*The material object of observation, the bicycle or the rotisserie, can't be right or wrong. Molecules are molecules. They don't have any ethical codes to follow except those people give them. The test of the machine is the satisfaction it gives you.*

— Robert Pirsig
*Zen and the Art of Motorcycle Maintenance,* p 146

# Chapter 8

# Interaction Techniques

The differential approach allows a range of interaction techniques to be created by defining connectors which compute interesting attributes of objects, attaching controllers to these connectors at the proper times to cause the object which they depend on to move, and by using combinations of these controllers to create more complicated behaviors. This chapter discusses these topics in that order.

Throughout this chapter, the differential approach will be applied to specific interaction problems. In many cases, we will simply recreate previous interaction techniques on top of the new abstractions, often in a way that allows them to be extended, generalized, or at least implemented with fewer of the typical hassles. Several of the interaction techniques, however, would be difficult or impossible to create without the approach.

## 8.1 Attributes to Control

An important feature of the differential approach is the flexibility it provides in the types of attributes that can be controlled. In this section, we provide examples of attributes which potentially make interesting connectors to control. All of these examples are available in Bramble's standard library.

One consideration is that some attributes work better than others, either from a mathematical or user interface perspective. While developing methods for determining the effectiveness of an attribute as a control is left for future work in Section 10.3, here are some intuitions developed from experience:

1. Simpler functions work better.

2. One constraint should remove one degree of freedom. It it much better to use multiple equations than to combine multiple constraints into a single equation.

For example, it is better to have $x = 0$ and $y = 0$ than $x^2 + y^2 = 0$.

3. The derivatives should not vanish when the constraint is met. This often relates to 2. In the example, the derivatives of $x^2 + y^2$ are 0 when $x = 0$ and $y = 0$.

4. Normalizations throw away information, and therefore should be avoided.

5. Controls with physical analogies are easier to explain to the user and to understand the mathematical behavior.

6. Controls that are positional (e.g. compute a position in space) are easier to connect to input devices and to understand.

Each of these issues will be discussed in the specific examples to which they apply.

### 8.1.1 2D Object Controls

The most basic attributes are the positions of points on 2D objects. These can be directly controlled by attaching their values to the mouse's position. Since the system must be able to compute points in order to draw the object, the functions required for manipulation are known. Often, points are placed at interesting points such as the center of an object, in addition to places distributed around the object.

The `define-shape` function in Bramble, detailed in Section A.2.1, provides a mechanism for defining 2D objects. It demonstrates how the same information used to draw an object is used to control it.

For parametric curves, the simplicity of the differential approach is especially apparent. A parametric curve is defined by a function

$$\{x, y\} = \mathbf{f}(u, \mathbf{q}). \tag{8.1}$$

This function computes the position of a point of the curve, given a value for the free parameter and the parameters defining the configuration. The ability to compute this function is necessary to draw the curve. Just as we draw the curve by specifying $u$ values for specific points, we can create connectors to manipulate the curve by specifying $u$ values. Using the parametric function for a connector requires the derivatives of the function that can be obtained easily and automatically. Even if the function is provided only as a black box that can be evaluated, the derivatives can be estimated numerically using finite differences.

The use of connectors from the parametric function provides an automatic and uniform methods for providing interfaces. All that is required to define an object type is the parametric function. Connectors can be placed along the length of the object, permitting the user to grab the object at various points. Each type of object is manipulated in exactly the same way: the user can grab any point on it and pull it. All parameters of the

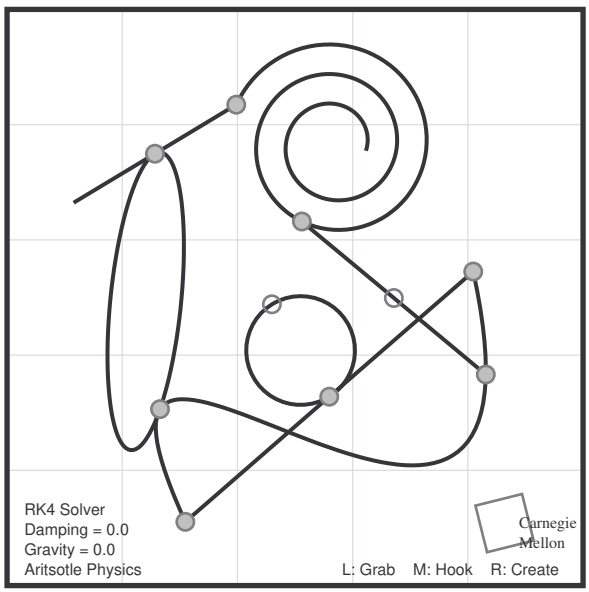


**Figure 8.1:** A simple curve editor showing a variety of parametric curve types. New types can be defined simply by providing the parametric function used to draw the objects. All objects are manipulated uniformly: pulling a point on the objects causes all of the object's parameters to be adjusted accordingly.

object are affected, not just their translations. The mass matrix metric of Section 3.4.1 provides a uniform metric for curves as well, as first demonstrated in the FF system by Witkin [Wit89b]. A later implementation that I wrote is shown in Figure 8.1, and was presented in [GW91a]. These systems both permitted the definition of new types of objects at compile time by specifying a parametric function to a symbolic mathematics package that automatically generated the code for the object. A later version, described in Section 9.4, permits new functions to be defined dynamically.

For a specific object, the uniform interface may not be better than a hand-crafted one. However, it may not always be practical or possible to devise good interfaces for all object types. For a parametric curve in the differential approach, interfaces do not need to be defined for each object type. Given the parametric function used to draw the object, the code to compute the connectors can be automatically generated. In fact, to add a new curve type to a drawing editor, all that needs to be provided by the programmer is the parametric function and a little auxiliary information such as how finely to sample the curve and initial values for the parameters. Everything else can be generated automatically. A prototype system, described in Section 9.3, even adds an icon creating instances of the object to the program's interface.

A connector for a particular $u$ value provides a point on the curve that can control

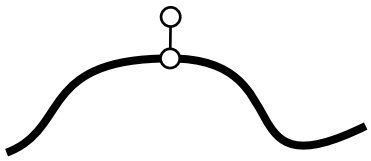

**Figure 8.2:** A crowbar point is computed by an offset of the normal to a point on the curve. The crowbar provides a handle to control the normal of the point that is positional, and therefore can be directly coupled to the pointing device.

the parameters. If the connector stores $u$ as a modifiable state variable, the point can slide along the curve as well as to move the curve. We call such a point that can slide around on its parent objects a *bead.*

Normals and tangents can also be computed for points on 2D curves. The functions to compute them can be determined automatically from the parametric function used to draw the curve by symbolic differentiation. One way to present the position of the normal or tangent to the user is by showing a "crowbar" point that is a point offset from an original point on the curve by the tangent or normal, as shown in Figure 8.2. This attribute provides a connector that can be grabbed just as a point on the curve can. It is computed by simply adding the tangent and position attributes together.

### 8.1.2 2D Geometric Relationships

A wide variety of attributes express geometric relationships among objects. Very often, these relationships can be expressed as functions of a number of points, permitting modularity: any point provided by any type of object can be plugged in. There is no need for special types of controls for each object variety. Attributes that compute geometric relationships are typically used as constraints with their values held constant, rather than being directly controlled by the user.

The most basic 2D geometric relationship is attachment or coincidence of two points. An attribute for attachment is computed by subtracting the positions of the two points. The distance between two 2D points is also a useful attribute. However, using a single distance constraint driven to a driven value does not function well as an attachment constraint. This case violates two of the rules for attributes: it uses a single constraint to remove two degrees of freedom; and the derivatives vanish when the constraint is met. An attachment constraint that uses a subtraction for each coordinate functions much better.

Collinearity of three points is a useful constraint. In 2D, it is straightforward to create by computing an attribute that is the area (determinant) of the triangle formed

by the points, or the distance of one point to the line formed by the other two. Driving either of these two attributes to zero creates the collinearity.

The simplest implementation of a point-on-curve constraint uses an implicit formula for the curve. While this works for a small class of objects such as circles, an alternative approach can be used for the general case of parametric curves. A "bead" point on the curve is created by creating a new point connector allowing its free parameter to be a variable. This bead is attached to the point that is to lie on the object. The point is therefore attached to the object, but free to move along its length.

Relative orientations of objects can be controlled by attributes involving dot products. For example, driving the dot product of two vectors to zero makes them perpendicular. Such a control might also simply drive the length of one (or both) of the vectors to 0, also achieving the desired dot product value. To avoid this, the vectors are typically normalized before the dot product. Inverse trigonometric functions typically have singularities in their derivatives making angle computations using them difficult to use in defining connectors, so it is difficult to place constraints on absolute angles.

The constraint that two line segments, or four points, are parallel can be expressed as the normal of one segment dotted with the other segment's direction vector must be zero.

An attribute that computes the depth of a point into a half-plane is computed by taking the dot product of the normal to the half-plane with the difference vector of the point and a point on the dividing line. This signed distance is useful for creating point inside polygon constraints, which are easily created for convex polygons by a conjunction of point half-plane constraints, or to make point-outside-polygon constraints, as will be described in Section 8.4.2.

### 8.1.3 3D Objects and Constraints

As in 2D, point positions are the main connectors for manipulating 3D objects. Points in 3D can define not only their positions, but also their normals and two tangent vectors to define a coordinate frame.

