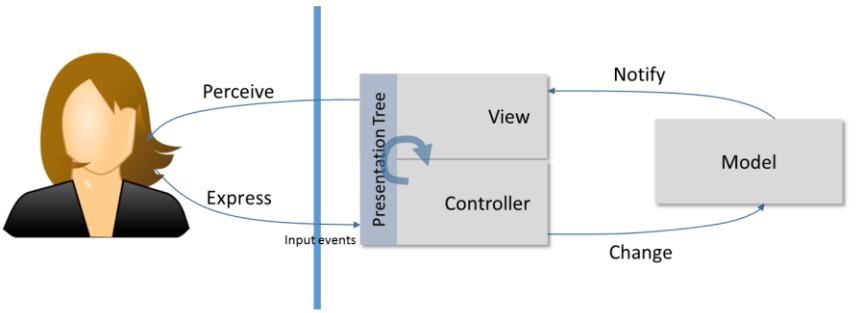
Input Handling

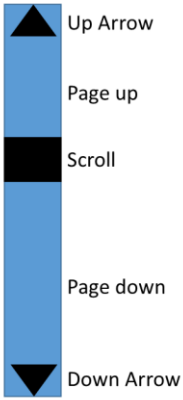
Figure 8-1 shows our basic Model-View-Controller architecture. The user perceives what is on the screen and in response to her own desires and what she perceives, she expresses herself to our interactive application. In modern user interface architectures, these expressions occur as input events. Input events are received by the controller and then, with information from the presentation tree (if there is one) and the view, the controller translates sequences of input events into changes to the model. When the model is changed, all views that are listening to that model are notified (as in chapter 7). The views update themselves (as in chapter 6) and inform the windowing system using damage(). The windowing system will call paint() on any damaged regions to update the screen. The user then perceives the changes and the process repeats. In an interactive program, the main program merely sets up the user interface. The real control of flow resides in the head of the user. This lack of direct, programmatic control is disconcerting for some programmers.



8-1 – Model-View-Controller

In this chapter we are going to focus on the role of the controller. When an input event, such as a mouse click is received by the controller, it really does not have any meaning by itself. What matters is what part of the presentation was clicked. That is where the meaning is established

by the user. The controller will consult the presentation tree, if there is one, and the view. It will pass in the location of the event and receive back the *essential geometry*. The essential geometry is basically "what was clicked?" For example the scroll-bar in figure 8-2 has 5 regions. The action on the model depends on which region contains the mouse location. Those regions are the essential geometry of the scroll-bar. Once the controller knows the region that contains the mouse location, the controller problem is quite easy.

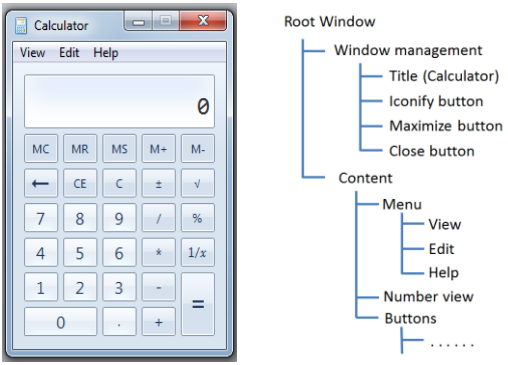


8-2 – Essential Geometry of a Scroll-bar

In the case of our scroll-bar the essential geometry is based on rectangles. Testing whether a location is inside of a rectangle consists of 4 if statements. Such simple geometry is widely used but not always adequate. In chapter xxx we will look at more complex geometry so that we can solve a greater range of interactive techniques. For the purpose of this chapter, rectangle geometry is sufficient.

As shown in figure 8-3 most graphical applications are constructed around a window tree. Each window is a rectangular region that is self-contained. Generally the Graphics object is configured to clip away any drawing that is attempted outside of the window's rectangular boundaries. We will call each such self-contained window a Widget (as discussed in chapter 7). A widget encapsulates a view and a controller

and in simple cases, such as the scroll bar of figure 8-2, the model is also included in the widget.



