A Framework for Robust and Flexible Handling of Inputs with Uncertainty

Julia Schwarz, Scott E. Hudson, Jennifer Mankoff
HCII, Carnegie Mellon
5000 Forbes Ave, Pittsburgh, PA 15213 USA
{julia.schwarz, scott.hudson, jmankoff}@cs.cmu.edu

Andrew D. Wilson
Microsoft Research
One Microsoft Way, Redmond, WA 98052
awilson@microsoft.com

ABSTRACT

New input technologies (such as touch), recognition based input (such as pen gestures) and next-generation interactions (such as inexact interaction) all hold the promise of more natural user interfaces. However, these techniques all create inputs with some uncertainty. Unfortunately, conventional infrastructure lacks a method for easily handling uncertainty, and as a result input produced by these technologies is often converted to conventional events as quickly as possible, leading to a stunted interactive experience. We present a framework for handling input with uncertainty in a systematic, extensible, and easy to manipulate fashion. To illustrate this framework, we present several traditional interactors which have been extended to provide feedback about uncertain inputs and to allow for the possibility that in the end that input will be judged wrong (or end up going to a different interactor). Our six demonstrations include tiny buttons that are manipulable using touch input, a text box that can handle multiple interpretations of spoken input, a scrollbar that can respond to inexactly placed input, and buttons which are easier to click for people with motor impairments. Our framework supports all of these interactions by carrying uncertainty forward all the way through selection of possible target interactors, interpretation by interactors, generation of (uncertain) candidate actions to take, and a mediation process that decides (in a lazy fashion) which actions should become final.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces.- Graphical user interfaces.
General terms: Performance, Human Factors.
Keywords: Input Handling, Ambiguity, Recognition.

INTRODUCTION

Input handling in most modern interface toolkits depends on an established framework for modeling and responding to input that has been tuned over time to the needs of conventional graphical user interfaces (GUIs). However, this framework makes a number of assumptions about the nature of the inputs it deals with. For example, standard GUI toolkits implicitly assume inputs are certain to have occurred as reported. As we move to promising new technologies such as computer vision, free-space gesture recognition, pen input, and touch sensing, this assumption of *certainty* is being violated. It is no longer the case that reports about inputs are completely correct, without ambiguity or significant error. For example, in touch input the location of a touch 'click' is partly *ambiguous* simply because a user's finger touches, not at a point, but an area (and the user cannot see how their contact area overlaps small objects underneath their finger). Similarly, a recognizer may produce one or more *uncertain* estimates of user intent.

If these inputs are processed using a conventional input framework, their uncertainty is resolved quickly and often simplistically. The result may seem arbitrary or unpredictable. Small errors in interpretation may lead to incorrect application actions that are difficult to recover from.

One solution to this problem is to develop interactive systems that are designed to work smoothly with one or more types of recognized input. Along these lines, there has been considerable work on multimodal systems, which in part aims to mitigate interpretation errors [18,19]. While very promising, this body of work has largely entailed adoption of a radical new input framework, designed to integrate information from multiple sensor channels. An alternate approach is to integrate uncertain input into the conventional input handling framework, making it possible to take advantage of existing buttons, sliders, menus, *etc*. that the vast majority of users are familiar with and have invested substantial learning in.

In this paper we describe the details of a new input handling framework that is compatible with the existing approach, accurately tracks the probabilities of alternative uncertain inputs, and provides mechanisms for dispatching input, intelligently making decisions, providing feedback, and acting in the light of that uncertainty.

Our framework can temporarily entertain multiple possible uncertain inputs (until the interface needs to perform an action with permanent consequences beyond feedback). This puts off the decision of which possibility is correct, so that it is possible to use more information to make the final decision. For example, Figure 1 shows two possible interpretations of a press-move-release sequence on a touch screen (left: dragging an icon; right: resizing a window). Rather than deciding which is correct at the moment the screen is touched, our framework allows interactors to temporarily provide feedback about both possibilities. Addi-