The position of a point is not only a function of the object it is part of, but also, the transformation hierarchy "above" the object. Surface geometry must be similarly transformed. The same transformation applied to points is applied to tangent vectors, and the inverse transpose is applied to the normal vector. Rather than compute the derivatives of the inverse transform, `DistinguishedPoint` connectors transform their tangent vectors and then take the cross product. Types of `DistinguishedPoint` do not compute normal vectors themselves.

The basic 2D constraints also transfer to 3D. Nailing a point, attaching points, and controlling distances are equally simple. A collinearity constraint is more problematic. The constraint removes two degrees of freedom. Point to line distance with driven to a zero value has the same problems as a 2D distance constraint driven to zero. Point

to line distance is a useful constraint for expressing that a point is on a cylinder with a non-zero radius. A better way to constrain a given point to lie on a line is by creating a connector with a free parameter inside it that specifies a position on the line, and constraining this connector to be coincident with the given point. The "bead" connector can be computed by $u\mathbf{p_1} + (1-u)\mathbf{p_2}$. Such an approach uses two constraints to remove the two degrees of freedom (actually, it uses three constraints, but adds an extra degree of freedom sliding along the line, so the net result is two constraints).

Aligning two coordinate frames requires 6 constraints: 3 to co-locate the origins, and 3 to equate the orientations. A related constraint aligns the positions and the normal vectors of two coordinate frames. This uses five individual constraints: 3 for position, and two that compute the dot product of the normal vector of one point with both tangent vectors of the other. The constraints express a contact-like relationship that allows the objects to slide along one another as if in contact. Such a constraint is best represented by attributes that compute the dot products of one point's normal vector with the other point's tangent vectors.

Expressing that the two planes are parallel, as in the last paragraph, can be done in two ways: by maximizing the dot product of the normals, or by driving the dot products of the normal of one point and the tangent vectors of the second to zero. The former has the advantage that its sign can be used to insure that the normals face the same direction. However, it is a single constraint that removes two degrees of freedom. Therefore, it does not work well to maintain that the planes remain parallel. It can be used to establish the correct relative orientations initially, and the other formulation can be subsequently used when the planes are close to parallel. Several of the systems I have constructed use this strategy: a demon (see Section 7.6.3) switches between a normal dot product controller used to get the orientations close initially, and the normal/tangent dot products used to maintain the relative orientations.

Various point-on-object constraints can be represented by using the implicit formulation of the object. Constraining a point to lie on a plane, cone, cylinder or a sphere can be easily achieved using an implicit representation. Surface beads permit point-on-surface relations for parametric surfaces.

### 8.1.4 Camera Controls

The problem of specifying a viewing transformation or virtual camera configuration is a central problem for 3D graphics. Previous work on the problem of camera control is discussed in Section 2.4.2. My work on using the differential approach for the problem of camera control was first presented in [GW92], and a video accompanying the paper [Gle92b] demonstrates the interaction.

Computer graphics viewing models are defined by linear transformations in homogeneous coordinates. That is, a viewing transformation is defined by a $4 \times 4$ matrix,

and the position that a point appears in the image is given by

$$\mathbf{p} = \mathbf{h}(\mathbf{V}\mathbf{x}), \tag{8.2}$$

where $\mathbf{x}$ is the world-space point that projects to $\mathbf{p}$, $\mathbf{V}$ is a homogeneous matrix representing the combined projection and viewing transformations, and $\mathbf{h}$ is a function that converts homogeneous coordinates into 2-D image coordinates, defined by

$$\mathbf{h}(\mathbf{x}) = \left[\frac{x_1}{x_4}, \frac{x_2}{x_4}\right], \tag{8.3}$$

where the $x_i$'s are components of homogeneous point $\mathbf{x}$. The matrix $\mathbf{V}$ is some function of the camera model parameters.

A perspective camera transformation can be thought of as rigid body camera object that exists in the same world space as the objects it views, just as a camera in the real world. If there are other views in which the camera object is visible, the camera can be manipulated as any other object in the scene. The transformation of the camera into world space can be found by inverting the viewing transform without the perspective projection. Because matrix inversion is difficult to differentiate, simply creating a connector for the inverse transformation given the camera transformation is impractical. However, the function for the inverse transformation can be provided easily for many camera models. Rather than inverting the entire camera matrix that is the composition of several simpler pieces, the pieces are inverted and composed to create the inverse camera transform. Often the pieces are primitive transformations that are easy to invert. Providing the inverse transform for cameras permits connectors on a camera for attributes such as the tip of the lens or the top of the camera. Such points are not only useful for dragging, but also for expressing constraints such as the camera is always relatively right side up.

Rather than controlling a camera by its position in the world, it is often useful to control it by manipulating what is seen in its image. We call such controls *through-the-lens* controls. The most basic through-the-lens control is an attribute that computes the position where a point in the world projects onto the film plane of the camera (the image seen through the camera). Equation 8.2 is used to compute an attribute that serves as the control. These controls are independent of the variety of camera model. Any viewing transformation can be controlled, although Equation 8.2 may be replaced by some other function if a non-standard camera model is used.

It is important to realize that the image position of a point depends on both the camera and the point's position in the world. A through-the-lens control can be used to affect either, or both, depending on which object's parameters are free to change. Through-the-lens controls are a particularly important type of attribute because they provide a way to couple the 3D world to the space of 2D input devices, permitting manipulation of 3D objects and viewpoints by pointing to screen positions.

A single through-the-lens control does not specify enough information to usefully control a 3D object or camera transformation by itself. With the many degrees of freedom involved, the 2 controls of a through-the-lens point are underdetermined, and good default behaviors are difficult to specify with the optimization objectives. Typically, other constraints are used in coordination with through-the-lens controls to provide desired behaviors. Many examples will be given in Section 8.2.

The same constraint relationships applied to points on 2D objects can also be applied to through-the-lens controls. For example, an attribute might measure the distance between two image points. Placing constraints to keep through-the-lens point controls within in a region, for example on the screen, is often useful.

Through-the-lens controls are the most basic element of *appearance-based manipulation.* Such an approach permits the user to control a 3D world by specifying what is seen in a picture of it. It is useful because images are 2D and therefore map nicely to common input devices, and also because often an image is the goal of a graphical application.

### 8.1.5 Manipulating Lights and Materials

The configuration of lighting and material properties can also be controlled by a variety of attributes. Like cameras, lights are typically objects in the scene and can be manipulated as such. This is especially useful for point and spot light sources. With both types of lights, connectors for the position of the bulb is the most obvious control. For spotlights, points on the aperture, both in the center and around the rim, are useful not only for direct grabbing, but also for use in the shadow manipulation techniques of the next section.

Light intensity and material surface properties are parameters that can be represented in the state vector. This permits their values to be constrained.

One appearance-based method for controlling lights and material properties is to directly manipulate the color of points in the image, for example by attaching a point's color to a set of sliders. Techniques for manipulating point colors are presented for changing material surface colors in [HH90] and for changing light colors in [SDS+93]. Using the differential approach to control point colors permits these techniques, but also can be used to alter other parameters that affect apparent colors such as surface orientations and light positions.

The equation that computes the color that a particular point appears must be known to the system so it can draw it. A shading model computes the color from the surface geometry, lighting, and surface properties. The color that a point appears is the result of a physical process which has been modeled to varying degrees of accuracy [Hal89]. Modeling and rendering systems most often use simpler models. The parameters to these models serve as the controls which are used to specify surface properties. Therefore, a user interested in the colors in an image must understand this model, even though

it is merely a historical artifact of computer graphics research.

The most commonly used shading model [FvDFH90, Sil91] divides lighting into three components, diffuse, specular, and ambient parts. A surface is specified by three colors which define how the surface reflects each of these components. The color of a point is

$$\mathbf{c} = \mathbf{c}_a * \mathbf{i}_a + \mathbf{c}_d * \mathbf{i}_d + \mathbf{c}_s * \mathbf{i}_s, \tag{8.4}$$

where the asterisk denotes component-wise multiplication, $\mathbf{c}_a$, $\mathbf{c}_d$, and $\mathbf{c}_s$ are the surface color properties, and $\mathbf{i}_a$, $\mathbf{i}_d$, and $\mathbf{i}_s$ are the intensities of each type of light at the point. Each is typically represented as a 3-vector containing RGB color values.

If we know how much light is available at a point $(\mathbf{i}_a, \mathbf{i}_d, \mathbf{i}_s)$, we can manipulate the color of the point in order to control the surface properties, $(\mathbf{c}_a, \mathbf{c}_d, \mathbf{c}_s)$. This is most interesting when we control the colors of multiple points which share the same parameters. In the special case of fixed lighting and geometry, all of the apparent colors are linear functions of the surface properties. This is exploited by the system described in [SDS+93].

The amount of light which strikes a point can be computed by summing the contributions of each light source. The amount of ambient light, $\mathbf{i}_a$, is merely a scene-wide constant. For other varieties of light, the contribution of each light source is summed. The amount of diffuse illumination given by

$$\mathbf{i}_d = \sum_{i \in lights} \mathbf{i}_i (\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}_i), \tag{8.5}$$

and the specular illumination by

$$\mathbf{i}_s = \sum_{i \in lights} \mathbf{i}_i (\hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^{shi}, \tag{8.6}$$

where $\mathbf{i}_i$ is the intensity of light source $i$, $\hat{\mathbf{n}}$ is the point's unit normal vector, $\hat{\mathbf{l}}_i$ is the normalized vector from the point to the light, and where $\hat{\mathbf{h}}$ is the normalized vector that bisects the vectors from the point to the eye and light, and $shi$ is a parameter which controls the highlights size.