8-3 Calculator with window tree

In this chapter we will first review the kinds of input events that we might receive. We then look at the event dispatch algorithm that takes an input event from the operating system and decides which widget should receive that event. Once a widget has been selected to receive the event, we must invoke some of the application program's code so that event can be handled. There are a variety of mechanisms for binding events to code.

In many systems the event-code binding is the limit of the architecture. It is up to the application programmer to write whatever event handling code they desire. However, in some systems there are default controllers that perform interactive operations directly on the presentation tree. In such architectures the input events are hidden from the programmer. Instead what are received are change descriptors on the presentation trees. We will discuss how to upgrade our model-to-view mapping techniques so that they can also perform view-to-model mappings in a consistent fashion. Lastly we will discuss how widgets can themselves generate events that are consumed by other, higher-level widgets to perform more complex behaviors.

Input events

There are four classes of input events that are of interest to us when building our user interface. The mouse events are the most common and are obviously generated when we move the mouse and press its buttons. Touch events come from touch screens, like those found on tablets and smart phones. If there is one touch it is generally treated like a mouse event, although there are differences. Touch gets more complicated when sensing multiple fingers. Keyboard events are special because they do not have any spatial location. Lastly there are widget events that are generated by widgets in response to patterns of simpler events.

Mouse events

Mouse events are characterized by the X,Y location of the mouse and the pattern of buttons on the mouse that are currently pressed. The most common events are MouseDown (when a mouse button is pressed), MouseUp (when a mouse button is released) and MouseMove (when the mouse changes location). A mouse event is generally accompanied by an event object that contains the X,Y location as well as flags indicating which buttons on the mouse are pressed at the time of the event. In addition, flags for modifier keys such as shift, control, alt or command keys are also included. In some applications, the meaning of the event is altered by whether one of these buttons is pressed. Some systems offer a MouseClick event which is MouseDown followed rapidly by MouseUp with few MouseMove events in between. This behavior is so very common that it is included by many systems even though it can be built from the three major mouse events.

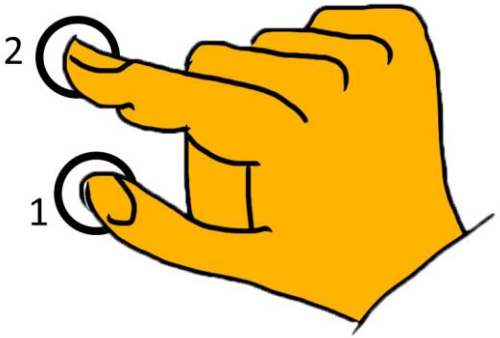
In addition to the three major events, many widgets also receive the MouseEnter and MouseExit events. These occur when the mouse enters the widget's window and when the mouse leaves. This allows for highlighting and responsive behaviors that are independent of the user actually offering input for changing the model.

Touch events

The first major difference between touch events and mouse events is that with most touch screens, there is no hover. With a mouse, the MouseMove event is reported whether any buttons are pressed or not. With touch events, we generally only know the location when the finger is actually pressing on the screen. This makes button hover or other responses impossible because no events are being received. The Samsung phones and tablets do sense proximity to the screen and can use that for hovering, but it has not been widely adopted by application developers.

In general, an application that responds to the three major mouse events can work effectively without considering touch. The touch events are simply mapped to the same methods as the mouse events and the application continues to work. The biggest difference is in the use of multi-touch. Smartphone techniques such as pinch/spread to control zooming require the sensing of two fingers at least. This cannot be handled as simulated mouse events.

Many touch event systems talk about Pointers where a pointer is any mouse, finger, stylus or other input modality that includes a screen location. As with mouse events there are the PointerPressed, PointerMoved and PointerReleased events. The difference is that the event object contains an array of pointers (X,Y positions) rather than a single location.



