If you don't know where you're going, you will wind up somewhere else.

— Yogi Berra

Chapter 6

Controllers

The previous three chapters provided the machinery required to implement the differential approach. This machinery permits specifying desired derivatives for function outputs. Appropriate changes to the state of the objects are computed from these derivatives. In this chapter, we consider how to specify the derivatives in order to create graphical interaction techniques.

With the differential approach, graphical objects are manipulated by specifying how particular attributes of the objects should change. Graphical objects provide their attributes as output connectors. A connector serves as a control when its behavior is specified. The specification of the derivative of a connector is encapsulated into an object called a *controller.* These objects are plugged into any connector output just as dependency inputs are. A controller can look at the output it is connected to and the external world (e.g. input devices) in order to produce a desired derivative value for the connector.

The capabilities of a controller are limited: they can specify only a rate of change. Objects move by having controllers attached to their connectors over a period of time, continuously specifying derivative values which are converted by the differential optimization to derivatives of the object parameters.

The basic building blocks of graphical interaction techniques are controllers and connectors. For the purposes of interaction, the set of graphical objects appears as a set of connectors, any of which can serve as controls by having a controller attached to them. The chapter begins by showing how basic methods such as dragging are created by attaching controllers to connectors for periods of time. Subsequent sections explain how the continuous model of time is coupled with events and describes the details of the controllers themselves. Some details of creating complex controllers that switch controls on and off are discussed in the chapter's final section.

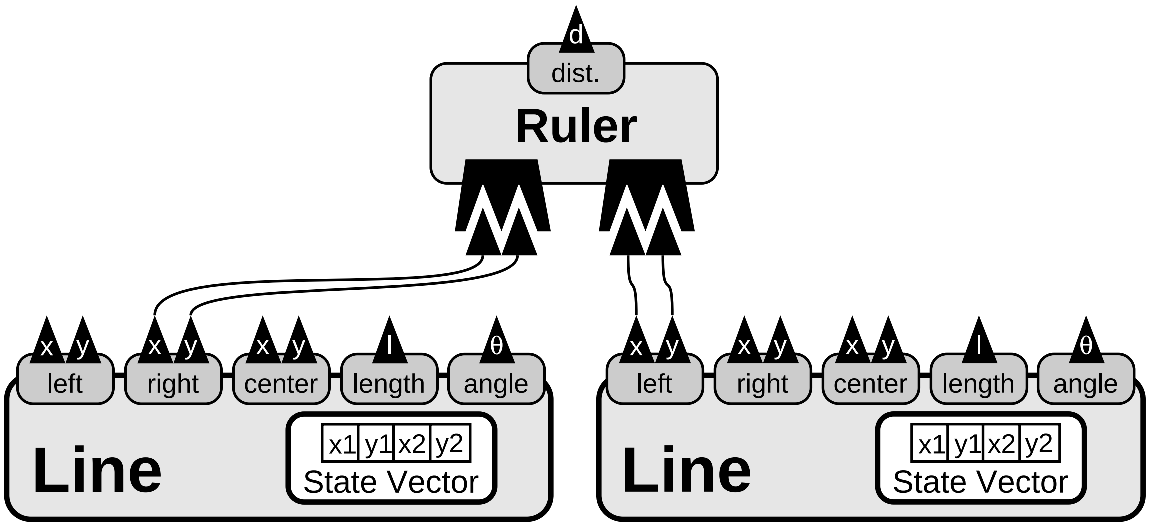

Figure 6.1: A schematic "wiring diagram" of two line segments with an attachment constraint between their endpoints. For the purposes of defining manipulations, these objects appear as a set of connectors awaiting controllers to be plugged into them.

6.1 Example Interactions

To show how the machinery of the differential approach applies to manipulating graphical objects, we consider some concrete examples. For these examples, our model will consist of 3 graphical objects: two line segments, and an attachment object that is connected to one endpoint of each line segment. A schematic of the data structures is given in Figure 6.1.

6.1.1 Specifying Values

A basic operation is to specify a value for a connector, for example to specify a position for an endpoint of a line segment. This cannot be done directly with the differential approach: only rates of change can be specified. In order to achieve a desired value, we must specify the derivative over a period of time, and wait for the object to achieve the desired value.

The `GoTowards` controller makes a connector's value move towards a target. At each instant, it computes a derivative that moves the value towards the desired goal, as will be discussed in more detail in Section 6.3. A `GoTowards` controller does not specify its attached connector's value. However, by pushing its value in the right direction, over time the connector will reach its target unless something impedes its progress. In order to specify a target value, we place a `GoTowards` and wait for the value to reach its target.

If the `GoTowards` controller is left attached after the attached connector reaches its target, the controller will continue to drive the value towards the target, holding it in place. This is how constraints are created. Because the `GoTowards` continually pushes toward the goal, the constraint will be restored if it drifts. The `GoTowards` controller, therefore, include constraint stabilization like that proposed by Baumgarte [Bau72].

To create a constraint, a controller must be applied to some connector. The attachment object which simply provides the connector is not a geometric constraint by itself. However, a geometric connection constraint object would be a version of the attachment object that created `GoTowards` 0 controllers on its connectors when it was placed, and removed these controllers when it was removed.

