Extensible Input Handling in the subArctic Toolkit

Scott E. Hudson, Jennifer Mankoff
HCI Institute, Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213, USA
{scott.hudson, jmankoff}@cs.cmu.edu

Ian Smith
Intel Research, Seattle
1100 NE 45th St, Seattle, WA, 98105, USA
ian.e.smith@intel.com

ABSTRACT
The subArctic user interface toolkit has extensibility as one of its central goals. It seeks not only to supply a powerful library of reusable interactive objects, but also make it easy to create new, unusual, and highly customized interactions tailored to the needs of particular interfaces or task domains. A central part of this extensibility is the input model used by the toolkit. The subArctic input model provides standard reusable components, which implement many typical input handling patterns for the programmer, allows inputs to be handled in very flexible ways, and allows the details of how inputs are handled to be modified to meet custom needs. This paper will consider the structure and operation of the subArctic input handling mechanism. It will demonstrate the flexibility of the system through a series of examples illustrating techniques that it enables – many of which would be very difficult to implement in most toolkits.

Author Keywords
GUI Toolkits, event handling, interaction techniques.

ACM Classification Keywords
D.2.2 Design tools and Techniques: User Interfaces; D.2.11 Software Architectures; H.5.2 User Interfaces

INTRODUCTION
Over the past 20 years, GUI toolkits have emerged as a significant success story in the area of tools for interface implementation. They have provided the highly reusable infrastructure which most of today's interfaces are built with, and provide a foundation for higher-level tools which allow high quality interfaces to be created rapidly enough to enable iterative development. However, most toolkits have been built around the notion that they provide a relatively fixed library of *interactors* (also called *components*, *controls*, or *widgets*), and nearly all interfaces will be constructed with this library. So while it is very easy to reuse library interactors, and modify them with simple sets of parameters, it is often quite difficult to create new interactors beyond the library. While this can have some advantages in terms of consistency, in the longer term, this has had a stifling effect on innovation in interfaces – we see comparatively few new interaction techniques being widely adopted in part because moving to them often involves significantly more implementation effort than older techniques.

Submitted to CHI '05

A central goal of the subArctic user interface toolkit has been to extend the role of the toolkit from simply a library of widgets, to an enabler for extension into new interactive forms. It does this in part by seeking to make it not only very easy to use it's existing library, but also quite easy to create new interactors, and to support this creation even when the new interactors operate in unusual ways. A central part of achieving this goal has been the extension capabilities of its input model.

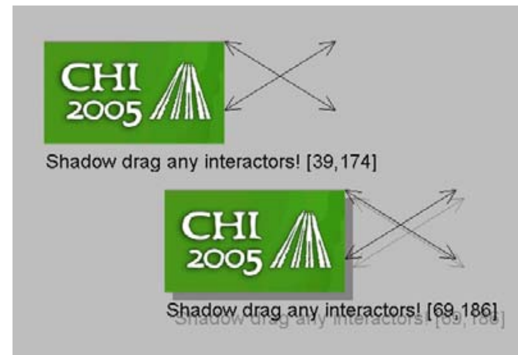

Figure 1. Shadow-drag container interaction

As a simple illustration of one capability enabled by the toolkit input model, Figure 1 shows the use of the `shadow_drag_container` interactor class. This object adds new input and feedback capabilities to any set of interactors placed inside of it. In particular, it allows that set of objects to be dragged by the user, providing a drop shadow effect which gives the appearance of picking up and then setting down the objects. Further it does this for *any* interactor (including those created well before this container was conceived of, and those which change dynamically or are animated) without requiring any modification to the interactor itself. Further as we will see in later sections, because of subArctic's extensibility support mechanisms, this general capability is remarkably easy to implement.

In the next section, we describe the key features of subArctic's input architecture, including dispatch policies, dispatch agents, picking, and the interfaces implemented by interactors wishing to receive specific types of input. We then discuss a range of examples that illustrate some of the capabilities enabled by the architecture. We conclude with lessons learned over the years we have used this input system.

THE SUBARCTIC INPUT SYSTEM

This section describes the key components of subArctic's input handling system. We begin with a brief overview of event handling. We then describe *input policies,* which make high level decisions about where different kinds of events should be routed, and *dispatch agents*, which perform input translation and embody input protocols for handling groups of related interactions. We then describe the picking infrastructure used by the system, and finally show how all these parts fit together.

Overview

At the most general level, the process of input handling is one of delivering inputs to appropriate objects in the *interactor tree*, the collection of interactors which implements the interface. Like almost all modern interactive systems, subArctic uses an event model of input. Under this model, inputs are represented by a series of *event records* (or simply *events*) each of which records the relevant facts and context associated with some significant action of interest to the system (*e.g.*, the manipulation of an input device by the user). Each event is delivered by the toolkit to one or more interactors within the interactor tree (which as a whole implements the interface and represents its current interactive state). To process (or *handle*) an event of interest to it, each interactor needs to interpret the event in terms of what the interactor is (*e.g.*, a button versus a slider), and its current state (*e.g.*, for a checkbox, what part of the interaction this is, such as whether we have seen an initial press, *etc*., and whether the checkbox is currently checked or unchecked). In doing this, each interactor can be seen as translating a low-level event expressed in terms of actions on a device (*e.g*., the left locator button has been released) into something with a higher level meaning (*e.g.*, that a particular menu item has been selected, and a certain application action should be invoked on behalf of the user).