8-4 – Multitouch example

In figure 8-4 we see two fingers being used for a gesture. The event stream that we might see is in figure 8-5.

```
PointerPressed (thumb down, single pointer, ID=1)
PointerMoved (thumb wiggles a little, single pointer)
PointerPressed (finger down, two pointers for thumb and
      finger, finger ID=2)
PointerMoved (thumb and finger moves with the gesture,
      two pointers are returned each time)
PointerReleased (thumb[ID=1] comes up, single pointer in
      array, finger ID=2)
PointerReleased (finger[ID=2] comes up, no pointers)
```

8-5 – Two finger gesture event trace

Our first PointerPressed event comes when the thumb goes down. Our code has no idea it is the thumb, only that a touch was initiated. The pointer array contains only one location (the thumb) and the thumb was assigned a unique ID (in this case a 1). Android assigns IDs to pointers, other systems might not. The purpose of the ID is to track particular pointers as they appear and disappear. When the second PointerPressed event occurs, there are now two pointers in the array and the finger has been assigned ID=2. Our gesture may have selected a center for rotation on the first pointer down. We want to track that pointer to make sure we keep the center of rotation on that point. After the movements of the gesture, the thumb comes up first. There is now

only one pointer in the array (at index 0) but that pointer still retains the ID=2.

Many systems add events for pointers entering and exiting a widget. Android also provides two variants for PointerPressed and PointerReleased. In Android there is ACTION_DOWN when the first pointer goes down and ACTION_POINTER_DOWN when any subsequent pointers go down. This simplifies knowing when a gesture starts. Android also has ACTION_POINTER_UP for when any pointer other than the last one goes up, and ACTION_UP when the last pointer goes up and the gesture is done.

Various systems have subtle differences in their handling of multitouch but they all follow the basic ideas presented here. This allows us to write applications where multiple touches, pointers, stylus or mice are possible.

Keyboard events

As we will see in the next section, keyboard events are special because they do not carry any location with them. We will visit this problem again in the section on event dispatching. In most cases our application responds to a simple KeyEvent. Accompanying this event is a character. Generally modifiers such as shift or control have already been considered and the KeyEvent produces the appropriate character. If a user enters a control-A, we would generally not key a KeyEvent for the control key. We would only get a KeyEvent on the “A” and the character we would receive would be a control-A because of the processing that occurred in the operating system. Most modern systems will return a Unicode character so that all issues of language and keyboard style have already been resolved before the input event is passed to the application.

It is also possible to directly receive the KeyPress and KeyRelease events. These are the primitive events from the keyboard. This allows applications to have commands based on multiple keys being pressed at the same time. This is sometimes used in gaming but rarely in other

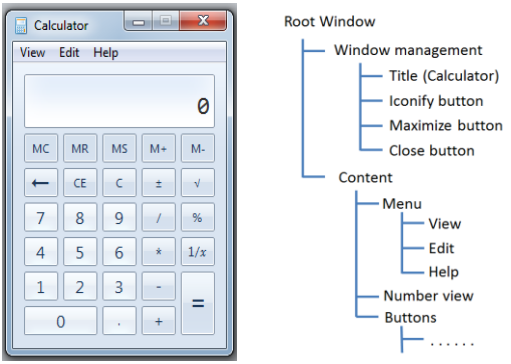
applications. These events are accompanied by an indicator of the key being pressed. In many systems the actual array position of the key on the keyboard is returned, allowing the application to remap the keyboard's meaning in some unique way. For the remainder of this text we will consider only the KeyEvent.

Widget events

One of the ways that widgets can be integrated into our user interface architecture is if they themselves generate events. For example a scrollbar might generate an event every time its value changes. There are a variety of primitive input events that can cause the scrollbar to change, but the rest of the application only cares about when the value of the scroll bar changes. Three scroll bars might be collected together for the red, green, and blue values of a color picker. Their scroll events might be received by the color picker which then generates "color changed" events. Again at the higher level, we do not care about the individual scroll events or about the mouse events. We only care about the change to the color.

Event dispatching

Event dispatching is the process of deciding which of the widgets in a window tree should receive a given event. There are three basic mechanisms for doing this: top-down, bottom-up and focused. Figure 8-6 shows our familiar calculator window tree.