6.1.2 Dragging

To drag an object, we can permit the user to specify where a particular point on it should be positioned. When the mouse button is pressed to grab a point on an object, a differential controller is attached to the output that computes the position of the point. This controller guides the point to follow the mouse. When the button is released, the controller is removed from the output connector of the dragged point. The addition of an optimization objective term with the mouse position to provide manipulation of a graphical object is first presented in [KWT88].

The `Follow` controller provides derivative values that cause its attached controller to move towards tracking a moving target. It is similar to a `GoTowards`, with the exception that rather than having a fixed target, it has a moving target. This behavior could be created with a `GoTowards` controller that periodically has its target value updated. However, `Follow` controllers can sometimes provide better tracking behavior by using information about the motion of its target. In the case of tracking an input device, where derivative information is difficult to estimate for lack of a good predictive model, a `Follow` controller is typically implemented as a `GoTowards`. More details about the `Follow` controller are provided in Section 6.3.

For many reasons, it is unlikely that the point being dragged will track the cursor exactly. When the cursor moves quickly, it might take the system a few steps for the dragged object catch up, and even then, if the cursor moves far during the step, the two will separate. Also, the rates at which graphical objects can move is sometimes limited by how well the ODE solver can solve for the motion; if the object moves too quickly, the ODE solver will be unable to accurately compute its motion. Faster steps and better algorithms can reduce the separation, however the mouse might still separate from the object being dragged if the motion is restricted by a constraint.

When the cursor and dragged point diverge, it is important to provide visual feedback to connect them. Typically, a line is drawn between the cursor and the point being

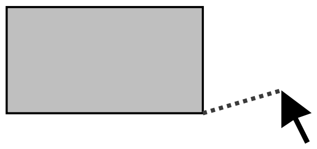

Figure 6.2: A small line is used to connect the mouse pointer to the object being dragged. This feedback is important because the cursor location will diverge from the location of the object because of lag, integration error, or restrictions on the object's motion.

dragged, as shown in Figure 6.2. Metaphorically the cursor is connected to the dragged point with a rubber band. This metaphor is a pretty accurate one for the spring attraction techniques introduced in Section 3.5.1.

The generality of the differential approach is apparent in the simple example of dragging. Graphical objects provide point position aspects. The dragging behavior can be plugged into any objects' output. The differential solving process will compute how to control the objects' parameters in order to achieve the desired motion. If a different pointing device, for example a 3D tracker, is available, it too could be plugged in to the same places.

6.1.3 Constrained Dragging

The ability to mix constraints and controls is important to the effective use of constraints. The constraints permit the user to drag objects without violating any previously established relationships. User defined persistent constraints can provide many services, including helping to avoid redundant work, helping to explore a constrained space, or helping to construct compound objects.

Because constraints are easy to attach and detach, they can be used with dragging controls to define interaction techniques. In addition to creating a `Follow` or `GoTowards` controller to cause the mouse to be tracked, additional constraints are simultaneously added. These constraints are removed when the dragging controller is removed.

For an example, consider rotating a graphical object. To create a rotation, the object could be dragged subject to a constraint that nails the position of its center in place. Dragging points on the object will then cause the object to rotate, assuming the object has that degree of freedom. Dragging might also cause the object to stretch, if the dragging motion is not circular about the center of rotation. This can be combatted by keeping the cursor on a circular track or by using another constraint to keep the size constant. In the latter case, some mechanism that makes the dragging constraint break rather than the others will be required. The dragging control must be subject to the other constraints. Techniques for achieving this are discussed in Section 3.5.

The ability to combine constraints and controls allows a set of building blocks to be

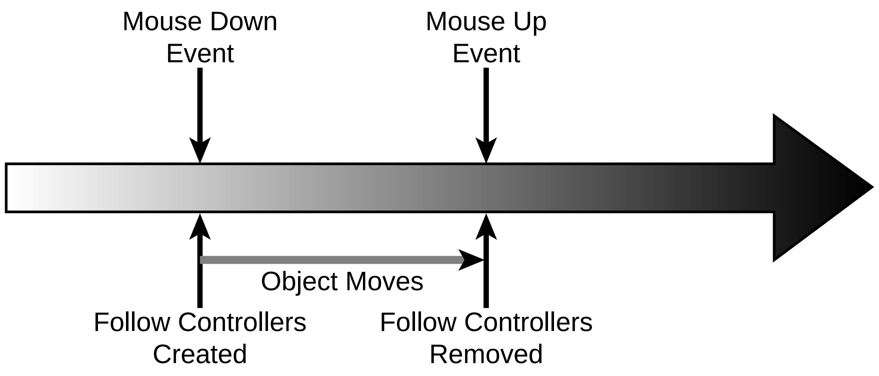

Figure 6.3: A timeline of an interaction in the differential approach. The arrow symbolizes the flow of time. At discrete instants, such as when the mouse events are received, the set of differential controllers is altered. The effects of these over time causes changes in object configurations.

provided for constructing interaction techniques. For example, it provides a constraint-based strategy for developing 3D manipulation techniques, as discussed in Chapter 8. The basic idea is that enough constraints are provided so that the input device is sufficient to fully determine the motion. This strategy is not unlike what we employ in the real world where we use things such as jigs and braces to help us manipulate hard to handle objects.

