near clipping plane (**Figure 11**). We call this proportion *VANISH_FACTOR*. In our applications we use a value of 0.01

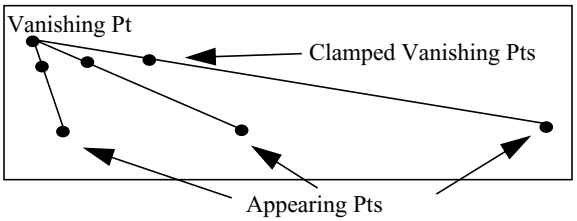


Figure 11. Vanishing and Appearing Points.

This method finds the screen space appearing point for a world-space line and returns the result in *appear*. It intersects the line in affine-space and then converts to screen space. It assumes that *line* is not parallel to the view plane, so it should not be called if getVanishingPoint() has failed on the same line.

```
void getAppearingPoint(line, &appear)
{
    // Transform line into affine space
    affPt1 = line.pt1 * affineMatrix;
    affPt2 = line.pt2 * affineMatrix;
    affLine = makeLine( affPt1, affPt2 );

    // Create the viewPlane in affine space.
    affViewPlane = makePlane( nearViewDist, zAxis );

    // intersect and transform result
    affAppear = intersect( affViewPlane, affLine );
    appear = affAppear * projMatrix * screenMatrix;
}
```

To calculate the clamped vanishing point, interpolate between vanish and appear by an amount of VANISH_FACTOR.

5.5 Projecting a Cursor onto a Line

The projectToLine() routine returns FALSE if the worldLine passes through the eye (case [C] in section 5.2). In this case the whole line maps to a single point and meaningful dragging is impossible.

```
boolean
projectToLine( worldLine, cursor,&result )
{
    // Does line pass through eye? (case [C])
    testPt = closestToPoint( worldLine, worldEyePoint )
    if ( testPt == worldEyePoint)
        return FALSE;

    // Find screen mapping, disregarding vanishing pt.
    screenLine = getScreenSpaceLine( worldLine );
    screenChoice = closestToPoint(screenLine, cursor);

    hasVanish = getVanishingPoint( worldLine, vanish);

    if ( hasVanish ) {
        // One vanishing point (case [B] ).

        // Get clamped vanishing point.
        getAppearingPoint( worldLine, appear );
        clamp = interp( vanish, appear,
                        VANISH_FACTOR );

        // Constrain our choice to lie between appear
        // and clamp.
        constrainBetween(appear, clamp, screenChoice);
    }
    else {
        // No vanish point (case [A] ). Needn't constrain.
    }

    // Project screen choice into world space and
    //intersect with worldLine.
    choiceWorldLine
        = getWorldSpaceLine( screenChoice );
    result = closestToLine(worldLine, choiceWorldLine);

    return TRUE;
}
```

closestToPoint() finds the point on the line nearest the input point. getScreenSpaceLine() creates a copy of the input line and transforms it from world-space to screen-space. constrainBetween() assumes three colinear inputs. It tests whether the third point lies between the first two. If not it moves it to the closer end. getWorldSpaceLine() creates a world-space line corresponding to the ray from eye toward the input screen point. closestToLine() finds the point on the first line nearest to the second line. In this case, they should be close to intersecting since *screenChoice* is a pixel atop the line.

5.6 Pitfalls In Motion Within Planes

Points on one side of the the horizon (**Figure 12**) will intersect nicely with the plane. We need only cast the cursor location into world space and intersect with the world-space plane. Screen locations above the horizon do not map onto the plane. If these locations are cast into world-space lines and intersected with the plane, the resulting point will be located behind the viewers head!

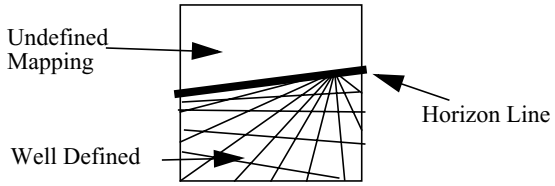


Figure 12. Regions for Mapping Screen Locations on a Plane

To get around this, we map screen points in the "sky" to the closest point on the horizon line. So as the cursor moves through the sky, the widget drags back and forth on the horizon, tracking the cursor. Note that screen points near the horizon map to near-infinite distances. As in the case of lines, we employ the *VANISH_FACTOR* to select a clamping distance. This yields good, continuous results.