8-6 Calculator with window tree

In a top-down strategy the event is passed to the Root Window and it does whatever it wants with the event. This is simple to implement but if a MouseClick event had occurred over the number 7, the root window is probably not the right place to handle it.

In a bottom-up dispatch algorithm the front-most window that is lowest in the tree receives the event. In essence the algorithm is as shown in figure 8-7. By considering widgets in front to back order, the front-most widget is always found first. By considering the children before handling the event itself, a bottom-up strategy is achieved.

```
boolean Widget.dispatchEvent(Event E)
{
    foreach ( child C in front to back order)
    {   if (C.bounds.contains(E.mousePoint))
        {   if (C.dispatchEvent(E))
                return true;
            else
                break;
        }
    }
    if (this widget can handle the event )
    {   handle the event
        return true;
    }
    else return false;
}
```

8-7 – bottom-up event dispatching

The code in figure 8-7 merely sends the event to the lowest appropriate widget. In some cases, however, the lowest widget may not want the event. If a widget does not want the event it returns false and widgets farther up the tree can handle it. This is where the “up” part of bottom-up comes in.

Since the advent of object-oriented programming we can make the code in figure 8-7 the default implementation of dispatchEvent(). Any subclass can override the method if they want. The result is a mostly bottom-up dispatch algorithm that can be customized for special purposes.

Event focus

KeyEvents have no mouse location to use in deciding where the event should be dispatched. For this situation we have the concept of *key focus*. The key focus is simply a global variable in the root window that points at the widget that should receive the KeyEvents. A widget can call a method to request the key focus. For example the calculator’s value type-in window would automatically request key focus so that it would receive all of the typed characters. When filling in a form like that in figure 8-8, each text box would request key focus whenever the mouse selects that widget.

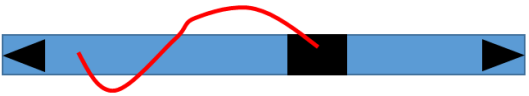
8-8 – Form with text-box widgets

In figure 8-8 we might select the first name text box and have it request the key focus. However, when we are through with the first name we would enter tab or return and want the key focus to automatically move to the next text box. Various widget systems define a *tab order*. When a widget releases the focus (for example when it receives a tab key), the key focus is given to the next widget in the tab order. There are various ways that tab order is defined in systems. They are all quite simple to figure out.

Also, when a text box loses focus it wants to remove the insertion indicator and other visual indications that it is active. For this the special event for LostFocus is defined. Whenever the focus is taken away and given to another widget, this event is sent to the widget that previously had the focus. Just to show that programmers have a warped sense of humor, in many systems the LostFocus event is called Blur.

There are also cases where we need to focus the mouse rather than dispatching mouse events to the widget indicated by the mouse location. For example, in figure 8-9 we show the path of the mouse while trying to scroll to the left. Because the scroll-bar is skinny it is easy

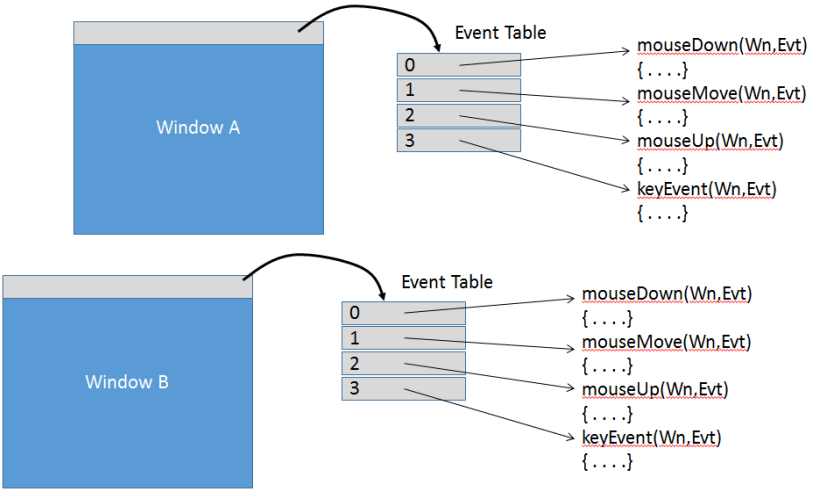
to drift outside of the widget's boundaries. The solution is to request the mouse focus when the MouseDown event first occurs in the slider. We release the mouse focus on MouseUp. Holding the mouse focus means that all mouse events will be sent to the widget regardless of which widget's bounds contains the mouse location. In many systems a widget automatically receives mouse focus on MouseDown and it is automatically released on MouseUp. Such dragging operations are by far the most common case and this automatic mouse focus mechanism relieves the programmer from worrying about the focus. Another example requiring mouse focus would be digitally painting near the edge of a window. We want to protect the user from the consequences of accidentally slipping outside of the window when working at the edge.