Both the maintenance of state and some parts of the translation to higher level terms performed by the interactor are conveniently handled by code which (implicitly or explicitly) implements a finite state controller [17,19]. For example, the finite state machine illustrated in Figure 2 is convenient for processing a drag interaction which consists of a locator button press event, followed by zero or more locator move events, and then a locator button release event. At each transition in this state machine, an interactor would typically perform actions. For example on the press transition, it might change its appearance to indicate it had been "picked up", on the move transition it might change its screen position, and on the release transition, it might return to its normal appearance and invoke some action.

Overall, this conceptual structure for handling input – modeling input as records of significant actions (events), delivering these events to appropriate interactor objects, using these events to drive changes in interactive state, and translating low level events into more meaningful concepts – is used by essentially all modern toolkits. The subArctic input system (and its predecessor the Artkit toolkit [6,11] which shares some of its constructs) fits within this conceptual structure. It differs from most other systems, however, in the extent and power of its specialized infrastructure supporting this overall process.

To support its goal of making it easy to create new interactor classes, the subArctic system attempts to move as much of the work of handling input as possible out of interactor objects, and "up" into the reusable toolkit infrastructure, while still allowing those capabilities to be modified. In particular, it provides a well developed infrastructure for determining which object(s) will receive different kinds of events in flexible ways, for tracking of interactive state for many common interactions (with the equivalent of simple finite state controllers implemented inside the toolkit infrastructure), and for translation of events in common higher level interactive concepts (*e.g.*, translating a press-move-drag sequence of events into "dragging").

Event Dispatch Policies

A central part of the subArctic input system is its infrastructure for deciding where events will be delivered, a process we call *event dispatch*. There are two primary ways that common interfaces determine where an event goes. Many events are dispatched *positionally* – that is they are sent to interactors found beneath the cursor on the display. This is typically appropriate for locator button presses, for example. Other events are typically *focused* on a particular object independent of where the cursor is. This is typically appropriate for keyboard input, which should be delivered to the current text focus object independent of where the cursor goes.

Positional and focus-based event dispatch represent two policies for delivering events. Most prior toolkits have a fixed and immutable set of such policies built into them – in many cases driven strictly by a fixed set of event types, with keyboard events being delivered in a focus-based fashion and most other events delivered positionally (*e.g.*, in a bottom up fashion in the interactor tree starting with the lowest interactor that overlaps the cursor position). One of the key insights in the design of the precursor Artkit input system was that flexibility in selecting input policies was useful. As a simple example, consider locator movement events. As a part of a painting interaction, or to invoke a rollover highlight, these should clearly be delivered in a positional fashion. On the other hand as a part of a dragging operation, if these events were delivered

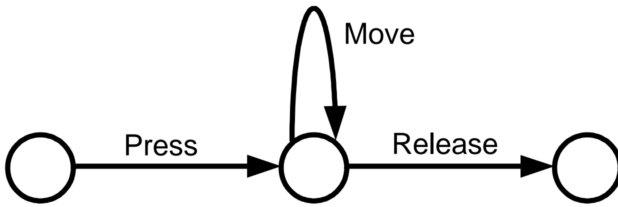

Figure 2. Finite state controller for a drag sequence

positionally and the cursor moved quickly, it might go outside the object being dragged and the remaining inputs would be misdirected. (Issues of this type lead to the introduction of the dubious feature of input "grabbing" for example in the X window system [4].)

A second insight was that, although most current interaction makes use of positional or focus-based event dispatch, other policies are also useful. For example, traditional modal dialog boxes could be easily implemented using special positional and focus policies that deliver events normally to the interactor subtree representing the dialog box, but filter out input intended for the main interface. As another example, when implementing a full screen crosshair cursor object, it is necessary to deliver locator move events to the cursor object (so it can update itself to match the mouse location), but not consume or otherwise disturb them, because they must still get to the actual interface. This leads to special *monitor-focus* and *monitor-positional* policies, which provide (and enforce) this behavior.

Once the notion of a policy for making input dispatch decisions is made explicit, it becomes clear that there could be many such policies. To provide for this, the input system implements an extensible set of *dispatch policy objects*. Like other parts of the system, the active set of policy objects can be changed dynamically (*e.g.*, on a per interface basis, or even over the life of a single interface) and can be changed independently (*e.g.*, a developer could add an event monitor to an interface they did not implement) simply by installing a new dispatch policy object.

Dispatch Agents

Each dispatch policy object attempts to deliver input using one of its *dispatch agents*. Each agent handles a certain type of input, such as text editing input, dragging of several forms, or pressing, clicking, or double-clicking. In order to make the implementation of new interactor objects easier, dispatch agent objects are responsible for implementing a number of the first level state maintenance and translation activities which in other systems are done in the interactors themselves. For example, a simple drag agent would implement the finite state controller shown in Figure 2. It would listen for press, move, and release events and use these to track its state. It would also communicate with interactors wishing to receive dragging input.

Dispatch agents communicate with interactor objects using a series of method calls making up an *input protocol*. Each agent implements one or more such protocols representing a particular type of input, and each interactor wishing to receive that type of input declares that it implements the Java `interface` which defines that protocol (the Java compiler then insures that the corresponding methods are in fact implemented by the object). For our simple dragging example, the protocol might be defined as:

```
public interface simple_draggable
    extends focusable {
       public boolean drag_start(…);
       public boolean drag_feedback(…);
       public boolean drag_end(…);
};
```

To support dragging input, a dispatch agent would call the `drag_start()` method on the first transition in the state machine shown in Figure 2, `drag_feedback()` on each middle transition, and `drag_end()` on the final transition. The parameters passed to each of these methods would include information useful to the interactor making use of this input, such as the absolute and relative position from the start of the drag. (In fact, in the real system a number of different drag agents are provided which compute and deliver parameters associated with moving, resizing, and other drag specialized tasks.)