5.7 Projecting a Cursor onto a Plane

The method is a straightforward extension of the approach taken with lines. Since all lines within a plane have vanishing points located on the horizon, we construct the horizon line by finding two unique lines in the plane, calculating their vanishing points, and then connecting them. Care must be taken to find two lines that do not lie within the view plane (lines in the view plane won't vanish).

Next we find the *appearing line*. This is the line where the motion plane intersects the screen and is analogous to the appearing point discussed earlier. We find it by connecting the appearing points of the two lines used to create our horizon.

The horizon and appearing lines are always parallel. The closer together they are, the nearer our view is to being edge-on. When they are coincident, the plane is viewed exactly edge-on

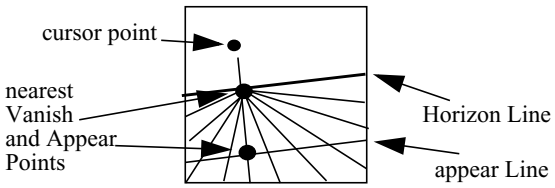


Figure 13. Finding the Nearest Vanishing Point to the Cursor

To get the best placement of our widgets when the cursor is above the horizon, we find the vanishing and appearing points nearest to the cursor by dropping perpendiculars to these two lines (**Figure 13**) . We then constrain the cursor point to lie between them, making sure to account for the VANISH_FACTOR. This assures a new screen point below the horizon, which we may cast into world-space and intersect with the plane.

projectToPlane() returns FALSE if the plane is edge on, since

each screen point maps to an infinite line and meaningful dragging is impossible.

```
boolean
projectToPlane( wldPlane, cursor, &result)
{
 // To start, assume cursor is over plane.
 screenChoice = cursor;

 // Is the plane parallel to the screen?
 hasHorizon = areParallel( wldPlane.normal,
                           wldEyeDir );
 if ( hasHorizon ) {

    // Find two vanishing points.
    // noVanishDir lies in both viewPlane & wldPlane
    noVanishDir = cross( wldEyeDir, wldPlane.normal );

    // These are not in the viewPlane. Both vanish when
    // projected to screen, unless plane is edge-on to eye.
    wldDir1 = cross( noVanishDir, wldPlane.normal );
    wldDir2 = 0.5 * (noVanishDir + wldDir1 );
    normalize( wldDir2 );

    wldLine1 = makeLine2( wldPlane.point, wldDir1);
    wldLine2 = makeLine2( wldPlane.point, wldDir2);
    hasVanish1 =getVanishingPoint(wldLine1,vanish1);
    hasVanish2 =getVanishingPoint(wldLine2,vanish2);

    // Unless both vanish, plane is edge-on
    if ( not( hasVanish1 ) or not ( hasVanish2 ) )
        return FALSE;

    horizon = makeLine( vanish1, vanish2 );

    // Connect appearing points for appearLine
    getAppearingPoint( wldLine1, appear1 );
    getAppearingPoint( wldLine2, appear2 );
    appearLine = makeLine( appear1, appear2 );

    // Find closest point on each line to screenChoice
    vanish = closestToPoint( horizon, screenChoice );
    appear = closestToPoint( appearLine,screenChoice);

    // Plane is edge-on if these overlap
    if (vanish == appear)
        return FALSE;

    // Get clamped point for our choice's line.
    clamp = interp(vanish, appear, VANISH_FACTOR);

    // Constrain choice to lie between appear and clamp
    constrainBetween( appear, clamp, screenChoice );
 }
 // Project choice to wld space, intersect with wldPlane
 choiceWldLine = getWorldSpaceLine( screenChoice );
 result = closestToLine( wldPlane, choiceWldLine );

 return TRUE;
}
```

The routine makeLine2() creates a line from a point and a direction, whereas the earlier makeLine() creates a line from two points.

6 Conclusion

We have presented methods that make it easier for users to understand the usage of complex multi-function 3D widgets. We have also given algorithms for making widgets move in a manner more consistent with the user's intent. Many challenging problems remain unsolved in this field. Of particular interest to these authors are: How can we best use 3D widgets for scene assembly? How do we allow users to constrain motion of 3D widgets to useful increments, and how can this notion be built into a clear user interface? How can we provide a way for users to gracefully change their frame of motion between different coordinate systems? Each of these areas is ripe with challenges for both the visual designer and the applied scientist.