6.2 Continuous Time

As illustrated by the above examples, object configurations are altered by having controllers specify their behaviors over a period of time. The state of objects cannot be changed instantaneously. Instead, it changes over time, the way an object in the real world does. This is quite different than traditional methods for constructing interactive systems, where the state of objects is updated at discrete instants corresponding to events or polling increments.

In concept, time in the differential approach continuously moves forward. While time progresses, any active differential controllers are causing the objects they affect to change. At discrete instants, such as a window system event, controllers may be created, destroyed or altered, however, the state of objects is not changed. To change the configuration of an object, a controller must be created and time must pass so that the object can adjust itself accordingly. A time line of such a process is depicted in Figure 6.3.

Since a controller can be plugged into any connector at any time, placing a controller can be asynchronous with the placement of other controllers. This makes it easy to

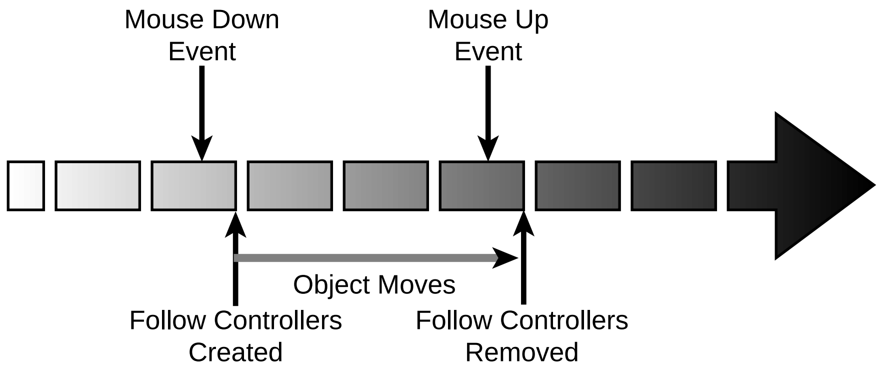

Figure 6.4: A timeline of interaction in the differential approach, as actually implemented. The arrow symbolizes the flow of time. Time is discretized into steps. A discrete event is deferred until the end of the step it occurs in.

perform concurrent operations, such as allowing for asynchronous two-handed input.

In practice, we can only approximate the continuous flow of time discretely. As discussed in Section 3.3, we must treat the ordinary differential equations describing object trajectories as a sequence of steps in order to solve them. During each step, time is advanced a small amount. By making these steps small enough, the user is given the illusion of continuous motion.

As implemented in the differential approach, events are deferred until the end of the step, as depicted in the timeline of Figure 6.4. If the duration of the steps is very small, the delays will not have an effect. If avoiding this lag is crucial, the approach can be modified slightly to trade total number of steps for reduced lag. For example, when an event occurs, the current step could be aborted and later restarted after the event is processed. Also, time may be stopped during periods when no objects are moving and user events are expected. For example, in a standard direct manipulation system, when the user is not dragging an object, nothing is moving so differential time may be stopped and the system may spend its full effort responding to events. An event may need to start time, for example if it initiates a dragging operation.

After each step, the system must redraw the displays. For the prototypes of this thesis, screen update always redraws the entire display. Other systems might attempt to speed redraw by selectively updating only objects that have changed. I have not taken such an approach because:

1. Selective redisplay is more difficult with the differential approach since many objects may be moving at once. In fact, the system must be able to determine when selective redraw is appropriate, and when it would simply be faster to redraw the entire view.

2. Selective redraw is difficult to implement for 3D applications when using a Z-

buffer hidden surface removal algorithm, because the state of the Z-buffer must be restored when objects are removed from the display.

3. The differential approach has been based on the assumption that fast drawing hardware is available.

6.2.1 Differential Time and Clock Time

The time in the differential approach is advanced at each ODE solver step. There is no assurance that the "clock" in the differential approach, that is the time that passes from solver steps, will correspond to wall-clock real time. In fact, making differential and real time correspond would be difficult: a system would have to be able to accurately predict how long (in real time) it will take to do the computations required for each step, and make sure the proper amount of differential time advances on each step. This rules out the possibility of the system adaptively controlling time steps when things grow difficult, and would require complicated synchronization when the real world clock and the differential time differ.

Rather than coupling differential time and real time, we instead let the clocks float. The differential "clock" can be thought of as a dimensionless quantity: it's mapping to "wall clock" time is unknown, and unimportant. Whether the clock moves quickly and velocities are low, or the clock moves slowly and velocities are high, the same effects can be achieved. It becomes impossible to express controls in terms of real time, for example to say that an object moves across the screen at 3 inches per second, or that a point reaches its target in 250 milliseconds. However, other imprecisions also make this impossible, both spatially (how far is 3 inches when the user can scale the window or run on a different monitor) and temporally (how can we be sure that in that 250 milliseconds the system will not have to swap or process a higher priority job). While multi-media systems researchers, such as [DNN$^+$93], are beginning to study with such real-time issues, the differential approach, like most interactive graphics systems, is not concerned with real-time performance. Coupling with real time is left for future work in Section 10.3, but a simple approach would take a step as fast as possible and then wait for the real time clock to catch up.