Controlling the colors of points is useful in controlling the lighting. Manipulating the color of a point can alter the intensities of the lights that contribute. This provides a way of achieving desired color effects when multiple colored lights are used, as in stage lighting. Colors can even be used to control the geometry of lighting. Altering the color can cause a face to turn to towards or away from a light, or cause a light to move so it provides the proper amount of illumination.

Using the lighting equation as a control has a great deal of potential, but, it is problematic. Even simple lighting models are complicated expressions of many parameters. More complicated lighting models may not even be expressible as closed form differentiable expressions.

Even in the simple shading model used by the Iris hardware and in Bramble, many terms of the equation do not function well as controls. In particular, the specular component of lighting is particularly problematic. The amount of specularity at a point is given by Equation 8.6. Most obviously, it involves an exponentiation which gives it very non-linear behavior with rapidly changing derivatives. Also, it involves normalizing a quantity twice in computing the normalized half angle vector. Although this makes it difficult to use color control to position specular highlights, the techniques of the next section can handle such tasks by treating specular highlights as the reflection of the light source.

### 8.1.6 Shadows and Reflections

We treat reflections and shadows in a similar manner: we compute where the image of a particular point appears on a particular surface. Shadows are the simpler case. A point is the shadow of another if the two points and the light source are collinear. This is typically implemented by creating a bead on the ray from the light source through the shadowing point. The bead can either be attached to some point that is to be shadowed or constrained to lie on the surface of the shadowed object. An additional constraint could be added that insures that the occluding object is between the shadow and the light source.

The explicit use of the ray between the light and the shadow also serves as a mechanism to display the shadowing. With available graphics hardware, general inter-object shadows cannot be drawn efficiently, so drawing the line serves as a feedback device, as shown in Figure 8.3. Manipulating the shadows can control the position of the light, the shadowing object, or both.

When the shadow is cast onto a planar surface, its position can be computed directly. The general projection matrix can be used to compute where the shadow ray hits a plane. This is a simple computation, given by

$$\mathbf{p} = \mathbf{h}(\mathbf{S}\mathbf{x}), \tag{8.7}$$

where $\mathbf{h}$ is given by Equation 8.3, the function which converts from homogeneous coordinates, $\mathbf{x}$ is the position go the point to be projected, and $\mathbf{S}$ is the projection matrix. In [Bli88a], the matrices are derived for projection from point and distant light sources onto a ground plane. More generally, the matrix can be computed for a light source point $\mathbf{l}$ in homogeneous coordinates onto a plane $(\mathcal{P}_a x + \mathcal{P}_b y + \mathcal{P}_c z + \mathcal{P}_d = 0)$ by[1]

$$\mathbf{S}(\mathbf{l}, \mathcal{P}) = \begin{bmatrix} \mathcal{P}_b \mathrm{l}_y + \mathcal{P}_c \mathrm{l}_z + \mathcal{P}_d \mathrm{l}_w & -\mathcal{P}_b \mathrm{l}_x & -\mathcal{P}_c \mathrm{l}_x & -\mathcal{P}_d \mathrm{l}_x \\ -\mathcal{P}_a \mathrm{l}_y & \mathcal{P}_a \mathrm{l}_x + \mathcal{P}_c \mathrm{l}_z + \mathcal{P}_d \mathrm{l}_w & -\mathcal{P}_c \mathrm{l}_y & -\mathcal{P}_d \mathrm{l}_y \\ -\mathcal{P}_a \mathrm{l}_z & -\mathcal{P}_b \mathrm{l}_z & \mathcal{P}_a \mathrm{l}_x + \mathcal{P}_b \mathrm{l}_y + \mathcal{P}_d \mathrm{l}_w & -\mathcal{P}_d \mathrm{l}_z \\ -\mathcal{P}_a \mathrm{l}_w & -\mathcal{P}_b \mathrm{l}_w & -\mathcal{P}_c \mathrm{l}_w & \mathcal{P}_a \mathrm{l}_x + \mathcal{P}_b \mathrm{l}_y + \mathcal{P}_c \mathrm{l}_z \end{bmatrix}. \tag{8.8}$$

[1] Thanks to Pat Hanrahan for lending us this very useful matrix.

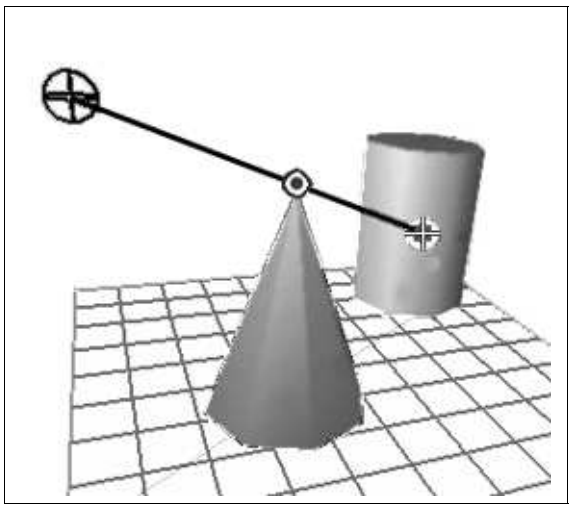

**Figure 8.3:** Lights and objects are manipulated by controlling a shadow. Because inter-object shadows cannot be drawn with the available graphics hardware, a line is used to connect the light, the shadowing point, and its image.

The shadow matrix is useful for drawing shadows using the standard rendering pipeline, as described by Blinn [Bli88a]. We merely concatenate the shadow matrix and the viewing transform, and redraw the scene in a dimmed grey color. Simply drawing a dark color on the ground plane fails when the shadows go off the rectangular stage surface and seem to float in space. I have used two methods to give the illusion of dimming. One is to draw the shadows using a dither pattern and the sky color, which is chosen to be darker than the ground color. An alternate approach is to choose the ground and sky colors such that turning a particular bit off in the ground color dims it, but does not affect the sky color. Shadows are then drawn using a write mask so that only this one bit is affected.

Positions of shadow beads can be computed using equation 8.7 and directly constrained and controlled. A system can permit the user to grab shadow points and to drag them. This permits not only the interaction techniques of [HZR+92], but manipulation of lights as well.

The spot caused by a spotlight onto the reference planes is also useful. Drawing this spot in a bright color, using techniques like those used for shadows, gives an indication of where the spotlight is aimed. This ring is an important part in some lighting effects. A spotlight can be manipulated by controlling the projections of aperture points on the ground. This is used in the Luxo lamp example of the introduction.

The same projection techniques apply to mirrors as well as to shadows. We use the same matrix, except that we place the projection point at the virtual eyepoint rather at the light source. The virtual eyepoint is the place where a viewer would have to stand to see the reflected image through the surface if the surface were transparent [Ups89],as

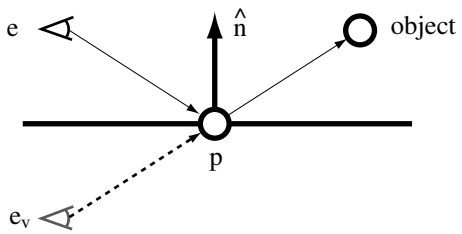


**Figure 8.4:** An observer with eye point **e** sees the reflection of an object at point **p** on a shiny surface with unit normal vector $\hat{\mathbf{n}}$. The virtual eyepoint $\mathbf{e_v}$ is where the eye would be located to see the image of the object at **p** if the surface were transparent.

shown in Figure 8.4. The position of this point is computed by

$$\mathbf{e}_v = \mathbf{e} + 2((\mathbf{e} - \mathbf{p}) \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}, \tag{8.9}$$

Where $\mathbf{e}$ is the position of the eyepoint, $\mathbf{p}$ is the position of the surface point on the mirror, and $\hat{\mathbf{n}}$ is the surface normal at the point. Although there is potentially a different virtual eyepoint for each surface point, all points on a planar surface share the same virtual eyepoint. Explicitly computing $\mathbf{e}_v$ is preferable to other reflection formulations as it provides a geometric position as an intermediate result that can be examined (guideline 6).

Computing the virtual eyepoint allows the projection matrix of Equation 8.8 to be used to compute the transformations that place reflections on a planar mirror. This allows reflections to be drawn using the rendering pipeline. To provide proper occlusions in a z-buffer, we draw the objects slightly above the surface of the mirror, with a height equal to the inverse of the distance from the mirror so that closer objects occlude ones further from the mirror. Also, because all the objects in the reflections are flat, their normals are invalid, so lighting calculations cannot be done. The positions of points in planar mirrors can serve as controls to manipulate the mirror, the reflected point, and even the camera.

The analogy of techniques for manipulating shadows and reflections extends to nonplanar surfaces as well. Although there is no way to efficiently draw these reflections with available graphics hardware, a line connecting object, image, and virtual eyepoint serves for feedback. Similarly, enforcing collinearity between these three points permits using the manipulation of one to control the others. An example of this is shown in Figure 8.5.

Reflection techniques can be used to position specular highlights on surfaces. The light source is placed such that it is seen in the reflection on a surface point. By sliding the point as a bead around on the surface, the highlight can be positioned. This can be used to position the light source or to alter the surface geometry.