7 References

[ALIA] Alias Research, "Alias Sketch",Modeling Software. Toronto, Canada. 1994

[BIER] Eric Bier, "Skitters and Jacks: Interactive 3D Positioning Tools", Proceedings 1986 Workshop on Interactive Graphics, ACM, New York, 1987, 151-169.

[CALI] Caligari, Inc. "TrueSpace," Modeling Software. Mountain View, CA. 1994

[CHEN] Michael Chen, S. Joy Mountford, and Abigail Sellen. "A Study in Interactive 3-D Rotation Using 2-D Control Devices", Computer Graphics (SIGGRAPH 1988 Proceedings), 22(4) pp.121-129 (August 1988).

[CONN] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam, "Three-Dimensional Widgets" Computer Graphics Special issue on 1992 Symposium on Interactive 3D Graphics, pp. 183-188 (March 1992).

[GLEI] Michael Gleicher and Andrew Witkin, "Through-the-Lens Camera Control", Computer Graphics (SIGGRAPH 1992 Proceedings), 26(2) pp.331-340 (Jul '92)

[ISAA1] Paul Isaacs, "Fun With Draggers: Building An Interactive Track Light", Silicon Graphics Developer News, August 1994

[ISAA2] Paul Isaacs, "Linkatron", SGI Demo Program,1994

[ISAA3] Paul Isaacs, "Creating a 3D User Interface: the Thief, the Industrial Designer, the Craftsman and the Nerd", Position Statement for Workshop on Challenges of 3D Interaction, CHI'94

[KENT] Jim Kent, "BluePrint", SGI Demo Program, 1995

[LOOK]Howard Look, "Curve", SGI Demo Program, 1990

[NIEL] Gregory Nielson and Dan Olsen Jr., "Direct Manipulation Techniques for 3D Objects Using 2D Locator Devices", Proceedings 1986 Workshop on Interactive Graphics, ACM, New York, 1987, 175-182.

[RADI] Radiance Software International, "EZ3D", Modeling Software. Berkeley, CA. 1994

[SGI1] Silicon Graphics "Showcase 3.0," Drawing Software, Mountain View, CA. 1993

[SHOE]Shoemake, Ken, from "Math For SIGGRAPH", SIGGRAPH 1991 course notes #2

[SNIB] Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, D. Brookshire Conner and Andries van Dam, "Using Deformations to Explore 3D Widget Design", Computer Graphics (SIGGRAPH 1992 Proceedings), 26(2) pp. 351-352 (July 1992).

[SOFT] Softimage Software "Creative Environment" Software, Montreal, Canada. 1994.

[TARL] Mark Tarlton and P. Nong Tarlton, "A Framework for Dynamic Visual Applications", Computer Graphics Special issue on 1992 Symposium on Interactive 3D Graphics, pp. 161-164 (March 1992).

[WAVE] Wavefront Technologies "PowerAnimator", Santa Barbara, CA. '94.

[WERN1] Josie Wernecke, *The Inventor Mentor*. 1994, Reading, Ma: Addison-Wesley.

[WERN2] Josie Wernecke, *The Inventor Toolmaker*. 1994, Reading, Ma: Addison-Wesley.

[ZHAI] Shumin Zhai et al, "The Silk Cursor", CHI '94 Conference Proceedings, pp. 459-464(April 1994)

A Manipulator for 3D Transformations

Paul S. Strauss
Pixar Animation Studios
pss@pixar.com

Paul Isaacs
Shout Interactive
pauli@shoutinteractive.com

Abstract

This paper describes a direct manipulation interface (manipulator) for transforming objects in a three-dimensional scene. The manipulator allows a user to transform, scale, or rotate an object using a single, integrated tool that was improved over several years of user tests and customer feedback. The design of the interface and some implementation details are described.

Introduction

Direct manipulation refers to the use of an input device such as a mouse or pen to transform an object in a three-dimensional scene so that device motion corresponds more or less directly to the desired transformation. There are two basic approaches to direct manipulation interfaces. One is to use the object itself as a handle: a user can click and drag on the object to move it, and modifier or "hot" keys can be used to access various transformation modes such as translation, rotation, or scaling. The other approach adds geometric constructs, which we will call *manipulators*, around or near the object to act as an interaction tool.

The latter approach has become widespread in recent years, due primarily to the less intimidating interface presented to users. Intuitive affordances can guide users to the correct modes and motions, rather than relying on hidden key sequences. Manipulators can also provide feedback about current size and orientation, as well as actions that will be taken upon device motion.