6.2.2 Breakdown of Interactivity

The differential approach relies on being able to provide a large number of steps per second. This is needed both to provide the illusion of continuous motion to the user, and to insure events are processed without too much lag since they are deferred to the end of a step. As the introduction to this thesis states, modern computers continue to provide increasing amounts of graphical and numerical processing capabilities. However, even though the capabilities of computers may continue to grow exponentially, the types of problems that users are interested in tackling may similarly grow without bound. As the task grows, the amount of computation required to support differential interaction also grows. At some point, processing each step requires too much time and the quality of interaction suffers. This is called *interactive breakdown.*

It is difficult to determine precisely the rate at which interactive breakdown occurs. Even when the motion appears jerky, it can sometimes be acceptable, depending on the user and the problem. For some applications, high frame rate is crucial, for example when the motion is needed to help the user comprehend the behavior of objects. Similarly, event delay lag is most bothersome when the user is generating many events in rapid succession or that are time critical, such as selecting a moving object. As described in Section 6.2, event processing lag can be reduced at the expense of throughput.

As the complexity of the problem goes up, so does the time required to do each step. The immense scale of the objects that people design – a modern jet airplane may have *millions* of components[Hei93] – makes keeping interactive rates impractical for some problems. However, the limitations of the human cognitive and perceptual abilities make it unlikely that a user would want to operate a model of that scale interactively. The prototype implementations show that models of reasonable complexity can be handled on the current generation of computers. Performance of the prototypes is discussed in Appendix B.

6.3 Basic Controllers

Controllers are objects which attach to connectors and specify values for their derivatives. The examples of Section 6.1 briefly introduced some types of controllers. Here, we examine these controllers in more detail, introduce a more complete set of controllers, and discuss how they compute the derivative values.

Since a controller's sole function is to provide a derivative value, deciding what values to specify is the critical issue in designing a controller. This is complicated by the fact that interface elements are traditionally not defined by derivatives, but rather by positions. Another issue in picking velocities is that simulation time is not coupled to real time in a meaningful way, as discussed in Section 6.2.1. This makes it impossible to specify velocities in terms of apparent velocities, although this would be attractive to insure that the user can follow the motion. However, the more relevant speed limit is typically given by the ODE solver, as discussed in Section 3.3.

The simplest controller provides a constant derivative value. These `Constant` controllers are rarely used because attributes typically do not move at fixed velocities regardless of other factors, and determining specific velocity constants is difficult because time and space do not precisely correspond to real-world quantities. In the cases where they are used, `Constant` controllers are typically given values which are found empirically to produce a desired rate in certain situations. The more common controllers do not inherently specify a time, but rather a target and provide the controller with flexibility in choosing the velocity that achieves it.

More useful controllers examine the value of the connector they are attached to in order to select a derivative value. The two basic varieties of such controllers are `GoTowards` and `Follow`. Each of these picks derivative values to make its attached connector achieve a target value. The difference is that a `GoTowards` has a fixed target, while `Follow` has a moving target such as a motion path or an input device. An important distinction is that a `Follow` may be able to use information about the motion of its target in order to track better.

A `GoTowards` controller picks its control velocity to get its value to the desired target. The value of the velocity must be related to the distance of the control from the target. There are several ways to choose the value. One is to pick a velocity that gets the control to its target in a set amount of time. The amount of time must be long enough to encompass a sufficient number of solver steps, and not too fast such that ODE solver inaccuracies cause the control to overshoot its target.

An alternate method of choosing the velocity for an `GoTowards` controller is to make it a scaled factor of the displacement. This creates spring-like attraction. When the control is far away it moves quickly, facilitating coarse positioning. As it gets close, it moves more slowly for precise positioning. One drawback of this technique is that it is hard to achieve target exactly: when the control is very close, its velocity is very low. One solution used in the prototypes is to switch between spring attraction and the constant rate scheme described above. When a control gets sufficiently close to its target, its velocity is chosen so that it would achieve the goal exactly in a step. This works well because for these small displacements, the approximations that the ODE solver makes may be sufficiently accurate.

An important attribute of `GoTowards` controllers is that they implicitly incorporate feedback. That is, they are continually adjusting the velocities to correct for any errors. If numerical drift or some other problem causes a control to make a turn for the worse, the error will be corrected in subsequent steps. Because of this, constraints will almost always use a `GoTowards` controller rather than a `Constant` controller with constant derivative 0.

An approximation to a `Follow` controller can be created with a `GoTowards` controller whose target is periodically updated. However, `Follow` controllers not only build in this continual update, but provide an opportunity to incorporate knowledge of the target's trajectory to improve tracking. A `Follow` controller effectively drives its value towards where its target will be, rather than where it is. In order to do this, the time derivative information of the target must be known. In practice, we rarely have good enough predictive models of input devices for this to work; however, it is useful for tracking key-framed motion paths, as described by [WW90].

6.4 Switching Controllers

The controllers we have discussed so far continuously perform the same simple task while plugged in. In this section we consider some more complex controllers that switch themselves on and off based on the values of the connector they are attached to and some measures of how hard they are working. In effect, these complex controllers simply plug and unplug more basic `GoTowards` controllers as needed. These behaviors are encapsulated as new controller types because: they are widely useful, so a simple encapsulation is handy; they can continuously watch the values, potentially switching more often than just at step boundaries; and they will look at internal solver quantities, the Lagrange multipliers, that we might prefer not to expose to system components outside of the solver.