Overall each input protocol represents a common input pattern (such as dragging or text editing) that one would expect to be used in multiple places, and each dispatch agent provides a reusable implementation of that pattern.

Arranging to get input delivered to a particular interactor happens in different way for different input policies. For positional policies, the interactor need do nothing special. As long as it remains enabled, agents under the positional policy will deliver input to it whenever it is *picked* (*i.e.*, it is determined to be under the screen position of the cursor; see below).

For focus-based agents, the interactor must register with the dispatch agent managing that particular type of input. Interactors may request to be the sole focus of a particular type of input, or may request to be included in a focus set for that input. An interactor may lose the input focus as a result of another interactor's request, in which case it is notified via a method call within the relevant input protocol.

For example, when a text editing interactor is clicked, it asks to become the (exclusive) text focus by invoking:

```
text_focus_agent.set_focus_to(this);
```

This in turn causes an `end_text_entry()` message to be delivered to the previous text focus (if any), allowing it to update its appearance accordingly (*e.g.*, remove the text cursor) and a `start_text_entry()` message to be delivered to the new focus also allowing it to update its appearance (*e.g.*, arrange to draw the text cursor). When keyboard input is received, it is processed by the text dispatch agent, which translates keys with special meaning (*e.g.*, backspace, delete, enter, *etc.*) into method calls for the current text focus interactor (which implements the `text_acceptor` input protocol) that make sense given the

semantics of text editing. For example, delete and backspace are translated into a call to `delete_char()`.

Note that because the input is delivered to interactors in a form related to what it is used for (text editing) rather than how it is produced (key strokes), interactor objects are typically not directly dependent on the low-level form of input driving them. For example, it would be a simple matter to produce a new text agent that got its input in some other way (such as *via* a recognized pen strokes [5], or a "soft" keyboard [12]). So long as the same input protocol is employed, interactors using it need not be updated to allow introduction of a new input source. This allows use of alternate devices and new input methods without giving up use of (or requiring changes to) the existing interactor library.

In addition, the layer of translation provided by dispatch agents allows the toolkit to perform some of the work that would normally be done by the interactor, for example interpreting special keys. Since this work is done in a reusable fashion inside the toolkit infrastructure, it does not have to be re-implemented each time a new interactor class is created. Similarly, this provides a central place for providing other new services that can be reused. For example, the text agent allows a character filter object to be supplied that affects which characters are passed on, and/or the translation into special actions. Standard filters are provided for things like numbers-only, translation to all upper or all lower case, and removal of white space. By providing new filter objects, further customization can occur, while still allowing any and all interactor classes which use text input to benefit from this work (potentially even after they are implemented).

Picking

To dispatch input positionally, it is necessary to know what input sensitive objects lie under the current cursor position. The process of finding such objects is traditionally called *picking*. There may be a series of objects which should be considered "under" a given position. For example, it is often the case that an interactor as well as its parent container, and the container's parent container, *etc*. all produce output which overlaps a given point. To deal with this situation, the process of picking produces an ordered series of objects, which are candidates to receive input.

Normally the first (and highest priority) object in this series is the one that has been drawn last (since its output then appears on top of other objects). This would typically be a leaf node in the interactor tree. In recognition of this, some toolkits use what amounts to a bottom up traversal of the interactor tree starting at the last drawn interactor whose bounding box overlaps the cursor position. However, such a rigid policy precludes several interesting manipulations of picking which we will describe below. Further, it does not handle non-rectangular or non-opaque interactors well, since overlapping siblings (or cousins) of the nominal "bottom most" interactor never get a chance to receive input (even though their shape on the screen may not have been overdrawn at the location of the cursor, and they appear visible through or past their non-rectangular or non-opaque sibling).

To handle picking more flexibly, the subArctic system creates an explicit list (called a *pick collector*) to represent the picking sequence, and delegates the details of how to fill in this list to the interactor objects themselves. Picking is performed by a top-down recursive traversal of the interactor tree which normally accumulates picked items as the recursion returns, thus picking occurs by default in the normal bottom up order.

Specifically, each interactor implements a `pick()` method which takes an x,y screen position (in its local coordinate system) and a `pick_collector` list. A default implementation of this method is provided by the toolkit in a base class for all interactors, and so does not require additional work for interactor programming in simple cases. This implementation first recursively traverses each child object in reverse drawing order – passing a reference to the same pick collector and adjusting the x,y position to be in the child's coordinate system – then determines locally if it should itself be considered to be picked. By default this is done by testing its own enabled status, and if enabled further testing with its own `picked_by(x,y)` method (which in turn defaults to doing a simple bounding box test). If `picked_by(x,y)` returns `true`, the object is considered to be picked, and it adds itself to the end of pick collector list (normally after any of its children, which have just added themselves to the list in the previous, recursive call to `pick()`).

Note that if a container objects overrides the default drawing order for its children, draws only some of its children, or otherwise has special needs with respect to picking, this can be properly reflected by overriding its `pick()` method correspondingly. This flexibility is important since a fixed picking order would, for example, preclude the proper operation of container interactors with pickable components drawn both on top of its children and underneath them. (Consider for example a toolglass [1] object that draws property manipulation "halos" behind and around certain of its child objects and also provides a series of controls along its frame on top of its children.)