8-9 – Mouse focus

Event-code binding

The various techniques for binding an event to a particular piece of code follow a similar structure to that described for listeners in chapter 7. The big difference comes in the various types of events that must be handled. In our model change listening, the key piece of information was that the model changed. With interactive inputs there are the mouse events, touch events and the others. When an input event occurs we need three pieces of information: the code to call, the display object effected by the event, and application data associated with that object. In many systems we rely upon the display object to contain a reference to the application data.



8-10 – Event tables

Event tables

One of the earliest techniques was event tables, in GIGO[xx]. Every input event was given an integer constant and every window had an array of function pointers, as in figure 8-10. When an event was received, the event dispatch algorithm would identify the appropriate window and then the integer event code was used as a subscript into the event table to extract a function address. This function was then called with the window and event record as its parameters. An application data pointer was stored with the window. In Microsoft Windows this mechanism is even more primitive. Instead of a table of event functions for each type of event, each window has a pointer to a function that handles all events for that window. It is up to the "WindowProc" to decide what to do with each event.

Event tables have the advantage that they are very easy to implement and every efficient. The downside is that because we are dealing with raw function addresses, there is a large opportunity for mistakes that can be difficult to debug. There is also a problem with interface design environments. Function addresses are not known until run-time which

makes it impossible for the design tool to specify the function to be called.

Object-oriented

The Smalltalk environment[xx] pioneered the use of object-oriented programming for input event handling. In this style there is a superclass or an interface that defines a method for each event. Figure 8-11 shows such a Widget class.

```
class Widget
{   void mouseDown(Window wn, Event evt)
    { . . . . . }
    void mouseMove(Window wn, Event evt)
    { . . . . . }
    void mouseUp(Window wn, Event evt)
    { . . . . . }
    void keyEvent(Window wn, Event evt)
    { . . . . . }
    void paint(Graphics g)
    { . . . . . }
}
```

8-11 – Object-oriented event handling

In this approach, every window has a pointer to an object that is a subclass of Widget. When a particular event is dispatched to a particular window, the event dispatch code grabs the Widget object from the window and calls the appropriate method. Inside of object-oriented languages such as C++, Java or C# there is actually a table of procedure addresses called the *virtual table*. In essence the object-oriented style repeats the event-table approach with the compiler managing the tables and ensuring their correctness.

Listeners

The object-oriented approach fails in two ways. The first is when Microsoft Windows defined thousands of different events for all parts of the operating system and pushed them through the same event-handler pipe. In this case the virtual table was thousands of bytes for every new class, most of which were not used for any given widget. The listener architecture where events were separated into meaningful groups or individual functions is far more effective than using one

mechanism for all possible events. The second problem was that forcing everything to be a subclass of some single class cause code architecture issues. Listeners resolve this problem as well.

For a better understanding of listeners, please review the listener section in chapter 7. This describes the various forms of listener including delegates, function closures and parsed code strings.

What is common in many systems is to assign functions to particular attributes of an object. For example, we can assign a function to the "onClick" attribute of a line. Whenever the user clicks on that line, the function is called. This is very common in web browsers. Each input event corresponds to a different attribute name. This is also simple, but only allows for a single target for the event.