By being clever about when controls are switched on and off based on the value of the connector they are attached to, a variety of interesting behaviors can be created. One behavior that this allows creating is an inequality constraint or boundary on a value. Switching of equality constraints is a standard technique for implementing inequality constraints, and is referred to as the *active set* method by the numerical analysis literature as it keeps a subset of all the constraints active at any given time. For the differential approach, we use similar techniques to create other behaviors as well. This section begins by describing three different variations of active set techniques, and then discuss some of the challenges in implementing them correctly.

6.4.1 Bounding

Placing boundaries or inequalities on aspect values is often useful, for example when a range of values is illegal, or only an approximate range is known. A `Bound` controller is created by enabling a `GoTowards` controller whenever the value has moved past the boundary. The `GoTowards` controller is used to move the value back within the boundary. When the output is within the legal region, the boundary constraint does nothing. The other major complication is that the boundary constraint should never pull the value further into the violated region, as depicted in Figure 6.5. That is, if another controller is trying to pull the value out of the illegal region, the boundary controller should not fight it. This case must be handled within the differential solver, as discussed in Section 6.4.4.

Inequality constraints operate by first being violated and using a `GoTowards` to fix themselves. If a value is moving towards a boundary, the `GoTowards` does not enable until after it has violated the boundary. One solution to this problem is that if such a violation is detected, back time up until the value is exactly at the boundary. This procedure is often done for physical simulation of collisions, as discussed in [Bar92b]. Such backing up could be used with the differential approach, and is discussed more fully in Section 6.4.5.

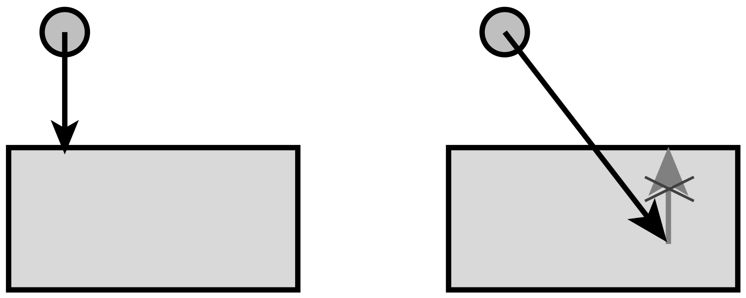

Figure 6.5: The point is bound to remain inside the rectangle. Left: when the point is in an illegal position a `GoTowards` controller pulls it back to the boundary of the legal region. Right: if some other controller will pull the point back into the legal region, a `GoTowards` controller that attempts to pull the point to the boundary of the legal region would pull against the other controller, and is therefore disabled.

6.4.2 Snapping

One of the fundamental issues in direct manipulation is precision: how can exact values be specified when they are required. This problem arises in many situations, for example when attempting to establish a precise geometric relationship in a drawing, when picking a small object, or when attempting to hold an object steadily in place with a noisy input device. One solution that has been employed to provide precision in direct manipulation is known gravity fields or snapping.

Gravity fields were first introduced in Sketchpad[Sut63]. Interesting points have a gravity field that draws the cursor in when it is close. The cursor follows the pointing device, however, when it is close to an interesting location it "snaps" precisely there. Many varieties of snapping have been used to help in direct manipulation. The most common are grids, which snap the cursor to points on a grid. Another important variety is object gravity, in which the cursor is drawn towards the graphical objects.

Just as we rely on tools such as straight-edges, rulers, and compasses to aid a pencil in drawing a precise drawing, gravity or snapping serves as a tool for creating precise graphical manipulations. Rather than having to manipulate objects to exact positions, the user can simply get them close, and the gravity takes the job of precisely positioning the cursor. This allows drawings to be created with more precision than the input device has, or to have precision even if the input device is noisy. Interface issues in snapping, such as providing feedback so the user can be confident that the correct relationship is being established and selecting among many close-together snap targets, can be handled, as shown in the *Briar* drawing program of Section 9.1.

One obvious way to incorporate snapping with differential manipulation is to use the snap target as the goal for dragging. This approach was used in the *Briar* drawing program, described in Section 9.1. One complication is that since the dragged object does not follow the cursor exactly, the object may not establish the relationship that the cursor does. This can be combatted with feedback to inform the user when the dragged point establishes the relationship that the cursor has, and by making the snapping persist long enough to allow the dragged point to catch up with the cursor.

It is also possible to use the differential approach to implement snapping. A controller is connected to drive the output to a precise value when the output gets close. If another controller attempts to pull the connected attribute from its target, the snapping controller is disabled. An analogy for this is pushing a marble around on a grooved table. When the marble is close to a groove it falls into the groove, and will roll around inside of it until pushed hard enough that it escapes from the slot.

A `Snap` controller is implemented using a variant of the active set technique used for `Bound` controllers. By default, the controller is inactive. If the value of its attached connector gets within a specified distance of the snapping target, the controller is activated and switches on a `GoTowards` controller to drive the connector to the target value. If the magnitude of the Lagrange multiplier for the `GoTowards` ever exceeds a limit value, the controller is pulling too hard against other controllers and must be deactivated.

The key intuition behind `Snap` controllers is that the Lagrange multiplier is a measure of how hard a control is pulling. Since the "pull" of a controller is actually determined by the product of the Lagrange multiplier and the gradient of the control function, the magnitude of this vector is actually used to determine the amount of effort the controller is applying. One difficulty with `Snap` controllers is that the parameters that determine their behavior, most specifically the limit magnitude, must be determined empirically.