We have designed and implemented a manipulator for transforming objects in three dimensions. The goals of its design are:

- Allow the user to scale, rotate, or translate an object with the same manipulator.
- Minimize the need to change views to perform any desired motion.
- Provide feedback about what action will be taken before the user initiates it.
- Use modifier keys when necessary to alter the interaction behavior, but reserve unmodified motion for the most common and useful actions.
- Keep the look and feel as consistent as possible within the constraints of the other goals.

A Brief History

The original work on manipulator design was part of the first incarnation of the Inventor toolkit from Silicon Graphics [3]. Inventor offered a variety of simple manipulators that performed single tasks, such as scaling, rotation, or translation, and also several compound manipulators that allowed several tasks to be accessed simultaneously. A similar approach was also taken by researchers at Brown University [1,2].

Customer feedback and usability studies over a period of years have led to the improvements found in the manipulator presented here. It has been used successfully in the Cosmo Worlds product for both SGI and Windows platforms.

Color and Feedback

The manipulator is shown being used to transform a cone in Figure 1a. The gray lines forming the manipulator box form rectangular faces that are used for translation, the white cubes at the corners are used for scaling, and the green

knobs on the protruding sticks are used for rotation. Using locate highlighting, each of these interactive parts turns orange when the cursor is over it, signaling that it will be active if the device button is pressed, as shown in Figure 1b.

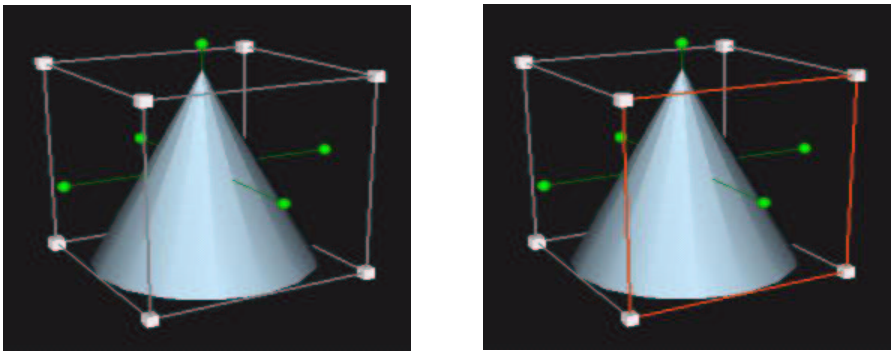


Figure 1. (a) The 3D manipulator in its resting state; (b) Locate highlight indicating which part will be active when the mouse button is pressed.

Once the button is pressed to initiate an action, irrelevant parts of the manipulator are hidden and feedback geometry is added. The feedback is updated when the state of the interaction changes. For example, Figure 2a shows the state of the manipulator when the mouse button is pressed with the cursor over a rotation knob. The translation and scaling affordances disappear, and the selected knob turns yellow to indicate it is active. Purple rings show the two possible rotation circles prior to motion, and orange arrows indicate a choice of two directions to move the mouse to begin rotating. Once a direction has been chosen, as in Figure 2b, the unused ring and arrow disappear, and the relevant arrow turns yellow. When the button is released, the manipulator reassumes its normal appearance.

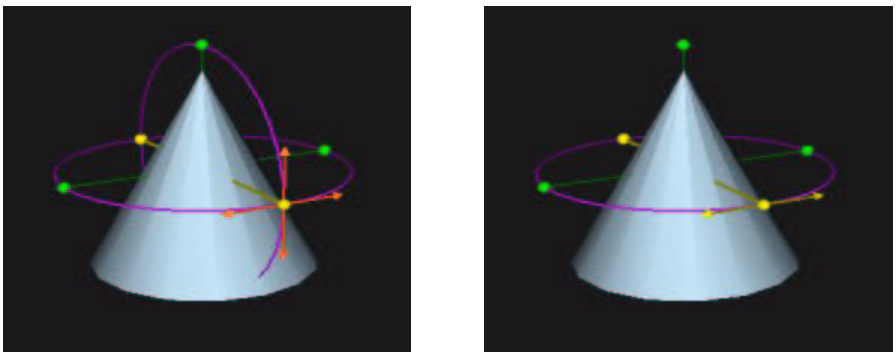


Figure 2. (a) Feedback for initiation of rotation gesture; (b) Feedback once the direction of rotation has been determined from mouse motion..