Further, explicitly representing the pick sequence as a data structure allows several other interesting effects to be supported. For example, the shadow drag container illustrated in Figure 1, provides dragging behavior for the group of objects placed inside it, regardless what kind of interactors they are. If a locator button is pressed down over one of the child objects (or a child's child, *etc*.), the entire group is highlighted with a shadow effect (by drawing them twice – once with a special drawing context which turns all colors to gray [2], and then again normally at a small offset), and begins to follow the cursor.

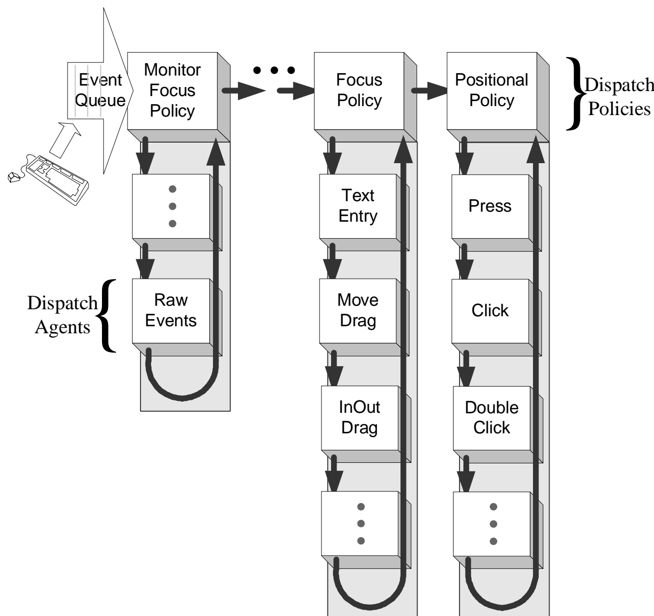

Figure 3. The Flow of Events.

To implement this properly, the container needs to consider itself picked if and only if one of its child objects is picked. Further to allow other aspects of the objects to operate normally, they need to appear on the pick list as well (but after the container). To accomplish this, the container's `pick()` method creates a local (empty) pick collector which it passes to the child objects in the normal order. If this pick collector returns with one or more picked interactors on the list, the container adds itself to the original pick collector, followed by all the interactors on the local pick collector (in order).

While this example is somewhat out of the ordinary, we think it illustrates an important point. It shows an interaction that is unusual and was not anticipated when the input system was originally designed, but is quite useful. Because of the flexibility of the system it can be accommodated in a robust fashion (*i.e.*, supporting any child interactors without regard to their type), with relative ease (*i.e.*, the new `pick()` method was implemented in less than 30 lines of code). The capability to do this easily in both small and large ways is fundamental to the goal of escaping the stifling effects of fixed widget libraries.

Fitting it all Together – Flow of Events

How do all of these different architectural components work together? As illustrated in Figure 3, each incoming event is passed to the current series of dispatch policies in a simple priority order with the highest priority policies (typically focus-based) getting the first opportunity to dispatch an event. Each dispatch policy object attempts to deliver this event using one of its dispatch agents. All the standard dispatch policy objects supplied with the system to date also use a simple priority order across dispatch agents. As a result, a given event will be passed to successive input dispatch policies, each of which will give it to a succession of dispatch agents controlled by that policy until the event is delivered to an interactor object, and the object responds that it has acted on the event and *consumed* it. An event is considered to be consumed when an interactor returns `true` from one of the method calls within an input protocol (*e.g.*, from `drag_feedback()` in the `simple_draggable` protocol described above). Note that positional policies traverse their agent lists multiple times: once for each object on the pick list associated with the current event, until an object consumes the event (picking is performed only once per event and cached across all positional agents). Focus agents do not use the pick list.

This overall structure has the advantage of being highly extensible or *pluggable*. If either an overall policy, or the details of a particular kind of input, need to be customized or extended, it is a simple matter of placing or replacing an appropriate object in one of the lists. Importantly, no changes to the toolkit infrastructure are needed – the toolkit still simply passes the event through the policy list. This means that fairly arbitrary changes to the way input is handled can be done with a minimum of difficulty, and with a little care, in a way that allows all the existing interactors to operate normally (and many times to even benefit from the changes).

EXAMPLES

As indicated above, subArctic's input system simplifies the creation of interactors, and enables various application-independent effects and architectural extensions to be created. This section describes some specific things that subArctic makes possible. While several of these examples are interaction techniques that were research contributions published elsewhere [6,7,8,9,10,11], we focus here on the input handling infrastructure issues that enabled them to be created.

Dragging

As described above, dragging is a feature that subArctic easily supports. In particular, dragging is a focus-based operation, and different specialized dispatch agents support the semantics of operations such as resizing an interactor by dragging its corner and moving an interactor. In addition to the kind of basic functionality described previously, subArctic's drag dispatch agents provide the following services:

Conventional Dragging: The toolkit provides standard dispatch agents to support dragging for the purpose of moving an object, as well as dragging to support resizing, and a generic form of dragging which can be used for other purposes. Although the dispatch agents for move-dragging and grow-dragging are very similar, they each compute specialized parameters suited to their particular task (*i.e.*, derived from the relative movement of the drag and offset by the initial position or size of the interactor). In particular, an interactor receiving this type of input needs only copy the

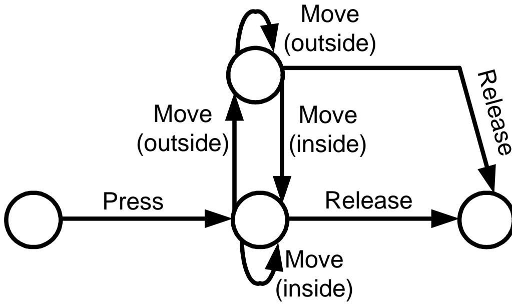

Figure 4. Finite State Controller for In/Out Dragging

parameters to either its position, or size, in order to implement these common actions.