Presentation tree changes

An alternative approach to input handling is to make the objects in the presentation tree interactive. The simplest form is selection. For example, we can give Ellipse a "selected" attribute that accepts a function value. When select is sent, the Ellipse object itself knows to handle input events and test them for selection of the ellipse. If the ellipse receives a mouseUp() event that is geometrically inside the ellipse, the Ellipse class will detect the relationship and call the function stored in the "selected" attribute. A Group could also have a "selected" attribute. Whenever any of the contents of the Group are selected, the Group's "selected" function will be called. This allows selection of larger objects than simple drawing primitives. The advantage of this architecture is that all of the geometry for selection is embedded in each of the drawing primitives. The application programmer does not need to consider geometry at all. The algorithms and math for such selections are discussed in chapter 9.

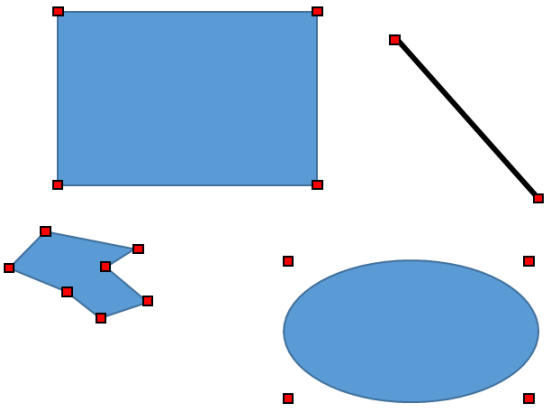
In our example we used a "selected" attribute that contains a function. Any of our listener mechanisms could be used to accomplish this. We could also define a selected() method on every drawable object and

allow the programmer to subclass those methods to add selection functionality to any particular part of the display tree.

In another variation of this, any object that does not have a “selected” attribute but is interactively selected by the user can propagate its selectedness up the presentation tree to its parent and so forth. It can also add a path to the event as it propagates up the tree. This allows the actual processing of the selection to happen at any level of the tree. The path provides access back to the object that was originally selected by the user, if that is helpful.

Control point interaction

Selection is an interesting place to start but there are other forms of interaction that could be embedded into the presentation tree objects. Figure 8-12 shows a set of drawable objects that have control points indicated on them. Many interactions with geometry are expressed by the dragging of control points. When the mouse goes down over a control point, the dragging process begins and as the mouse moves the control point is moved until the mouse is released.

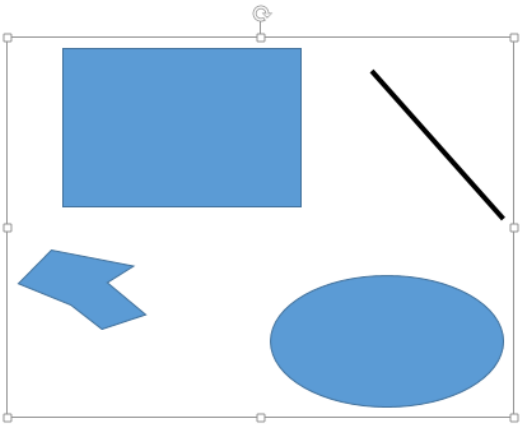


8-12 – Control points

If we want our graphical objects to provide such functionality they need only provide two methods: getControls() and changeControl(index,x,y). When an object is first selected one can call getControls() to have the

object return an array of points, one for each control point. For example, the line would return its two end points. The rectangle would return its four corners. Other software could display the control points and process the selection and dragging of the control points. Every time a control point is moved to a new location it would call changeControl() on the original object and tell it which control point should be moved and its new location. Adding the additional method move(dx,dy) to objects would allow them to be dragged around the screen.

The concept of dragging objects and control points can be extended to groups. Figure 8-13 shows a group of drawable objects that is surrounded by 9 control points. The control points on the corners and sides control scaling of all objects in the group and the control point at the top controls rotation around the center.