The color scheme described above is used consistently throughout the rest of the interface. Orange indicates a choice between options, and yellow indicates selection of one of those options. Purple is used for auxiliary feedback.

Interaction

The manipulator was designed to make default interaction perform the most common operations:

- *Translation*: Translate in the plane of the selected face.
- *Rotation*: Rotate around one of the two primary axes perpendicular to the selected knob, based on the first user motion.
- *Scaling*: Scale uniformly about the center.

The Shift key is used to add or remove constraints to the above operations:

- *Translation*: Translate in one of the two principal directions in the selected face, based on the first user motion.
- *Rotation*: Rotate freely to keep the selected knob near the cursor.

- *Scaling*: Scale (nonuniformly) about the center in one of the three principal directions, based on the first user motion.

The Control key is used to access an alternative transformation:

- *Translation*: Translate perpendicular to the selected face. (Note that this allows the user to translate in all three directions with one face.
- *Rotation*: Translate the center of rotation in the plane perpendicular to the selected knob. (The Shift key constrains this to a single direction in that plane.)
- *Scaling*: Scale the object holding the corner opposite the selected knob fixed. With the Shift key pressed, keep the opposite face fixed.

The user may press and release modifier keys during actions to switch modes without releasing the mouse button.

Implementation Details

Most of the implementation of the manipulator is straightforward, but there are a few tricky areas involving transformation of the manipulator geometry itself.

The size of the manipulator should relate to that of the object it is transforming. Given an aligned bounding box for the object, it is simple to scale the manipulator geometry appropriately. However, it would be bad to scale all the individual pieces of the manipulator, since this could distort the shapes of the knobs and the angles between them. If the manipulator is represented as a scene graph, this requires the insertion of nodes in the correct place to undo the nonuniform scaling. We find that using the average size of the three dimensions as the uniform scale factor works well.

Another problem occurs when the relative size of the manipulator on the screen is too small or too large. The knobs may be too tiny to click on in the former case and may take up too much space in the latter case, obscuring other objects. To avoid these problems, we insert a scene graph node above each knob that scales the knob's size to stay within a fixed screen-space range. The scaling can be applied continuously or only when the mouse button is released.

Care must also be taken when multiple views of a scene are visible simultaneously. When a manipulator appears in more than one view, the locate highlighting feedback should be visible in only the view the cursor is over.

Acknowledgements

The work described in this paper could never have been done without the contributions of Howard Look, Mike Mohageg, Rob Myers, Deb Galdes, David Mott, and the rest of the Inventor Team at Silicon Graphics. The authors extend their thanks to all of them.

References

[1] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam, "Three-Dimensional Widgets," In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, pp. 183-188 (March/April, 1992).

[2] Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, D. Brookshire Conner, and Andries van Dam, "Using Deformations to Explore 3D Widget Design," *Computer Graphics (SIGGRAPH '92 Proceedings)* pp. 351-352 (July, 1992).

[3] Paul S. Strauss and Rikk Carey, "An Object-Oriented 3D Graphics Toolkit," *Computer Graphics (SIGGRAPH '92 Proceedings)* pp. 341-349 (July, 1992).

The Design and Implementation of Direct Manipulation in 3D

Introduction

Paul S. Strauss
Pixar Animation Studios

What Is Direct Manipulation?

- WYSIWYG interaction with visual data
  - [Schneiderman ’82]
- "Direct": interaction is in same visual context as data
- "Manipulation": mapping from input valuator to changes to data
- Typically uses a *manipulator*

Why Is It A Good Thing?

- Result directly coupled to input motion
- User's focus stays in work area
- Can guide/constrain interaction in natural ways

Haven’t I Seen This Before?

- Common in 2D
  - Handle boxes in drawing programs for scale, stretch, rotate, and drag
  - Drag−and−drop
- Becoming more common in 3D
  - View navigation
  - Manipulators for transforming/editing objects