Constrained Motion: Each of the conventional dragging dispatch agents allows an interactor to install a *drag filter* that limits motion to a region defined by the filter. Standard filters are provided for keeping a moved object within its parent, within an arbitrary rectangle, or along a defined line. Like the text translation filters described earlier, this capability allows a richer set of interactions to be supported without necessarily requiring changes to the interactors which use this form of input

In/out dragging: In/out dragging is provided for interactors, such as a button or check box, which are only interested in knowing when the drag enters and exits their extent (*e.g.*, so that they can update their highlighting). While this dispatch agent responds to the same set of events as other drags, it interprets them with a slightly different finite state controller as illustrated in Figure 4.

Semantic snap-dragging: Semantic snap-dragging, described in [6,7], is a form of dragging in which selected targets act as gravity wells [18] for the object being dragged. When a dragged object nears a target (such as a wastepaper bin) that passes appropriate semantic tests (such as indicating that deletion of the type of object being dragged is "safe"), it snaps to that target rather than following the exact path of the user's mouse.

Snapping occurs by considering active feature points within snapping distance of a target. Each `snap_draggable` object advertises a set of feature points. Each active feature point is eligible to snap to a snap target object (information about snap target objects is kept by the dispatch agent that handles snap dragging). Snap target objects include a geometric test to determine if a point is considered close enough to the target to snap to it. For each geometrically possible snap, semantic tests are also performed to determine if the snap is semantically acceptable. The closest candidate snap that passes semantic tests (if any) is performed.

While dragging objects is a capability supported by nearly any toolkit, because the subArctic mechanisms are specialized, it is quite easy to add dragging behavior to new interactor types. Because an infrastructure for reuse is provided, devoting extra effort to create more complex dragging interactions (*e.g.*, semantic snapping) is a good investment, and subsequently this capability is easy to take advantage of in new interactors.

Currently Selected Sets

Another common interface capability is the selection of objects, and the maintenance of a currently selected set of objects. Interaction patterns for selecting a set of objects (based on clicking to select, and shift-clicking and/or control-clicking to extend) have been established across a range of applications, so in subArctic, this capability is supported by a specialized dispatch agent (under the positional dispatch policy). This agent manages a currently selected set based on the conventional interaction sequences, and delivers input at the level of notifications to interactors that they have entered or left the currently selected set. This represents another example where a common pattern of interaction can be "moved up" into the toolkit. To take advantage of this capability new interactor classes need only declare that they are `selectable` and implement the `select()` and `unselect()` methods (*e.g.,* to add and remove selection highlights). The selected-set dispatch agent takes care of the rest of the details, and makes the resulting currently selected object set available to the application with a simple API.

Lenses

As mentioned above, the subArctic input system makes it very easy to create non-rectangular (and even non-contiguous) interactors, including toolglasses and magic lenses [1,9] (or simply *lenses* for short). These see-through interactors sit "above" an interface, and may change the look and behavior of things seen through them (for example, converting a color drawing into black and white, or drawing behind or over objects to add additional information), and may allow interaction with either themselves, or the interactors they affect, or both. For

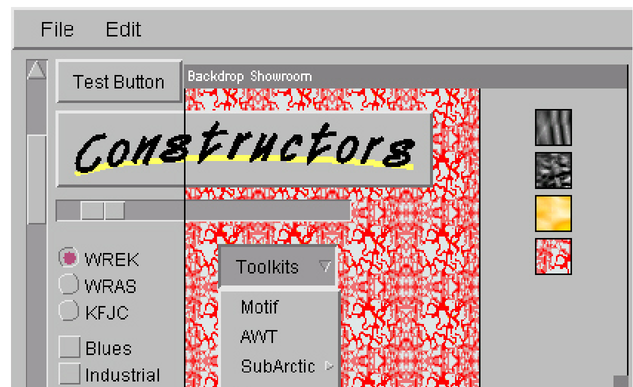

Figure 5: A lens that previews the use of a background pattern behind an interface.

example, in Figure 5, the user has clicked on the pulldown menu *through* the lens, causing it to appear. A lens may also include a border or control area that allows the user to interact with the lens itself. For example, in Figure 5, the user can move the lens by dragging its title bar, and change the background pattern it displays behind the normal scene by clicking on one of the pattern icons at the right.

To create a lens requires the ability to systematically modify output (as discussed elsewhere, subArctic supports this in flexible and systematic ways [2]), and to intercept some (but not all) input. The latter problem is solved by using subArctic's picking infrastructure to indicate which areas inside a lens' bounding box are interactive and which are not. Lenses which filter their drawing (*i.e.*, omitting some interactors which would normally be drawn) can be implemented by modifying the pick list to exclude filtered interactors, so they will not receive input, as needed.

Animation

The subArctic toolkit includes specialized support for handling animation in a robust and flexible way. While animation may seem to be strictly a matter of output, it is driven by the passage of time. Because the passage of time is an "action of interest" that needs to be handled in the same frameworks as other "actions of interest" such as user manipulation of input devices, it is convenient to model the passage of time as a series of *tick* events delivered with other inputs.