A `Snap` controller can be attached to any object output, not just positions. For instance, a `Snap` controller on a line segment's orientation could make a segment that was easy to place into a precisely horizontal configuration. The differential implementation provides more flexibility than traditional methods for implementing snapping. We therefore call the differential snapping *generalized snapping.* In Section 8.4.1 we discuss some applications of generalized snapping.

6.4.3 Click Stops

Another way to combat the precision problem with direct manipulation is to require a value to have one of a discrete set of values, for example to create a grid for dragging. This is easy to do with the traditional implementations of direct manipulation, which often use integer representations anyway. However, the differential approach operates on continuous values. One way to implement an interaction that limits a value to a discrete set is *clicking.*

Clicking is a variant of snapping that effectively constrains a connector output to have a discrete value, either from a finite set, or to be an integer or multiple. `Click`

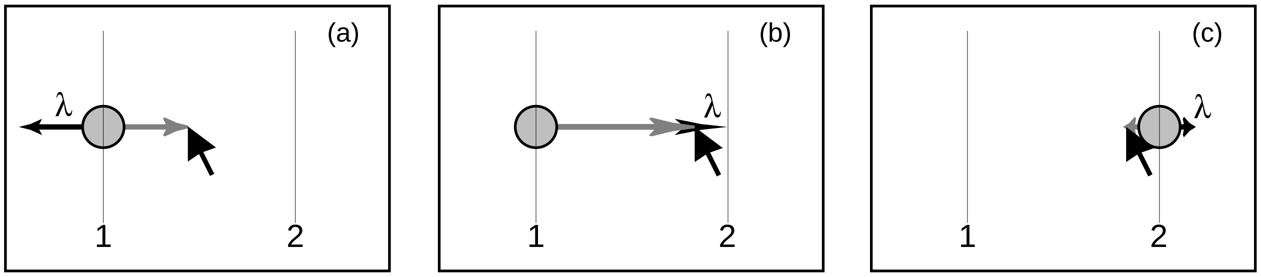

Figure 6.6: Clicking forces a value to be in a discrete set. The value, in this case the horizontal position of the circle, is constrained to have the value 1 or 2. Grey arrows represent the pull of the mouse control, black arrows represent the pull of the `Click` controller. Initially (a) it has value 1, so a `GoTowards` controller pulls to maintain this value as the mouse attempts to drag it with a pull labeled λ. At some point, the mouse pulls hard enough so that the pull vector's magnitude exceeds a limit value causing the controller to advance to the next click stop (b), causing the point to attain the next value in the set (c).

controllers work by constraining the output to have a value in the set. If the constraint is pulling too hard, the value is changed to another element in the set, as illustrated in Figure 6.6.

6.4.4 Implementing Active Set Methods

The `Bound`, `Snap`, and `Click` controllers all operate by switching simpler `GoTowards` controllers on and off. They are all variants of the active set methods commonly used for realizing inequality constraints. Active set algorithms are detailed in standard textbooks such as [GMW81] and [Fle87]. The difficult part of such algorithms is determining which constraints to enable and disable, a combinatorial problem. The simple method discussed here tries a few guesses at the correct set, and then gives up, selecting a solution that fails to satisfy all the requirements. The solver will be likely to provide an acceptable interactive behavior. We will, therefore, begin by examining the active set problem in the context of the differential approach, and then described a simple algorithm. While the simple technique does guarantee the correct answers to the problems, it has the following advantages:

- The methods always fail in ways that can be corrected later.
- The methods are easy to implement as extensions to the existing differential optimization.
- The methods do not access any of the matrices, except by calling the differential optimization. Therefore, as with the differential optimization techniques, we can use any linear system solver.

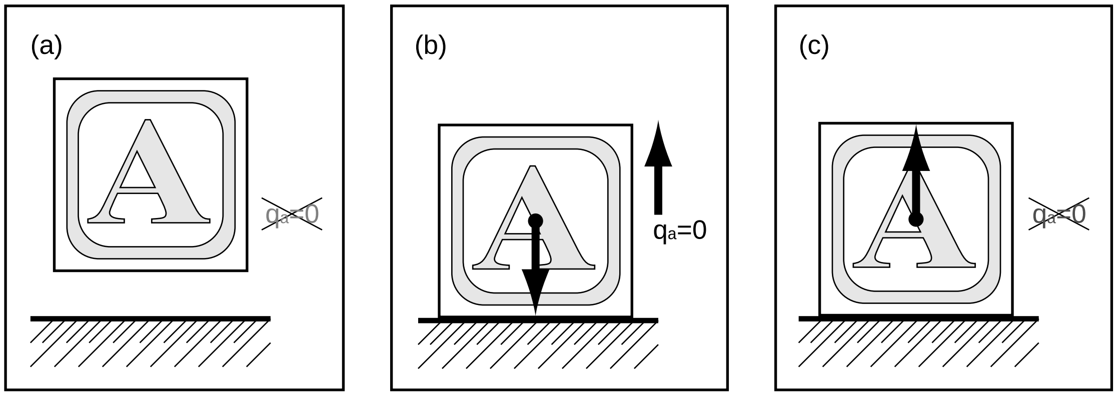

Figure 6.7: An inequality constraint $q_a > 0$ keeps the block above the floor. The inequality is implemented by switching an equality constraint $q_a = 0$ on and off. When the block is above the floor (a), the equality constraint is inactive. If the block is on or under the floor (b), the constraint pushes the block towards being on the floor, counteracting any controls that are pulling the block downwards. However, if other controls attempts to pull the block above the floor (c), the control used for the inequality would pull downward, causing sticking. Therefore, it is deactivated.