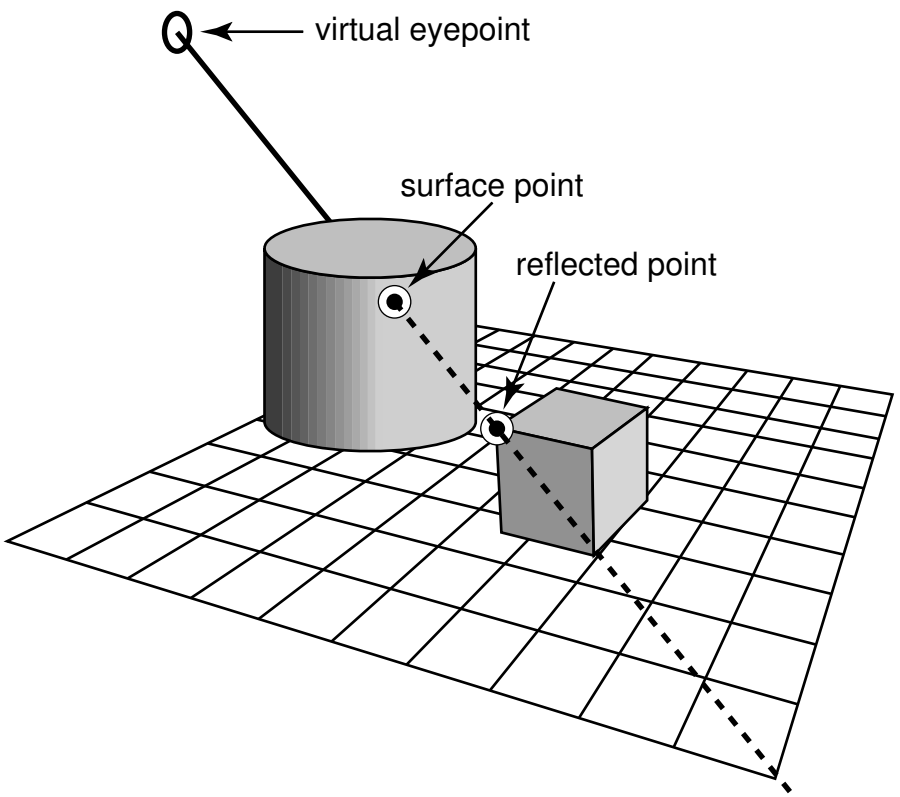


**Figure 8.5:** The cube is controlled by manipulating the position of its reflection in the cylinder. Since the reflection in the curved surface cannot be drawn, the line connecting the point of the cube, its reflection, and the virtual eyepoint is drawn.

### 8.1.7 Free Form Curves and Surfaces

Free-form curves and surfaces are traditionally a difficult problem for interface design. Typical approaches to creating such objects require devising representations that are sufficiently expressive, are convenient for the user to manipulate directly, and have good mathematical properties. This typically leads to interfaces like the control points of B-Splines, that sacrifice usability. Some of the difficulty stems from the fact that the representations also serve as the parameterizations of the objects. Because of this, it would seem that the differential approach would be a natural solution.

For many types of curve and surface representations, the differential approach can be applied. For many types of parametric surfaces such as B-splines and NURBS, the positions of points are simple functions of the parameters that can be computed as connectors and manipulated differentially. Several of the approaches in the literature are variants of this, using different schemes for computing parameters changes. For example, Welch, Gleicher, and Witkin [WGW91], Welch and Witkin [WW92], Fowler [Fow92], and Hsu et al.[HHK92] all use various constrained optimization techniques to map manipulation of points on surfaces to the underlying parameters. This permits the objects to be manipulated by controlling points on them, rather than control points. In some cases, users may prefer to use control points to manipulate curves or surfaces. Since the positions of the control points can be computed from the curve or surface, control points can be provided even if another representation is used. This allows, for example, to use Bezier control points to manipulate a B-Spline curve.

Unfortunately, the differential approach of this thesis is insufficient for adequately addressing the issues in manipulating free form curves and surfaces for many reasons:

- free form objects have too many degrees of freedom;
- free form objects require global control for effective manipulation. Such control can be provided only with attributes that compute properties of the entire surface, or large regions of it;
- free form objects often need fine detail local control, requiring adaptive subdivision;
- many of the constraints that are used to sculpt free form objects' are specialized cases that can be handled more effectively for the large numbers of degrees of freedom.

In short, the methods of this thesis do apply to the manipulation of curves and surfaces, as we showed in [WGW91], but, by themselves, the methods do not address many of the difficult issues and more specialized methods apply. Controlling free form curves and surfaces is a very important problem, and is therefore well studied. Some interesting optimization-based approaches are explored by Celniker and Gossard [CG91] and by Welch and Witkin [WW92] and [WW94].

## 8.2 Strategies for Interaction

The previous section described a large number of attributes that can be computed and provided as connectors on objects. With the differential approach, we can control the objects by attaching controllers to the connectors. In this section, we consider some strategies for determining what controllers to attach to which connectors at what time.

### 8.2.1 Presenting the Abstractions to the User

The most obvious way to employ the differential approach is to provide the abstractions to the user as directly as possible. At an extreme, the user could be presented with a schematic representation, like the diagrams of Section 1.3.3. While this graph editing approach has been attempted in systems such as the SPAR Modeling Testbed [FW88], Condor [Kas92] and ThingLab [Bor86], I do not believe it is in the spirit of direct manipulation, and prefer to hide such representations from the user.

The direct application of the differential approach gives the user a palette of connectors to which controllers can be applied. Connectors that represent positions can be controlled by attaching them to the mouse. Non-geometric values can be controlled with sliders or similar widgets. To specify multiple controls, the user might either nail connectors to their current values, or place goal points for connectors to seek.

Presenting the abstractions directly to the user has a number of benefits. It permits the user to select controls that are applicable to their problem, and to mix and match controls as needed. It maximizes flexibility over what the user can do. Such a direct approach is a useful strategy for testing out new types of controls, as shown in the scene composition program of Section 9.7. By utilizing efficient mechanisms for selection, a large array of attribute types can be provided for control.

The advantages of this direct application are also its most serious problem. The question of how to present the controls to the user can be difficult, especially as the number of types that are available grows. The interface must help the user understand the potential choices and to find the controls that are applicable to their task. While the flexibility of a range of controls is useful, it also means that a user might need to spend time deciding which control to use and how to employ it. A system must show the user what is being controlled and explain the behavior of the objects. The issues of scalability also arise as the user adds more controls, creating more complicated behaviors and worse performance.

One variety of direct application of the differential approach is the "live world." A live world is an environment where just about any point can be grabbed and manipulated. In such an application, objects, shadows, reflections, lights, or just about any other entity can be grabbed and dragged. This provides a uniform interface for a wide variety of tasks. However, it still does not solve the problems. A user must be made aware of what can be dragged and what cannot. Some mechanism for specifying what

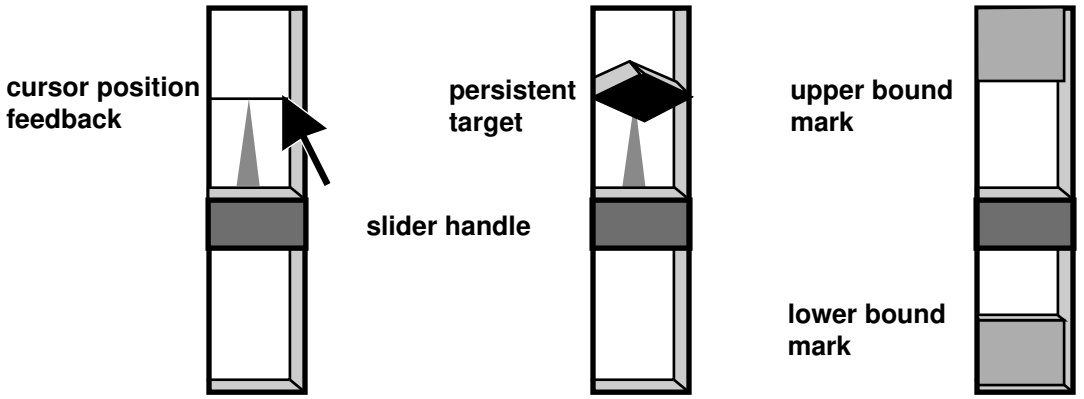


**Figure 8.6:** A differential slider. Left: feedback shows not only the value specified by specifying dragging, but also the actual value of the attached connector. Middle: the `GoTowards` controller used to drive the connector toward the specified value can be retained after dragging is completed to nail the value in place. The persistent value is displayed by the diamond. Right: users can place boundary constraints with the sliders. Upper and lower bounds are displayed as greyed regions.

might change when a point is grabbed is important for controlling things like shadows, where many things might be affected by a single control. Some mechanism for permitting the user to specify the desired behavior in under-constrained cases, for example by permitting grabbable points or objects to be nailed in place, needs to be provided.

### Differential Sliders

To present controllers and connectors the user directly there needs to be a mechanism for controlling any connector output, even ones whose output is simply a dimensionless scalar value. The mechanism must permit the user to specify the various controller types, and provide values for those types that require them. Bramble has a widget called *differential slider* that is a variant of a standard slider designed to control a connector with the differential approach. A basic differential slider is shown in Figure 8.6. Better designs that are more self-revealing and easier to use are certainly possible.