8-13 – Group controls

So we can put these all together to form a basic interaction mechanism for a presentation tree. So for every drawable object in our presentation tree, we expect the following methods:

Path selected(x,y) if the point (x,y) will select this object then return this object's path from its parent.

Controls getControls() returns an array of control points that can manipulate this object.

void changeControl(index, x, y) will change the control point indicated by index to the new x,y value.

void move(dx, dy) will move the object from its current position by dx,dy

Let us also assume that Group's selected() method will call selected on its children in front to back order. If any of the children are selected, the Group will add its own index onto the front of the path. This will build up a path of indices that leads back to the originally selected item.

Based on this capability in all of our drawable objects in our presentation tree we can create a subclass of Group called Interactor. What Interactor does is accept input events as they propagate down the tree and use the selected() method on its children to see if any of the children have been selected. If one has been selected, Interactor uses the returned path to find that object and to ask for its control points. The Interactor can then draw those control points after the contents have been drawn. This puts the control points in front of everything else.

The Interactor can now use the mouse events it receives to select and drag control points. It changes selected objects using their changeControl() method. As presentation objects are changed, they cause the screen to be updated and redrawn appropriately. We might alter the behavior of Interactor by putting a "controls:true" attribute on some presentation objects. Interactor would only display control points and allow interaction with objects whose controls attribute was true. Interactor could also look for restrictions such as "controlMode:"vertical"". If Interactor found this it would only allow changes in Y for any movement.

The nice thing about this architecture is that each drawable object manages its own geometry, but the Interactor object manages the full

interaction. Interactor can define for as many different attributes of its own on objects as it wants, because all objects can have an arbitrary number of attributes, even if they do not know what most of them mean.

Presentation mapping

The above discussion showed how we can write objects like Interactor that can provide rich interaction with a variety of presentation objects. The problem is that such interaction does not change the model. We need a mechanism for propagating presentation changes back to model changes. For this we revisit the mapping functions of chapter 6. One of our examples from chapter 6 was a linear map governed by the equation:

$$\frac{Pv - Pmin}{Pmax - Pmin} = \frac{Mv - Mmin}{Mmax - Mmin}$$

This time we want to solve for Mv because we know the presentation information from our user's interactions, but we want the new model value. The resulting equation is:

$$Mv = (Pv - Pmin) * \frac{Mmax - Mmin}{Pmax - Pmin} + Mmin$$

We can now augment our Linear class from figure 6-25 to the code in figure 8-14.

```
public class Linear extends Map
{   SV presentPath;
    SV modelPath;
    double modelMin;
    double modelMax;
    double presentMin;
    double presentMax;
    public void fromModel(SV presentation, SV model)
    {   SV v = getPath(model,modelPath);
        double mv = v.getDouble();
        double r = (presentMax-presentMin)/
            (modelMax-modelMin);
        pv = (mv-modelMin)*r + presentMin;
        setPath(presentation, presentPath, pv);
    }
    public void fromView(SV model, SV presentation)
    {   SV v = getPath(presentation, presentPath);
        double pv = v.getDouble();
        double r = (modelMax-modelMin) /
            (presentMax-presentMin);
        mv = (pv-presentMin)*r + modelMin;
        setPath(model,modelPath,mv);
    }
}
```

8-14 – Linear Map

We have added the fromView() method that will use the linear equation to map a presentation value to a model value. Now when Interactor or whatever other class we define allows a user to change the presentation, the Map subclasses can also convert back to the model. A single Map object defines the transformation in both directions. As discussed in chapter 6, there are many other Map objects that are possible than just Linear.

Text

Our control points discussion concerned itself only with geometry. We can also enable Text presentation objects to support editing. Instead of control points the selection would be a position within the string. Any object receiving keyEvents could change the selected Text object based upon its cursor location.

Widget Encapsulation

As with other software systems, user interface software also needs the ability to break down the design into manageable parts. We also have a need to develop reusable pieces that do not require new programming in each application. Reusable widget libraries not only simplify programming but also unify the look and feel of the user experience. If all buttons look and act the same, it is easier for the user to understand the behavior of the interface. The way to achieve such uniform behavior is to reuse widget implementations from a standard library.