Recently, an inequality solver specifically designed for interactive systems was developed by David Baraff [Bar94]. The algorithm was designed for implementing physical simulations of collisions, a problem very similar to implementing graphical manipulation with the differential approach. Baraff's algorithm could be used in computing the differential optimization. Application of this algorithm to implementing the differential approach is left as a topic for future study (Section 10.3). However, without significant work, such algorithms do not have many of the advantages of the iterative solvers discussed in Section 4.3.2, such as the ability to exploit sparsity.

Active Set Methods in the Differential Approach

To introduce the basic idea of an active set method, consider the simple example of Figure 6.7. The example has a single state variable, q_a that measures the block's height above the floor. An inequality constraint is used to keep the block above the floor, $q_a > 0$. An active set method implements this inequality by switching the equality constraint $q_a = 0$, or in differential terms q_a `GoTowards` 0, on and off as needed.

If the block is above the floor, the inequality constraint does nothing. It is *inactive*. The more interesting case is when the block either sits on the floor, or has fallen below the floor. In this case, the `GoTowards` controller is activated to either push the block towards sitting exactly on the floor, or to prevent the block from sinking underneath the floor. When the constraint works to keep the block above the floor, its Lagrange multiplier will be positive, that is, the constraint is pushing upwards.

Consider a case where the block is sitting on the floor, with the constraint activated. Suppose another control, such as connection to the mouse, attempts to pull the block off the floor. The `GoTowards` controller will attempt to keep the block on the floor, with a negative Lagrange multiplier pulling downwards. In essence, the inequality constraint will be sticky. The solution to this is to disallow negative Lagrange multipliers. When one is found, the constraint must be deactivated.

When there are multiple inequality constraints coupling objects, deciding which Lagrange multipliers to deactivate becomes more complicated. Consider the example of Figure 6.8. Two keep the two blocks stacked, one inequality constraint keeps block B above the ground, while the other keeps block A above block B. The former constraint is implemented by switching the equality $q_b = 0$ on and off, the latter by switching $q_b - q_a = h$. For each solution of the Lagrange multipliers, the solver must determine which of the two constraints should be activated. If a control is used to pull block A upwards and both constraints are active, the equality constraints would maintain the stacked configuration, causing the blocks to stick in place by pulling downward on them. Since the inqualities can only push upwards, we might deactivate the ones pulling downwards (e.g. both of them). However, deactivating all of the constraints pulling in illegal directions is unnecessary, and wrong. If there is a soft control pulling B downwards, that constraint should stay active.

The process of determining the active set is iterative: the solver must determine which ways the constraints are pulling, deactivate any constraints pulling in illegal directions, reactivate any constraints that should not have been disabled, and repeat. Determining the correct active set can be difficult, and might require many attempts.

If the correct active set is not found, there are two types of errors possible for any particular constraint. Either a constraint that should be active is deactivated, or a constraint that should be inactive is activated. This latter case is extremely problematic. Consider what would happen in the example of Figure 6.7: the user would not be able to lift the block off the floor. If the algorithm made this error in one step, it would make the same error in subsequent steps since the configuration does not change[1].

Deactivating too many constraints is less of a concern. Consider the example of Figure 6.8. If both constraints are deactivated, the lower block will not be prevented from moving downward for the step. However, in the next step, the algorithm could correct for this error. It is unlikely to make an error activating the constraint because the block will be separated from the upper block, which is what caused the confusion.

If we resign ourselves to not always being able to find the correct active set, we must at least make sure to always err on the side of deactivating too many constraints. Such an algorithm is simple to devise. An example algorithm is presented in the next

[1]Assuming the algorithm is deterministic and it bases decisions solely on the configuration, which the differential methods do.