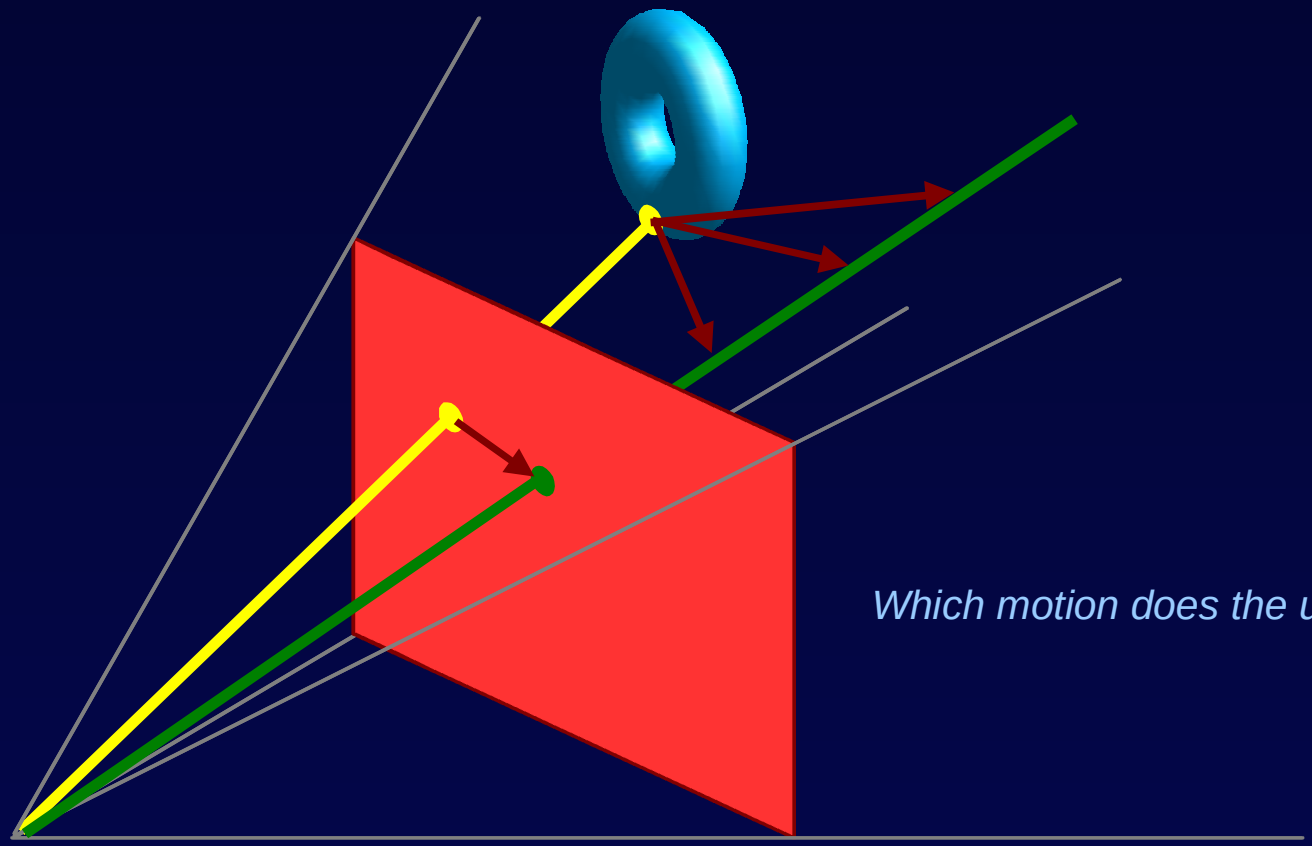
Isn’t It Pretty Easy?

- Easy in 2D
  - One−to−one mapping from cursor position to spatial domain
  - Easy intersection testing
  - Straightforward interpretation of motion
  - No perspective distortion

Why Is It Hard in 3D?

- 2D view of virtual 3D world
- Ambiguity – infinite mappings from cursor position to spatial domain (line)
- Even worse when motion is considered

Ambiguous Input Mapping in 3D



Which motion does the user want?

Is That All?

- Perspective problems
  - How large handles appear relative to data
  - How large handles appear relative to each other
- Occlusion problems
  - Handles obscured by data
  - Entire manipulator obscured
- Precision problems
  - How to deal with exact placement
  - Mathematical instabilities

A Little Ancient History

- 3D dragging
  - Softimage [’88]; Snap−dragging [Bier ’90]
- Direct rotation
  - Virtual sphere [Chen et al. ’88];
    Arcball [Shoemake ’92]
- Direct general 3D transformation
  - Inventor [Strauss/Carey ’92]
- Other 3D manipulation
  - 3D Widgets [Conner et al. ’92, Snibbe et al. ’92]

Related Areas

- Immersive environments (VR)
- Two−handed input
- Haptics