A standard slider converts the position of the mouse into a value for its attached parameter. For the differential approach, such a slider cannot be attached to a connector since the value cannot be set directly. To create a basic differential slider, a `GoTowards` controller is connected to drive the value towards what is specified by the mouse, rather than directly specifying new values for its attached parameter when dragged. Because the connector's value may not perfectly track that of the mouse, both values are displayed to the user.

The ability to nail a slider's value in place is useful feature for a differential slider.

Such a facility can be implemented by simply retaining the `GoTowards` controller beyond the completion of the dragging operation. With the Bramble differential slider, this is done by holding a modifier key as the dragging button is released. Key presses can be used to create controllers that drive the slider to its minimum, maximum, or middle value.

A differential slider can be used to apply other controllers as well. For example, the Bramble slider can place boundaries on the value by pressing the third mouse button either above or below the present value. The slider displays the current boundaries, as shown in Figure 8.6. Snaps can also be placed on the slider's value. The slider must display where the snaps are and provide feedback for when the slider is snapped.

### 8.2.2 Drawing or Modelling with Constraints

Drawing or modelling with constraints can be seen as a variant of directly providing the abstractions of the differential approach to the user. In such an approach, a user declaratively specifies relationships among parts of the model. Differentially, this entails creating `GoTowards` controllers for selected attributes that compute the relationships. Often, this is coupled with dragging: the user can drag pieces of the drawing either by placing and dragging positional constraints, or dragging the model subject to the constraints.

The idea of using constraints in interactive drawing and modeling dates back to Sketchpad[Sut63], the earliest system. Since then, there has been considerable interest in the approach of declaratively specifying parts of the drawing. Some of the advantages of the constraint-based approach are:

- the user can specify what is most convenient, in any order;
- the constraints can give structure to the model, potentially embedding the semantics of the thing being modelled;
- the constraints can be used to specify exact relationships in the model precisely;
- the constraints are persistent so they maintain previously established relationships to avoid redundant work in reestablishing them after editing.

Many difficult issues have limited the success of constraint-based systems for drawing. Not only must a system be able to solve constraint satisfaction problems, but it must make it easy for users to specify, debug, and edit constrained models. Constraints change the nature of interaction in a graphical application. Without them, actions only affect the objects to which they refer. For example, dragging an object in a traditional drawing program moves only the object. With constraints, this locality is lost: altering one object may cause other objects to be affected. This global nature of constraint operations is at the core of many of the difficult issues in employing constraints.

Without user specified constraints, graphical objects have fixed behaviors. For instance, an ellipse in a drawing program behaves like an ellipse. The system designer can design a good, usable behavior which the user can learn and apply to all ellipses. When user specified constraints among objects are introduced, the situation changes. To begin with, the behaviors can become more complicated because of interactions among objects. Each combination of objects and constraints will have its own behavior. These behaviors are specified by the user in terms of the constraints; the user is effectively programming.

As in more traditional programming, complexity in the constrained behavior of a graphical model becomes a problem when it has bugs, e.g. when the behavior is not what is desired or expected. The most obvious form of bug is when the constraints force the model into a configuration that is not what the user desires, or the constraints prevent the user from achieving a desired configuration. Another class of constraint bug stems from bad constraints where solutions cannot be found, either because of conflicting specifications or solver failures.

Because constraint errors occur, interactive graphical applications which provide constraints to users must deal gracefully with bad situations, such as conflicting or redundant constraints. Underdetermined models must also be handled, as it is impractical to expect the user to specify all possible degrees of freedom. Because of the potential for errors, it is crucial to aid the user in understanding the complex behaviors of constrained models.

The differential approach helps with some of the issues in creating constraint-based applications. Continuous motion facilitates understanding the behavior since users are able to employ their perceptual skills to help understand change. The methods for implementing the differential approach handle under and over determined cases. The dynamic implementation of the differential approach can serve as a backbone to providing a constraint-based system where users can experiment with their models in order to comprehend and debug them.

The Briar drawing program, described in Section 9.1, builds on the differential approach to provide a strategy that addresses other issues in constraint-based tools as well. By using constraints only to maintain existing relationships during direct manipulation, Briar is able to avoid problems with conflicting or unsolvable constraints and unpredictable selection of constraint solutions. A key to making systems like Briar possible is the ability to enforce constraints during continuous-motion, direct-manipulation dragging, as provided by the differential approach.

Although many modern constraint-based systems, such as Briar, show promise in addressing the issues of the approach, it is not certain that the issues can be sufficiently resolved to become the standard and dominant tools. In particular, issues of scalability may be the ultimate Achilles heel for constraint-based drawing and modelling.

### 8.2.3 Building Interaction Techniques

In the previous sections, we discussed how the abstractions of the differential approach can be presented directly to the user. In this section, we consider leaving them in the hands of the interaction technique designer. The idea is to use combinations of controls in ways that define desirable behaviors for traditional direct manipulation style interactions.

The typical way the differential approach is employed is to use a control to provide interactive dragging, but to define other constraints so the correct behavior occurs. Rather than having the user define the constraints, the interface designer can choose them. Objects or handles are predefined and can be given carefully designed behaviors.

For the designer of direct interaction techniques, the differential approach provides a new set of abstractions. The approach can lead to techniques that could not have been implemented using traditional techniques. Other times, the abstractions are applied to more conventional interactions. In fact, most of the examples given later are simply recreations of existing interaction techniques. The approach may be most practical for prototyping interaction techniques: the technique can be designed using the tools of the differential approach, permitting it to be quickly defined and evaluated. Then the mathematics can be re-derived to compile the interaction technique into a more traditional implementation.

Because controls are used in predefined combinations, many of the most serious drawbacks of the differential approach and constraint-based systems are avoided:

- the number of controls is a small constant that does not grow as the model becomes larger;
- the combination of controls can be checked to insure that they are well behaved before being handed to users;
- the behavior of objects is explicitly designed, rather than just coming about as a byproduct of placing constraints. This can leave the specification of interactive behaviors in the hands of the "trained professionals."

Many of the same arguments are given for the design of the Siri constraint-based programming language [Hor93]. Siri uses constraint techniques within objects to help define their behavior, but uses more conventional methods for inter-object communication.

There are two main ways that the differential approach is used for defining interactive behaviors. One is to define types of objects that behave as desired. The other is to define handles that have particular functionality. This latter, more common category typically involves creating a number of constraints in addition to the mouse attachment during a dragging operation. Many examples are given later in this chapter.

**Figure 8.7:** A 3D image overlaid on top of a corresponding real image. Our goal is to place the virtual plant on the real table. In each image, there is a table (denoted by the white rectangle for the graphics image). Initially, the images do not correspond because the virtual camera is not in the same place as the camera used to create the real image.

To define the behavior of a object that will be manipulated using the traditional direct manipulation interface of dragging the position of a single point at a time, the differential approach offers advantages. First, constraints can be used internal to the object to define relationships that must be maintained as the object is dragged. Secondly, by providing a way to implement the dragging, it spares the object designer the effort of mapping from position values to parameter values. It also permits a uniform mechanism to be used to drag all points on all objects.

### 8.2.4 A Concrete Example: Aligning a Camera to an Image

We now consider a very specific interaction technique as an example, and discuss how it can fit in with the strategies of this section. The problem is to align a synthetic image of a 3D scene with a real image by configuring the virtual camera creating the synthetic image. We assume that both the real scene and the synthetic scene each have a table of identical size in them. An example is shown in Figure 8.7.

With through-the-lens controls and the differential approach, the correspondence problem can be solved easily. Through-the-lens controls are used to drag the corners of the virtual table to their corresponding positions in the image, causing the camera to be moved. Each control is locked as it is placed. Manipulating the positions of the four corners of the table causes the virtual camera to be placed in a position where the two

**Figure 8.8:** The synthetic and real images are registered by successively dragging the corners of the table to their corresponding positions in the image.

images correspond. This process is shown in Figure 8.8.

The registration task is an excellent example of the benefits of the differential approach. It solves an important and useful task that would be extremely difficult with traditional interaction techniques. In order to define the interaction technique without the differential approach, the mathematics to determine the camera configuration from the corners of the rectangle would have to be derived. While such a derivation is possible, it would be extremely difficult to do in a robust manner as it is an overdetermined problem. With the differential approach, it is a straightforward combination of four 2D point controls. The method over-determines the solution, specifying eight controls for a camera which has only seven degrees of freedom. However, the methods of Chapter 3 can handle these over-determined cases.

The table registration required the use of four through-the-lens point controls. Those controls could have been specified either by the user, or by an interaction technique designer building a mechanism for the particular task of image registration. If the abstractions of the differential approach were directly provided to the user, the registration task would require the user to freeze the position of the table in the world, enable the camera to be controlled, select through-the-lens position controls for each corner of the table, and then drag these controls. In a more constraint-based interface style, there might be a command for dragging points with through the lens controls, which the user could successively apply to each corner of the table.

Because image registration is an important task, an interaction technique designer might want design an interface for it. For example, the designer might create the virtual table as a special object that when its corners are grabbed, the system knows to apply a through-the-lens control on the point to manipulate the camera, and to leave these controls locked in place even after they've been manipulated. The user would see a command to create the alignment table, and would directly manipulate table corners, but the abstractions of the differential approach would be hidden, serving simply as a tool for the interaction technique designer.