However, simply delivering tick events provides only very basic help in creating animated objects. Like other forms of input the subArctic input system goes much further than this by providing a richer and higher-level abstraction which reflects more of the way the input is used. Rather than simply delivering timed ticks, the animation dispatch agent uses the richer abstraction of *animation steps*, which are scheduled, sequenced and paced along trajectories established by the programmer. As described in [11] these abstractions make it easy to apply sophisticated effects such as slow-in/slow-out motion, anticipation and follow-through, and squash and stretch. Again, the structure of the toolkit input architecture makes these kinds of higher-level input abstractions easy to use for new interactors, and allows the effort of creating rich abstractions to be readily reused.

Dwell and Trill

Dwell and Trill are two common features that can be easily supported with the subArctic input system. An interactor supporting *dwell* reacts when the mouse hovers over it for a designated amount of time. An example is an interactor that supports tooltips. Rather than implementing a one-time solution in the interactor itself, tooltips are supported by a positional dispatch agent that listens for tick events and keeps track of the length of time the mouse has been over any interactors that implement the dwelling protocol. These interactors are informed when an appropriate dwell time has elapsed, and again when the user moves away.

An interactor supporting *trill* would repeat an action if a key or locator button were held down over it and not released within a certain time interval. For example, "holding down" a scroll bar arrow could cause repeated motion of the thumb. This interaction can be implemented in a fashion analogous to dwell, with a positional dispatch agent tracking tick, press, and move events, which are translated into higher level press and press-held inputs.

GLOBAL CHANGES TO INTERACTION

The primary aim of the subArctic input architecture is to support the kind of rich and varied new interactions briefly touched on above in a way that supports reuse, and makes new custom interactor types easy to create. However, the flexibility of the system also makes it possible to make more radical modifications, such as making global changes in the way inputs are handled.

Hyperlinking from Everything

Early in the deployment of the subArctic system we were contacted by a researcher wishing to create an interface to their experimental generic hyperlinking system. This system worked by using an external association table which maintained relationships between an arbitrary internal identifier (such as a Java object hash code) and external content which was to be associated with the corresponding object. The researcher wished to create a system in which every interactor object could potentially have content associated with it, and have that content accessible when the user held down the control key. In particular, when the control key was held down and any object was then clicked on, the hyperlinking system was to be consulted to see if there was an association between that object and some previously linked content. If so, the content would be brought up in a new window. If not, the click would be processed normally.

In a toolkit with a conventional input model, this kind of capability requires a radical change. Every single interactor which handles locator button presses would need to be modified to add this new hyperlinking capability. Even in cases where source for the full library is available, this is a daunting task, and even if this task is undertaken, this capability would be broken whenever a new interactor class was created by someone else.

On the other hand, making this kind of change is very easy within the subArctic framework. One need only add a new positional dispatch agent which intercepts press events, and install it before the standard "press" agent. This new agent checks if the control key is held down, if it is, it passes the hashcode of the interactor it would normally be positionally dispatching to (*i.e.*, taken from the pick list supplied by the positional input policy) to the hyperlink system. If the hyperlink system finds an associated link and acts on it, it consumes the press input. If the control key was not held

down, or there was no association found for the object, then the event is not consumed and continues through the system normally.

Because of the flexibility of the subArctic input system, this otherwise radical change, which affects the action of many different interactors in the standard library, can be accomplished very easily (in about 20 lines of code) without modifying any of the existing interactor library. Further, this change will work with any new interactor types added later.

OTHER BENEFITS

In addition to enabling the easy introduction of new input techniques, representing both minor and large changes, the existence of a fully developed structure for translating and delivering inputs can have additional benefits which go beyond direct handling of inputs.

Recording input

There are two ways to record information about input in subArctic. The simplest approach can be used to record input *events*, that is, the stream of input produced by the user. subArctic support this via a dispatch agent under the monitor-focus policy. This agent simply delivers all the events to a recorder object, but otherwise does not consume or modify them.

A more interesting capability enabled by the system is the ability to record basic information about what inputs were used for. In particular, because the system does the first level of input interpretation, *e.g.*, treating inputs as a particular form of drag, or structured as text editing, *etc.*, and does this in a standardized way, it is possible to capture some semantics related to recorded inputs in addition to their surface form. This is done by recording the input protocol and particular method within that protocol, used to deliver each input, along with the object it is delivered to. This capability has been used, for example, to build a tool for automatically generating Keystroke-Level Models from interface demonstration sessions [8]. Here the exposed semantics of the input was used to successfully place the *mental* (M) operators required for these models, in a way that would not have been possible from the event log alone.

Introspection for Input Adaptation

The existence of input protocols, and access to information about when they are used, makes it possible to reason about the way different interactors use input, and to act on that knowledge. For example, it is possible to enable keyboard-based navigation to any interactor, and to create keyboard commands to control different aspects of an interactor by mapping them to different methods in an interactor's input protocol. This approach was used, for example, to re-implement a slightly simplified version of Mercator, an X Windows system that renders interfaces in audio for the blind [3], in subArctic. In addition to changing the output modality of an interface, Mercator (and our re-implementation of Mercator) supports keyboard-based navigation, and removes any dependencies on visually-based or locator-based navigation.