So far in this chapter we have discussed the basic mouse, touch and keyboard events. For purposes of encapsulation we need widgets to generate their own events. Take for example the scroll-bar in figure 8-15.

8-15 – Scroll-bar widget

To implement this scroll-bar we need to process the various mouse events that will click and drag the components of this scroll-bar. We also need to consider all of the pieces of its geometry and presentation. However, for programmers using the scroll-bar, they do not want to see all of that.

To use the scroll-bar we want to consider 4 things: 1) the scroll-bar's model , 2)the geometric placement and sizing of the scroll-bar, 3) the styling (color, texture, shape), and 4) events generated by the scroll-bar. For an implementation we can create a class of object for our presentation tree called HScrollBar. HScrollBar is drawable and can handle input events. For its model, it has three attributes: max, min and value. The value is the current value of the scroll-bar and the primary part of its model. For styling information such as color, texture, arrow shape, etc. we can add attributes to HScrollBar. In The HScrollBar implementation, any changes to these attributes will cause the scroll-

bar's presentation tree to be changed or rebuilt to reflect the attribute specification. For other styling issues we will defer to a later chapter on styling.

The widget needs to generate its own events. There are several ways that we can handle these events. The HScrollBar can consume all of the mouse events that are sent to it and then whenever the value attribute is changed it can send a notify() message to its parent, as in chapter 7. This would propagate up the presentation tree and eventually be handled by one of the higher nodes in the tree. The notify() message could be modified to accumulate the path to the widget that generated the event as well as the name of the event. This would simplify the parent's process of identifying and correctly using the event. Mapping objects such as Linear (chapter 7) could be attached to the widget to map changes in value back to changes in the model. This allows programmers to ignore all of the geometry inside of the scroll-bar and focus only on its current value.

Another alternative is for HScroll to generate its own events. This might be a simple "valueChanged" event that is generated whenever the value changes or it might be separate events for "stepUp", "stepDown", "pageUp", "pageDown" and "scroll". This is really a widget design choice. There is also the design question of whether "valueChanged" should be called on every event that changes the value or only on mouseUp. Some implementations provide two different events, one for continuous value changes and one for changes only at the end. The HScroll object itself would use one of the delegate, listener, or function models to register listeners for the event(s). All mouse events would be handled internal to HScroll with users of HScroll listening for the higher-level events of scrolling.

The separation of widget specification into model, geometry, styling and events can be applied to any number of interactive objects such as buttons, radio buttons, check boxes, text boxes, dials, knobs, sliders and others. The encapsulation makes them all reusable. However, for a particular widget toolkit it is recommended strongly that all widgets in

the toolkit use similar geometry, styling and event handling techniques to simplify the programmer's job in using them.

Summary

For interactive behavior to occur, our software must accept and process user input events. In most interactive situations, the location of where an event occurred is critical to understanding what an event means. For this we introduced the concept of *essential geometry* which translates an event location into a meaningful action. We discussed four major types of events: mouse events, touch events, keyboard events and widget events. Each of these has their own challenges in use. We also discussed the use of a window tree for deciding how location-based events (mouse and touch) can be dispatched. We discussed the two major algorithms of top-down and bottom-up event dispatch. We also discussed the need for an implementation of key and mouse focus in event dispatching.

The binding of events to actual code to process those events has a variety of options. Primitive event tables, were translated into object-oriented event handling. Listeners, delegates, function closures and parsable language strings were all presented as mechanisms for binding events to code.

We then discussed how interaction can be attached directly to drawable objects themselves. This provided a variety of interactive opportunities with little programmer effort. The general notion of control point interaction was discussed, as well as constraints on such interactions. The use of mapping objects was extended to allow presentation changes to map back to model changes.

Lastly we used the concept of widgets to encapsulate interaction so that the widget behavior as a whole could be used instead of dealing with the internal events and geometry. Widget specification was broken down into model, geometry, styling and event handling.

Exercises