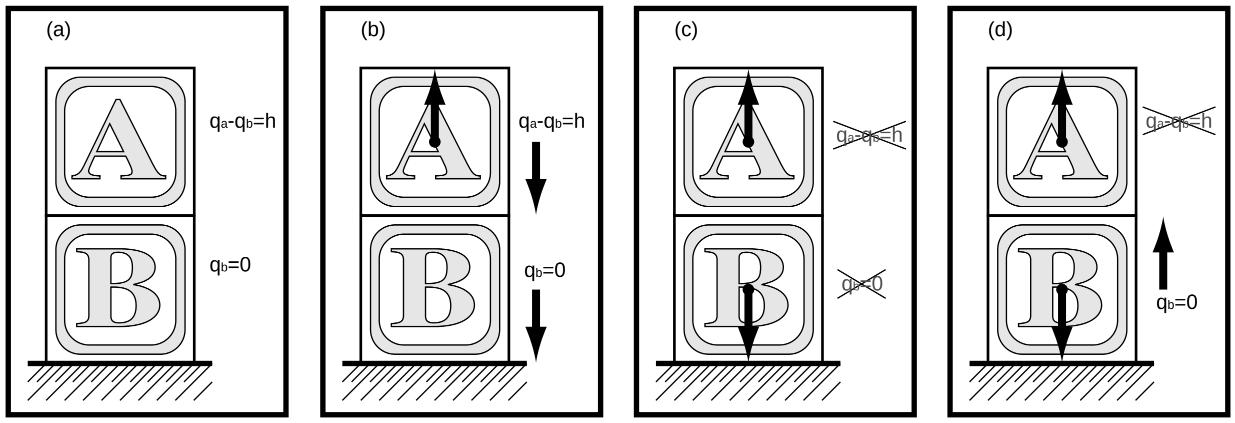

Figure 6.8: Multiple blocks are kept stacked on the floor by inequality constraints. (a) One constraint keeps block B above the floor, the other keeps block A above block B. (b) If a control pulls upwards on block A, the active constraints impose equalities that cause the blocks to stick in place. (c) Removing all inequalities that are pulling downward in (b) would cause the constraint on block B to be erroneously removed. If another control were pulling down on it, no constraint would keep B above the table. (d) The correct solution reenables constraint B.

section.

It is important to notice that excess deactivations are only a problem when there are many interacting inequality constraints. There are interesting applications, such as collision simulation, where this occurs. In such applications, these errors might be problematic. However, there are other potentially more significant sources of errors in such applications, such as improperly handling the piecewise continuous nature of the ODEs, as discussed later in this section. In Section 8.4.2, we will look at using the differential approach for collisions, and consider the problems.

An Active Set Algorithm

The basic outline of a simple active set technique is as follows:

1. Select an initial active set by activated any inequality constraints that may be active. Candidate constraints are those whose values either violate their boundaries or are within a tolerance of the boundaries.

2. Select some deactivated constraints that are potentially active and reactivate them.

3. Solve for the Lagrange multipliers.

4. Deactivate any constraints with negative Lagrange multipliers.

5. If the active set was changed, return to step 2.

The simplest method omits step 2. This leads to a method that meets the requirement that it will always err on the side of having too few active constraints. It also is guaranteed to terminate because on each iteration the size of the active set must decrease, since there is no step inside the iteration that adds constraints. It has the disadvantage that it often removes too many constraints.

A constraint is a candidate for reactivation in step 2 if the derivative of the function is negative. A constraint $f_i(\mathbf{q}) > k$ is a candidate for reactivation if $\nabla f_i \cdot \dot{\mathbf{q}} < 0$. Implementing reactivation can be complicated. The method must not to cycle, that is to infinitely loop among several active sets. A method that I have used to prevent looping is to permit a given constraint to be reactivated at most once. While this falls far short of a reliable inequality solver, it gets correct solutions on many problems on which the simpler method fails, for example the problem of Figure 6.8.

6.4.5 Piecewise Continuous ODE Solving

Solving the ODE is difficult because the derivatives are continually changing. In Section 3.3, we were concerned about them changing due to the functions being non-linear. However, the derivatives might also change discontinuously when the set of controls change. Such ODEs are called *piecewise continuous* as they consist of continuous pieces between breaks. Note that the discontinuities are in the derivatives, not in the values. Methods for solving piecewise continuous ODEs are discussed in an appendix of [Bar92c].

An ODE solving technique such as Runge-Kutta attempts to fit a continuous function (a polynomial) to the ODE over a step. If the ODE is discontinuous, this may be problematic. Even if multiple smaller steps are taken to fit the function with piecewise polynomial segments, methods that take fixed step sizes are not likely to have their discontinuities match the discontinuities in the derivatives.

Many of the discontinuities are avoided in the differential approach, since events that change the set of controls are deferred until the ends of steps. The ODEs are therefore step-piecewise continuous, or continuous within the steps. This means that techniques such as Runge-Kutta are acceptable, because they will not see the discontinuities. Techniques that require continuity between steps, such as Predictor Corrector methods, may have more problems.

Active set methods may change the set of controls during a step, causing a discontinuity in the derivatives. This is potentially problematic. However, the alternative would involve finding the precise time that the controls switch and adapting the step sizes so that the step boundaries occur exactly at this instant.

The most severe symptom of not finding discontinuities is that the system can respond to changes only after they happen. For example, an inequality constraint activates after it is violated. Therefore, there will be a brief instant when the constraint is violated. Subsequent steps will move to a it valid state. The ill-effects of an ODE solver misfitting a continuous polynomial will similarly be repaired by subsequent steps. If such error is unacceptable, cleanup steps can be used.

Piecewise ODE solving by backing up can be used with the differential approach. I have instead used the approach of letting later steps correct for any errors caused by violations. There are many reasons to prefer such an approach:

- The general root finding problem of determining when to back up to is difficult.
- The ill-effects of adaptive step sizes, as described in Section 3.3.1 are applicable - possibly even more so as adjusting the step size requires a potentially time consuming root finding operation.
- When there are many changes close together in time, a backing-up system can take only tiny steps between them. In a cleanup afterwards approach, many can be cleaned up simultaneously.
- With the cleanup approach, objects not affected by the discontinuity continue to move as they otherwise would. If the step size were adapted, all of the objects would slow down whenever any discontinuities occurred.

For certain applications, such as collision simulation, having instants of violated inequalities will be unacceptable, and piecewise ODE methods will be required.