LIMITATIONS, LESSONS, AND EXTENSIONS

The subArctic toolkit was first widely released in 1996, has been downloaded tens of thousands of times, and has been used for teaching user interface software courses at several universities. Through our own experience, and those of our users, we have seen subArctic's input infrastructure used to create a wide variety of interaction techniques and tools, just a few of which have been described here. These experiences have largely been positive. However, as a part of this experience we have also learned several lessons about how future systems might improve the architecture by small additions or changes.

Picking

Although explicitly representing the results of picking as a list which may be manipulated is a very powerful model for controlling the target of incoming input, our choice to pick based only on a point (*e.g.* locator coordinates) has some limitations. One possible alternative is to support an "area pick" of which a point is a special case. While this raises potential new issues (*e.g.*, what happens if the input event area partially overlaps an interactive area), it would also increase the power of the input system. For example, this would allow subArctic to more easily support the creation of an area cursor that makes it easy to hit a target with low dexterity motions [20].

Policy and Agent Priorities

SubArctic's policies and agents are kept in a simple, ordered list, which represents their relative priorities for receiving event. While the priority of a policy or agent can be changed dynamically by changing its location in that list, other, more complex ways of selecting a dispatch policy are not supported. For example, a privacy-sensitive policy priority system might skip the monitor policy entirely if the user's privacy preferences are set very high, and the input contains text.

Controlled Access to Focus

Currently to request focus from a focus-based dispatch agent an interactor communicates directly with the agent. Once an interactor has the focus, there is no way to interpose between it and the relevant focus dispatch agent. We believe it would be a slight improvement to route all requests for input focus up through the interactor tree, rather than having interactors communicate directly with the agents. Then a container could selectively consume some input, while passing the rest on. For example, a special container could consume and handle keystrokes corresponding to commands while allowing plain text through to a child text entry interactor. This makes it very easy to change the way commands are handled by simply replacing the parent container.

Hierarchical events

Currently, the subArctic system only treats original device inputs as events. Another potential extension would be to allow additional higher-level input types to be treated as events, and to build hierarchical events where the higher level events are linked to the lower level events used to create or trigger them. Extending to hierarchical events allows for better infrastructure to encode commands and other semantic information [13]. This, in turn, could enable structured support for undo and recognition.

One particular application of hierarchical events is input coming from a recognizer. SubArctic can handle simple forms of recognition-based input without modification. It is possible to create a dispatch agent that accepts, for example strokes from a pen serving as the locator device, sends them to, for example a gesture a recognizer, and then dispatches the results to interested objects. However, far more flexibility is gained by storing hierarchical information (about what events were derived from), and allowing recognized inputs to be redispatched as events through the same mechanisms as device oriented events.

Extensions to Support Recognition and Ambiguity

In addition to the usefulness of hierarchical events, we learned several interesting lessons in the course of expanding subArctic to fully support recognition [16].

Ambiguous events

Simple hierarchical events are not sufficient to encode all of the information important to users when recognition is occurring. A further necessary expansion is to add support for ambiguity (events that may or may not have happened, such as competing recognition alternatives). This can allow an interface to display competing alternatives to users, who may then select or reject them (*mediation*). Interactors should be able to receive input before or after ambiguity is resolved, and be notified if an ambiguous event is accepted or rejected by the user.

Expanded model of event consumption

Along these lines, a binary model of event consumption is not entirely sufficient. In subArctic, an interactor may consume an event (in which case, no one else may consume it and dispatch halts), or reject it (in which case dispatch continues). An expanded model might have several levels of consumption. One key addition is the ability to *display* information related to an event without consuming it. For example, an interface element may wish to display tentative feedback from an ambiguous event, but likely would not consume it until it was accepted or confirmed.

Better communication with input producers

When we added support for recognition to subArctic, we found ourselves in a situation where input was coming not only from hardware but also from recognition software. Recognizers often benefit from knowing what happens to the input they create (for example, they may maintain a user model, or learn from the fact that certain choices are rejected). Also, an interface may be able to generate constraints on valid input that could be passed back to recognizers to help guide their choice between ambiguous alternatives. A provision for a reverse communication channel from input consumers back to input produces could help enable this.

RELATED WORK

Several prior or contemporary toolkits have used input systems with aims related to those of the subArctic system. For example, the standard Java GUI toolkit, *Swing*, provides a very flexible input model based on the concept of *listeners*. Objects interested in receiving notification of various actions of interest (including user manipulation of input devices) may register as a listener with the object which manages that action and/or provides notification for it. When the action of interest occurs, all objects which have registered as listeners receive a specifically typed method call whose parameters provide details associated with that notification.

In the terms used by the subArctic system this can be seen as quite similar to use of single message input protocols dispatched through focus-based dispatch agents. It is more flexible in the sense that essentially any object can serve as the source of notification. Further, this mechanism is quite amenable to use in areas beyond input handling, and so a single mechanism helps support several aspects of interface construction. On the other hand the Swing listener-based approach, while very general, provides substantially less specialized support for input handling. For example it does not provide flexible mechanisms for picking or positional event dispatch. Input handling capabilities similar to the subArctic model could be built within the Swing framework. However, they are not directly provided by the toolkit, hence would require substantial effort to implement, would not work with the current interactor library, and likely would not be amenable to reuse.

Another input model of interest is the Garnet/Amulet model [15]. This model was developed based on very similar goals to ours, notably a desire to ease the creation of new interactive objects by automating common input patterns within the toolkit, rather than requiring them to be implemented within interactive components. Interestingly, these systems took an approach to achieving this aim which is almost the opposite of the subArctic approach. Instead of supporting an extensible set of agents each of which implements a different specialized finite state controller, the Garnet/Amulet model seeks to build a very small number of more universal finite state machines – with the most common interactions being handled by a single cleverly designed (and highly parameterized) finite state controller. To make use of this controller in support of a particular interactive pattern, one provides a set of controlling parameters that change which inputs invoke which

transition, how feedback is handled, and many other aspects of the interaction.