## 8.3 Sources of Constraints

The basic idea behind each of the strategies of the previous section is to provide constraints so that manipulation operations like dragging have the desired effects. The major difference in the strategies is how these constraints were specified, whether they were provided by the interaction technique designer or by the user. In this section, we consider what constraints can be useful for manipulation, and how they are used in conjunction with dragging to create interaction techniques. This section is organized by "sources" of constraints, the kinds of things that specify constraints on manipulation.

The general problem in manipulation will be to handle the underdetermined nature of dragging. Our input devices will undoubtedly specify far fewer degrees of freedom than the model contains, or even than typical objects contain. For example we might control an articulated figure with a 6 degree of freedom tracker or control a 3D position with a mouse.

The large number of degrees of freedom problem is especially important in 3D manipulation. Unlike in the real world, the objects we manipulate with a 3D user interface can float freely in space, and therefore have extra degrees of freedom. The problem of 3D manipulation is to somehow control these degrees of freedom with the limited input devices we have. One approach to this problem is to develop better input devices. For the work in this thesis, I have considered only the mouse. However, the differential approach is input-device-independent, and even with better input devices, the constraint-based approach applies – in the real world we still use constraints for manipulation even with our dextrous hands.

When we encounter an object with many degrees of freedom in the real world we employ a variety of tactics to manipulate it. We use manipulators in parallel, for example by coordinating our hands or fingers, or we create constraints, either by using an extra hand or finger or by using interactions between objects. For example, we might turn 3D problems into a 2D one by placing the objects on a flat surface, or for more complex manipulations we might build a jig to limit the objects' behaviors to make it easier to achieve the desired manipulations.

However, we normally design our objects so that they don't have as many degrees of freedom. The behaviors of most objects we manipulate are constrained by their relationships with other objects or their structure constrains them to move only in the correct fashion. For example, operating a door requires only one degree of freedom (its hinge) or two (its knob and hinge) and car steering wheels and levers rotate only in useful ways.

These real world tactics for manipulation all rely on constraints on object motion to simplify the manipulation problem. The sources of these constraints vary: they come from the mechanical structure of the object, interactions among objects, conventions on the uses of objects, or from other hands or fingers. In this section, we will use the same approach to address the problem of manipulation in interfaces. The general idea

for creating an interaction technique will be to provide a sufficient set of constraints so that control of a single point defines a desired behavior. For defining 3D interaction techniques using the mouse, we will attempt to define a sufficient set of constraints so that the two degrees of freedom specified by a through-the-lens control are sufficient to control the 3D object.

### 8.3.1 Intrinsically Constrained Problems

For some manipulation tasks, the object being manipulated is intrinsically constrained sufficiently. For example, even in a 3D world, dragging a point along a fixed plane, such as the floor, is a 2D problem. So, if a point is by definition in a fixed plane, it is sufficiently constrained so that a 2D input device can be used to control it. An example of such a point would be shadows on the floor. This is part of the reason for the interest in shadow manipulation.

### 8.3.2 Artificial Constraints

A common way to create interactions is to create synthetic constraints based on some user command. For example, based on the state of a modifier key or a command mode, the system might constrain object motions to rotate about a particular axis or translate in a particular direction.

An example 3D interaction technique using artificial constraints is the mousepole, introduced in Section 7.8. The mousepole allows the user to position a point in 3D using the mouse by constraining the point to lie in the plane parallel to the ground. The ray cast from the mouse position defines a unique point on this plane. Depressing a button switches the mousepole to operate in a plane perpendicular to the ground. We draw a vertical pole from the point to the ground in order to indicate height, hence the name mousepole.

The advantages of these artificial constraints are that they can be applied to any object and are easy to create. However, it is easy to make such interfaces complicated by including large numbers of modes, modifier keys, and commands. It can be difficult to make such interfaces self-revealing. Using a standard of artificial constraints on all objects has the advantage of uniformity, but has the problem that it does not permit objects to special behavior.

### 8.3.3 Object Semantic Constraints

The graphical objects that we manipulate often have structure or semantics that may dictate how they should behave when manipulated. This behavior can often be expressed as constraints, sometimes corresponding to the mechanical structure of the object, and sometimes to the intended purpose or conventions of use of the objects. For

example, the Luxo lamp of the introduction has many pieces each with many degrees of freedom. However, the mechanical structure of the lamp significantly reduces the degrees of freedom in the lamp, making it much easier to control.

Many of the objects we must manipulate have only a few degrees of freedom: useless motions are constrained away. There are many example of objects whose behavior is sufficiently constrained so that one or two controls are sufficient, for example, steering wheels, airplane yokes, doors, levers, or the handle of a slot machine. There are some implied assumptions in these manipulations, for example when the handle of a slot machine is grabbed, we assume that we are attempting to pull the lever, not move the machine.

Sometimes, the constraints on objects stem not from the objects' mechanical structure, but rather from conventions on how they are used. For example in moving a piece of furniture, we are most typically interested in sliding it along the floor or turning it, not necessarily lifting it or flipping it over. In such cases, it is conceivable to build interfaces which imbue the objects with the constraints that cause the more common behavior, and require some less direct method for other manipulations. In analogy to the real world, I can push the furniture around on my floor, but to lift it or flip it, I need to get some extra help. The desire to perform non-standard manipulations also exists for mechanically constrained objects, for example, we might want to rip the head off the Luxo lamp.

Often, the manipulation task provides sufficient information to adequately constrain objects so that simple point manipulations suffice. For example, positioning a picture on a wall or a lamp on a desk are both 2D problems. Pulling the arm of a slot machine is a one dimensional problem if the model behaves like a slot machine. I call the strategy of attempting to use the natural constraints of the task to create interaction techniques *manipulation from structure.* Such constraints are called "context specific constraints" by [Hou92]. These constraints that arise from the structure of the model are advantageous because they only restrict the user from performing operations which are typically undesirable, they are often already inherent in the parameterization of the model, and they are easily understood by the user.

There are many issues in employing manipulation from structure in realistic systems. For one, it requires that the system have some knowledge of what the objects are, and how they should behave — a collection of polygons might look like a painting which should remain hung on a wall to the user, but the system must not only know how paintings are to behave, but also how to identify them. The structure of the model must be created in a manner that has the proper semantics. When manipulating such models, it can be difficult to know if the model is sufficiently constrained such that 2D manipulations will provide enough control. While the differential approach does not directly address these problems, it does make it far easier to explore manipulation from structure by providing a vocabulary with which the semantic constraints can be expressed and by mapping controls to whatever parameters are used to represent the

models.

### 8.3.4 Handles

Handles are particular points associated with objects that can be grabbed. Handles are often given specific meanings as to the manipulation that they perform. For example, in a typical Macintosh direct manipulation interface, a graphical object has particular handles that cause it to be scaled.

Handle behavior can be defined by associating each handle with a set of constraints that are applied to the object as the object is manipulated. For example, to create the Macintosh-style scale handles, the position of the opposite corner of the object must be pinned down while the handle is dragged. Bramble's grab and ungrab hooks were specifically designed to help define handle behaviors with constraints.

### 8.3.5 Widgets

Although building the constraints required for manipulation into each graphical object has many attractions, it does have some serious drawbacks. Most obvious is that behavior must be built into each object, and that each object must have sufficiently well-designed behavior so that it is manipulable and that the constraints are apparent to the user.

An alternative to giving every object a behavior is to define special objects which have behaviors and to manipulate the other objects by attaching them to these *widgets*[2]. These widget objects can be specially designed to have desirable and self-revealing behaviors. A widget can be thought of as a tool for providing a specific type of manipulation. Widgets have become ubiquitous in graphical interfaces. Most graphical user interfaces now have users specifying values with widgets such as sliders, dials, and buttons.

With the differential approach, connections are bidirectional. If a widget displays a value, it can also serve to control the value. The functions that map from the screen position of the input device to the values stored by the widget do not need to be provided, only the forward direction functions required for drawing. This is exemplified in the fuel gauge widget example shown in Figure 8.9. The complete code for this example is presented in Section A.2.1, but a simplified version is defined using the Bramble `define-shape` function:

[2]The definition of widget here is slightly different than that of [CSH+92]

**Figure 8.9:** A fuel gauge, define with Bramble's `define-shape` function.

```
(define-shape gas-gauge (val) (0)          ; object with 1 variable
  ((t (+ .2 (* 2.7 val)))                  ; convert percentage to angle
   (x (- 0 (* .3 (cos t))))                ; x position (left is 0)
   (y (* .3 (sin t))))                     ; y position
  (drawf                                   ; define draw method
   (prog (color gl-black) (linewidth 3)    ; set color and line width
         (arc 0 0 .3 0 1800)               ; draw the shape of the gauge
         (move -.3 0) (draw .3 0)          ;     and the bottom
         (move 0 0) (draw x y) ))          ; draw the needle
  (> val 0)                                ; limit the needle to valid
  (< val 1)                                ;     range
  (handle x y) ))                          ; put  a handle on the needle
```

The values that are computed for the position of the handle tip, used for drawing, also serve as a handle. No code for mapping from the mouse position to the angular value was required.

Bramble provides a small set of widgets that can be used in windows outside of the views. However, widgets are often created as graphical objects that exist in the world with the user objects. Because Bramble's world is 3D, the widgets actually exist in 3D space, even though they are usually flat onto the $z = 0$ plane. However, since the widgets are 3D objects they can be transformed into other places. For example, we might place gauges on a model of an instrument panel as shown in Figure 8.10.