The advantage of this approach is that it is very easy for the programmer to use. Rather than having to understand and select from a large library of input protocols, the programmer can create many relatively common interaction patterns very simply with a few parameters. However, this approach relies heavily on the design of the few general controllers. While these have been cleverly designed to cover many interactive situations, in later versions of the model, more than 30 different parameters are needed to accomplish this (which begins to erode the simplicity of the approach). Further, the flexibility and extensibility of the system is inevitably bounded by these controllers which cannot be readily extended or replaced to meet the unique needs of particular interfaces.

CONCLUSIONS

We have presented subArctic's input handling infrastructure. SubArctic's architecture separates the job of selecting input targets (picking), and extracting semantically-relevant information from raw input (performed by dispatch policies and dispatch agents), from that of providing feedback and application functionality (performed by interactors and the application). This separation of concerns makes it possible to encapsulate interactors in containers that add functionality (such as the shadow drag container in Figure 1); modify input before it arrives at an interface (adding recognizers or changing input in arbitrary ways); and create advanced interactions such as lenses and other non-rectangular interactors. In addition to its powerful architectural features, subArctic includes a comprehensive and sophisticated library of dispatch policies and dispatch agents. This library includes reusable support for common interaction patterns such as text entry, a variety of forms of dragging (including moving, resizing, constrained motion, snapping, and in/out dragging), monitoring input, animation, and more. Overall the subArctic input architecture makes it easy to expand interaction beyond a fixed widget set by supporting custom input technique – it allows new interactions to be explored without giving up the use a well developed library of conventional interactors.

REFERENCES

1. E. A. Bier *et al.* "Toolglass and magic lenses: The see-through interface", In *Proc. of 1993 ACM SIGGRAPH Conference*, pp. 73-80, 1993. ACM Press.
2. W. K. Edwards, *et. al*. "Systematic output modification in a 2D UI toolkit", In *Proc. of ACM UIST'97*, pp. 151–158, 1997. ACM Press.
3. W. K. Edwards and E. D. Mynatt. "An architecture for transforming graphical interfaces", In *Proc. of UIST'94*, pp. 39-48, 1994. ACM Press.
4. H. Gajewska *et al.* Why X is not our ideal window system. Software – Practice and Experience. **20**(S2):137-171, 1990.
5. D. Goldberg and A. Goodisman. "Stylus user interfaces formanipulating text", In *Proc. of ACM UIST'91*, pp. 127–135, 1991. ACM Press.
6. T. R. Henry, *et al.* "Integrating gesture and snapping into a user interface toolkit", In *Proc. of ACM UIST'90, Proceedings of the ACM SIGGRAPH Symposium,* pp. 112– 122, 1990. ACM Press.
7. S. E. Hudson, "Semantic Snapping: A Technique for Semantic Feedback at the Lexical Level", In *Proc of CHI'90*, pp. 65-70, 1990. ACM Press
8. S. E. Hudson *et al.* "A tool for creating predictive performance models from user interface demonstrations", In *Proc. of UIST '99,* pp. 93-102, 1999. ACM Press
9. S. E. Hudson *et al.* "Debugging lenses: A new class of transparent tools for user interface debugging", In *Proc. of ACM UIST'97*, pp. 179-187, 1997. ACM Press
10. S. E. Hudson and I. Smith. "Supporting dynamic downloadable appearances in an extensible UI toolkit", In *Proc. of ACM UIST'97*, pp. 159–168, 1997. ACM Press.
11. S. E. Hudson and J. T. Stasko. "Animation support in a user interface toolkit: flexible, robust, and reusable abstractions", In *Proc. of UIST '93,* pp. 57-67, 1993. ACM Press.
12. I. S. Mackenzie and S. X. Zhang, "The design and evaluation of a high performance soft keyboard", In *Proc. of CHI'99*, pp. 25-31, 1999. ACM Press
13. B. A. Myers and D. S. Kosbie. "Reusable Hierarchical Command Objects", In *Proc. of CHI'96,* pp. 260-267, 1996. ACM Press
14. J. A. Landay and B. A. Myers. "Extending an existing user interface toolkit to support gesture recognition", In *Proc. of INTERCHI'93 – Adjunct Proceedings*, pp. 91–92. 1993. ACM Press.
15. B. A. Myers. A new model for handling input. ACM Transactions on Information Systems, **8**(3):289–320, 1990.
16. J. Mankoff *et al.* "Providing Integrated Toolkit-Level Support for Ambiguity in Recognition-Based Interfaces", In *Proc. of CHI'00*. pp. 368-375, 2000. ACM Press.
17. W. M. Newman. A system for interactive graphical programming. In *AFIPS Spring Joint Computer Conference.* 1968, pp. 47-54.
18. I.E., Sutherland. Sketchpad--A Man-Machine Graphical Communication System, in *AFIPS Spring Joint Computer Conference*, May 1963.
19. A. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. IEEE Transaction on Software Engineering. **11**:699-713. 1985.
20. A. Worden *et al.* "Making computers easier for older adults to use: Area cursors and sticky icons. In *Proc. of CHI'97*, pp. 266-271, 1997. ACM Press.