The behavior of a widget can be defined using the same techniques as used for other objects. For example, to create a gauge with a dial, the arrow could be constrained to have its endpoint at the center of the dial, and its orientation equal to the value of the widget.

An example of some experimental widgets in Bramble are the aircraft gauges. The set of gauges include an altimeter, a heading indicator, and an artificial horizon, shown in Figure 8.10. In each case, the gauges draw themselves based on the values of their parameters, and provide `DistinguishedPoint` connectors on their moving parts. For example, either hand of the altimeter or many points on the heading indicator's compass ring can be grabbed.

This set of airplane gauges is potentially interesting because it provides a 2D display, and therefore control, of the orientation of a 3D object. However, I have not found

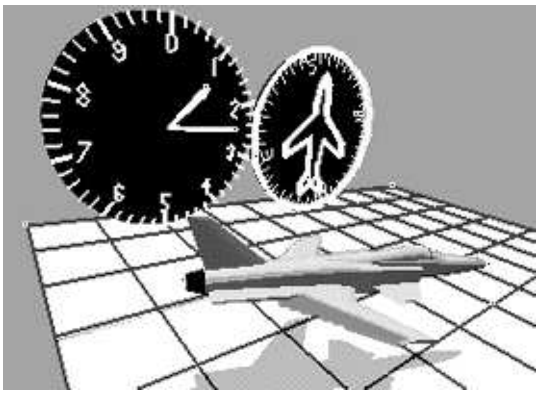

**Figure 8.10:** Gauges serve as both displays and controls of the plane's configuration. Although they are 2D objects, the gauges can be placed anywhere in the 3D scene.

the airplane gauges to be a useful interaction technique for controlling 3D objects.

### 8.3.6 3D Widgets

Traditional 2D widgets can be applied to 3D interfaces. To avoid the drawbacks of such an approach, [CSH+92] introduces the notion of 3D widgets as tools in the object's space.

The differential approach offers a mechanism for defining the behavior of 3D widgets, which is a central difficulty in their design[ZHR+93]. With our approach, we define the behavior of a 3D widget with a set of constraints that sufficiently restrict its behavior so that a through-the-lens control can be used. These constraints are applied while the widget is operated.

Figure 8.11 shows some example 3D widgets modelled after the ones presented in [CSH+92] and [SC92]. The "jack" widget on the left is used to translate an object along an axis. Pulling a tip of the jack causes the widget and its attached object to move in the direction shown by the arrow. When a handle is grabbed by pressing the mouse button, the object is constrained so it can only translate along the axis, and a through-the-lens control is applied to the handle. When the mouse is released, the constraints are removed. Similar techniques are used to create widgets for rotation about a point or axis, or to scale either uniformly or along an axis. Better graphic design can make the widgets more self-revealing.

Widgets to control the viewpoint, such as a virtual sphere, can also be created with the differential approach. The exact virtual sphere technique of [CMS88] is harder to create using the building blocks provided because the sphere that the user grabs is in screen space, rather than in object space. This has a number of disadvantages as it loses the kinesthetic coupling between dragging and the visible behavior. The arcball method of [Sho92] makes this problem worse by introducing a scaling factor to simplify the

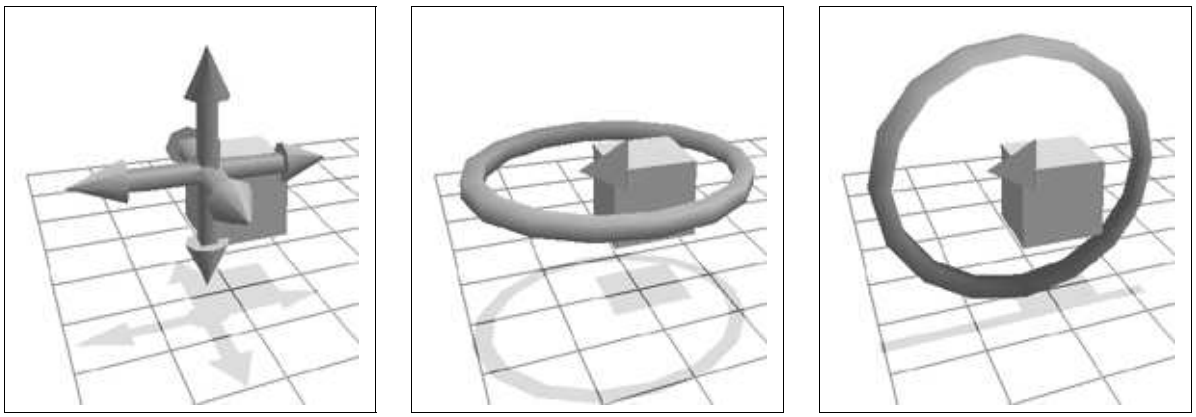

**Figure 8.11:** 3D Widgets for translating an object, rotating an object about a vertical axis, and rotating about a horizontal axis. The object is made semi-transparent to avoid obscuring the widget. Notice that since the widget is an actual scene object, it casts a shadow on the floor.

derivation of the mathematics.

A virtual sphere variant that we prefer is the virtual trackball. The virtual trackball can be thought of as a glass sphere surrounding an object in the world. To rotate an object, the sphere is grabbed with a through the lens control and is constrained so it can rotate only around its center. To make the manipulation of the object more "direct," the sphere is often omitted. Points on the object are grabbed through the lens, the object's center of rotation is nailed in place, and the object is made rigid so it revolves around the center. One addition to the virtual trackball is the addition of an elevator key that freezes the rotation but frees the uniform scaling of the object. This causes the virtual trackball to work like a spherical coordinate mousepole.

To use the virtual trackball to control the viewpoint, we consider the objects in the world to be encased in the sphere that is dragged. In practice, when the through the lens control is applied, the distance between the camera and the center of the world is constrained, as is the apparent position of the center on the screen. This virtual trackball behavior is part of the standard Bramble 3D interface of Section 7.8. Whenever a corner of the groundplane is selected, the trackball behavior is used. A dollying (sometimes incorrectly referred to as panning) behavior, created by allowing the camera to translate but not rotate, occurs when the center of the groundplane is dragged.

There are several advantages to developing a 3D widget with the differential approach and through-the-lens controls. First, it permits defining the widget's behavior without deriving any of the mathematics for converting from the input device's motion to parameter changes. Secondly, it defines the widget in a manner that is independent of the underlying representation of the object. Finally, the descriptions of the widget in terms of constraints provides a concise, executable specification of the widget's behavior.

### 8.3.7 Discussion

The differential approach lets us define and implement a variety of strategies for manipulating objects. The strategies are not mutually exclusive, for example we might attach a widget to a part of a constrained object (particularly if it is not sufficiently constrained) or use modifier keys to alter the behavior of a widget. As we experiment with different strategies, we find that they all have advantages and drawbacks. Here, we discuss some considerations.

**User provided vs. designer provided:** Making the user responsible for providing constraints may give the user more flexibility, but might also make them expend more effort as they specify behavior in addition to geometry. It also exposes them to the range of problems inherent in specifying behavior. Interface designers are (hopefully) better at devising good interactive behaviors, but cannot tailor designs to specific operations. If the objects to be manipulated will be used often, or replicated many times, it becomes worthwhile to spend effort in building behavior into them. This might be done by the user, the interface designer, or by someone else who is simply an object designer.

**Smart objects vs. smart tools:** Do we put lots of behavior into the objects, so that they can be manipulated with simpler tools, or do we develop better tools so that our objects need less behavior? An extreme case of the former would be an interface where every object had sufficiently well defined behavior so that any point could simply be grabbed with a though-the-lens control. An extreme case of the latter would be to have uniformly simple objects, for example ones that are being subject to the standard translate, rotate, scale transformations, and provide a set of widgets or commands which operate on them.

**Context-sensitive vs. context-free:** Is object behavior uniform, or do different objects (or even parts of objects) behave differently? Tailoring behaviors to objects has advantages, as discussed by [Hou92], but uniformity does too.

**Bounded scope vs. unlimited connection:** Permitting arbitrary relationships among objects can simplify manipulation by restricting unwanted behaviors, but when many objects interact, their coordinated behaviors become complicated as longer chains of causality are possible. Strategies which manipulate single (or a small number of) objects, such as widgets, have the advantage of keeping their simple behavior as the environment scales, but may grow tedious to use as the user must consider an increasing number of objects and interactions.

## 8.4 Employing Switching

In Section 6.4, controllers that operated by switching simpler controllers on and off were presented. In this section, we examine how these switching controller might be used.

### 8.4.1 Generalized Snapping

In Section 6.4.2, a controller for snapping to values was developed to aid in providing accurate direct manipulation. The `Snap` controller causes a connector to be driven to a particular value when it approaches that value. Because the controller can be attached to any connector, it provides a general method of snapping and its use is, therefore, referred to as generalized snapping.

Generalized snapping can be used to recreate the typical cursor snapping by representing the cursor as a differentially controlled object and creating `Snap` controllers that drive it towards desired snap targets.

Other behaviors can be created by placing `Snap` controllers on connectors other than those that are being directly manipulated. Because controller handling happens asynchronously in parallel with manipulation, precise relationships can be established away from where the cursor is. For example, if users often desire line segments to be accurately vertical, that is when something appears vertical it should be vertical, `Snap` controllers for the values $\pi/4$ and $3\pi/4$ can be placed on the angle connectors of each line segment. If a line segment is brought near being vertical as it is dragged, the configuration of the line segment would "snap" to a configuration where it was vertical. While this small jump may violate the continuous motion, it is usually small and predictable enough that it is permissible, especially when proper feedback is provided to express what snapping operation has occurred.

With generalized snapping, the snapping can occur away from the dragging action. For example, if the line segment of the preceding paragraph was attached to some linkage mechanism, the user might be controlling the line segment by manipulating some other part of the mechanism, as shown in Figure 8.12. This makes it even more crucial to provide proper feedback to the user to denote when something is snapped. Because the focus of attention is possibly away from the snap, for example at the dragging site, auditory cues might be useful as well.

The generalized snapping technique has many drawbacks that need to be resolved. First, it does not revert objects to their original configuration when unsnapped. Secondly, it needs to be connected to mechanisms that handle multiple snap targets. To do this properly might require the ability to determine whether two constraints conflict. Third, because the differential approach only drives connectors towards particular values, there is no guarantee that the snap controller accurately reaches its target. Finally, the snap controller has a number of potentially sensitive parameters that must be fine-

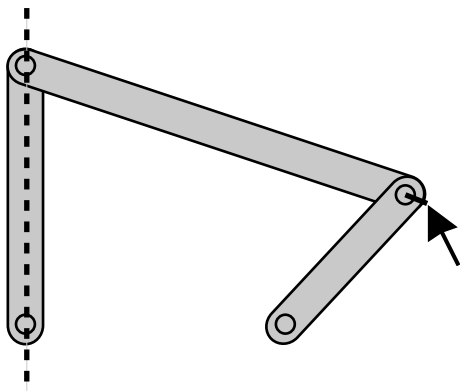

**Figure 8.12:** Generalized snapping can occur away from the dragging action. As the user drags the linkage mechanism, the left line segment gets snapped to the vertical position. Because such snapping can take place away from the focus of attention, it is important to provide proper feedback.

tuned to provide the correct "feel."

### 8.4.2 Collisions

Non-interpenetration is a useful constraint on objects. Such a constraint causes objects to collide and contact one another when they touch, rather than to simply pass through. A collision is the initial impact between two objects, which may remain in contact with one another afterwards.

With the differential approach, it is possible to create non-interpenetration constraints using inequalities. While the methods are not as sophisticated as the special purpose collision simulation methods reviewed in Section 2.2.3, they are simple to build with the abstractions of the differential approach, and are sufficiently effective to be interesting.

A particularly simple case of collision avoidance is used in box-and-arrow diagram creation, an application discussed in Section 9.3. The particular constraint we would like to enforce is that two axis-aligned rectangles do not overlap. This can be expressed as a disjunction: if at least one of the following inequalities hold, then the rectangles do not overlap:

$$\begin{aligned} x_1 + s_x &< x_2 - s_x \\ x_1 - s_x &> x_2 + s_x \\ y_1 + s_y &< y_2 - s_y \end{aligned} \tag{8.10}$$

$$y_1 - s_y \quad > \quad y_2 + s_y,$$

where $x$ and $y$ are the coordinates of the center of each rectangle, and $s$ is the size. Each clause represents one of the possible ways for the rectangles to be separated, either 1 is above 2, below 2, left of 2, or right of 2. Alternatively, the disjunction can be expressed as a maximum operation of the four terms.

The differential approach permits conjunctions of constraints to be handled easily, however disjunctions are more difficult. Enforcing the disjunction is not a problem when it is true, as each term can be checked until one that is true is found. However, when none of the expressions of the disjunction is true, a choice must be made. As with inequalities, when the constraint is violated it will be activated in order to "pull" it back to the admissible region. In the case of the disjunction, at least one of the expressions must be chosen and used as a constraint. Even though all of the inequalities are violated, all cannot be enabled because they would conflict.

The strategy for handling disjunctions of inequalities will be to select one of the inequalities to enable when the disjunction is violated. Because we are not backing up the instant where the disjunction began to fail (see Section 6.4.5), we must resort to enabling a violated inequality and permitting it to pull the configuration back to a legal configuration. The difficulty is selecting the inequality, which must be done by a heuristic. Although we would like to pull things out the same way they went in, the lack of history makes it impossible to do exactly.

A heuristic used for the simple rectangle overlap problem is to pick the inequality that is least violated. This technique is used for object non-overlap in the box and arrow diagram editor program of Section 9.3. The heuristic fails in two cases depicted in Figure 8.13. First, if the overlap is near a corner, an object may be pushed out to the side, rather than the way that it came in. Second, if the ODE solver step moves the object too far through the second object, it might be closer for the object to push out the opposite side that it came in from.

The basic rectangle non-overlap can be extended to the general case of rigid (e.g. rotatable) convex polygons. The basic element of a non-overlap constraint for two such objects is a point outside of polygon constraint. This is created by a disjunction, there must be at least one edge of the polygon that the point's distance inside the half-plane defined by the edge is positive. A point outside polygon constraint can be created with a similar heuristic as the rectangle non-overlap, with similar selection problems.

Two convex polygons do not overlap if and only if all the points on each are outside of the other. This makes it easy to break the polygon collision problem into point outside of polygon problems. What a polygon collider must do is find points that are inside the other polygon, called contact points, and enable constraints to push them outside. It will always be sufficient to choose at most two contact points, as this will establish a dividing edge between the polygons. After being selected, each of the contact points chooses an edge to be pulled out through. The heuristic must make sure that the points both exit through the same edge, even if they are on different objects.

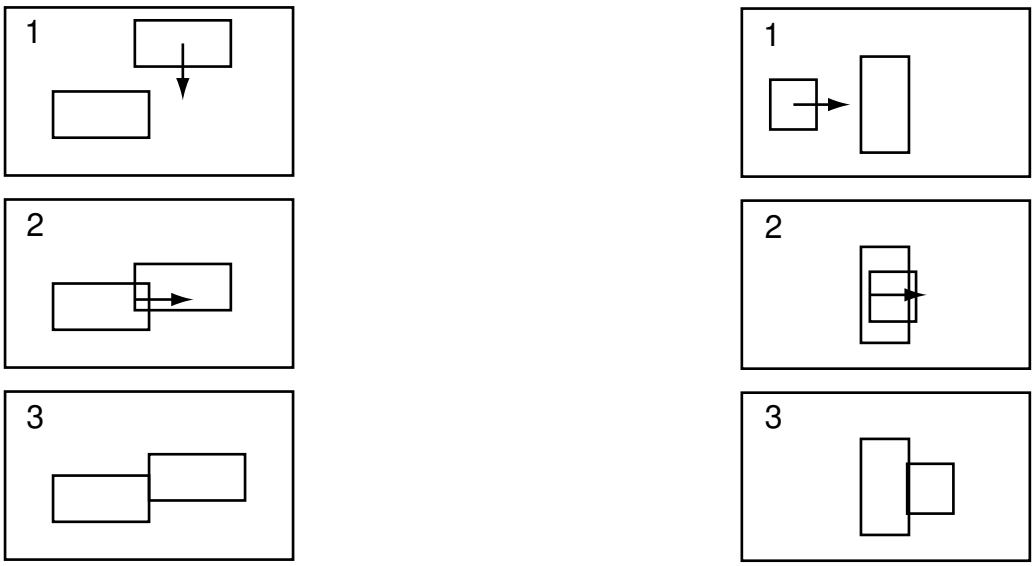


**Figure 8.13:** A disjunction of inequalities constrains two axis-aligned rectangles from overlapping. Because the selection of a direction to pull the block out is made without knowledge of the direction it came in, two types of errors may be caused. If the overlap is in a corner, the distance out the side may be less than the distance back out the top, as shown in the sequence on the left. Also, if the object moves too quickly it may pass through the other object, as shown in the sequence on the right.

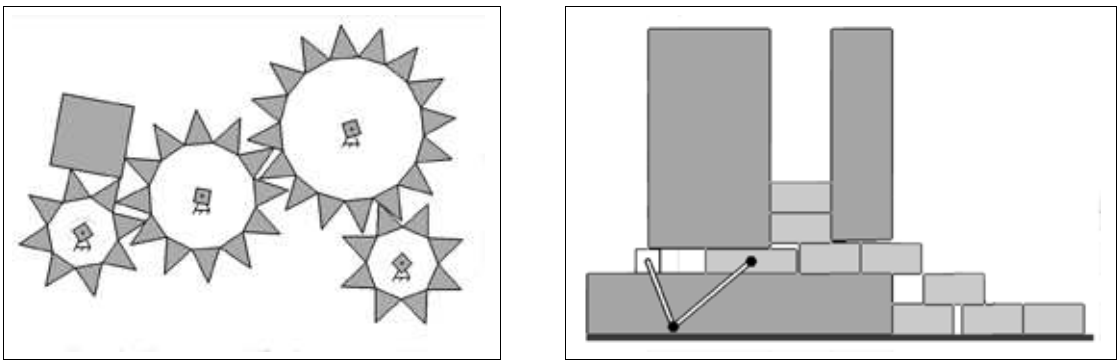

**Figure 8.14:** Mechanisms that can simulated using the simple collisions methods of the differential approach. The left shows a set of sawtooth "gears" jammed with a square block, and the right shows a block feeder.

While this simple method for collisions is not perfect, it is extremely simple to implement, and achieves good enough performance to allow some interesting mechanisms to be created with it. Two examples are shown in Figure 8.14. Since these methods were developed, techniques to permit robust simulation of collisions interactively have been developed by David Baraff [Bar94].