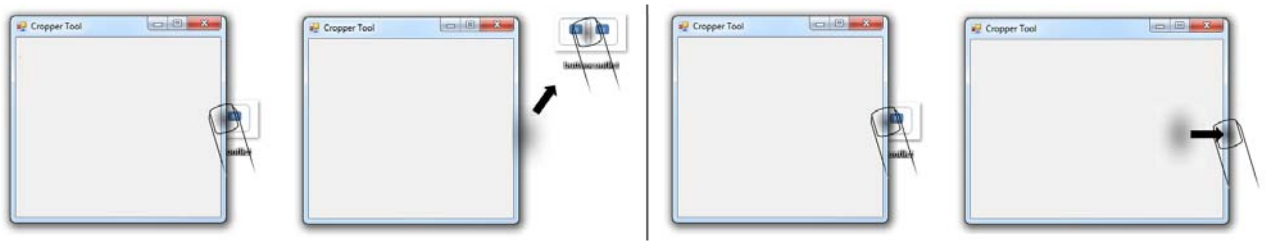


Figure 1: Illustration of implicit disambiguation for touch input. **Left:** Users presses down and moves diagonally, moving the icon. **Right:** User presses down and moves horizontally, resizing the window instead. When the user presses down, both interactors are equally likely to respond. The user's motion later disambiguates their intention and interactors in our framework respond appropriately.

tional information such as the direction of motion can then be used to make a decision. For example, if the user moves vertically, he or she is unlikely to be resizing the window since only horizontal movements affect window width.

In the next section, we review the major tasks inherent in conventional input handling. We compare this to our framework for handling uncertain input, which preserves the basic mechanisms of the conventional event-oriented input model. Next, we discuss our framework and implementation in more detail. Like the conventional input framework, our framework provides a general and reusable structure – including a model for probabilistic input and a means to make informed decisions on the basis of accurate probabilistic tracking of alternatives. This allows different interactors taken from a library to be used largely without regard to what other (probabilistic or conventional) interactors are being used in the same interface. Finally, we give an overview of how our framework supports existing interaction improvements and provide six examples of how the framework can be used to create improved interactions.

COMPARING CONVENTIONAL AND UNCERTAIN INPUT

Nearly all modern user interface toolkits implement interfaces as a mostly independent collection of interactive objects managed by an infrastructure for handling input, producing output, and numerous other tasks. The input handling component of these systems is now well evolved and works very well for its task. This conventional input handling framework can be thought of as providing four major capabilities: (1) *modeling* of inputs, by providing a way to record all the relevant details of what input happened (in the form of *event records*), (2) a process for *dispatch* of those events – deciding which interactor object(s) should receive and handle a given input, (3) *interpretation* of those events by the interactors in terms of their own interactive state, and finally (4) *action* by the interactor (including presentation of user feedback and requesting action from the application). By giving interactors structured, yet independent control over how they respond to input, this framework gains *uniformity* and *extensibility*. Interactors from an expandable library can be used together on a *mix-and-match* basis with little explicit cooperation. This same basic structure is appropriate for inputs with uncertainty. Figure 2 gives an overview of this process and below we discuss each component, comparing conventional input and its probabilistic equivalent.

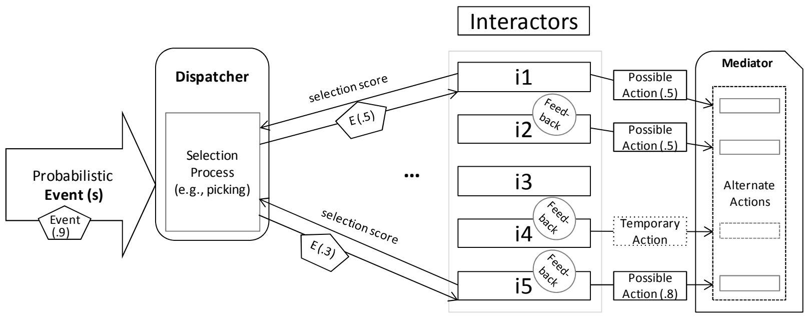
Modeling Input

In the input framework employed by most modern graphical user interfaces, all input is modeled as a discrete sequence of *events*, each of which records information about some significant occurrence that may affect an interactive program. While the exact form of event records used in various systems varies somewhat, with a small amount of abstraction most can be characterized as recording the following five categories of information:

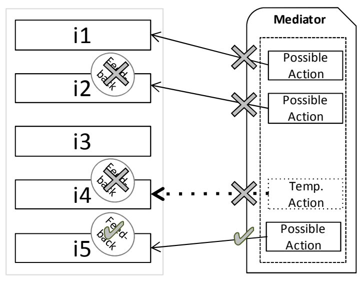
- What **type** of input happened. A record of what occurred, such as "a keyboard key was pressed down". The type of input may determine the exact structure of the remaining information in the event record.
- A detailed **value** describing what happened, such as "Key cap #12" for a key press event.
- **When** the input happened. A timestamp.
- **Where** the input happened. Most typically the (x, y) position of the primary pointing device.
- Important **context** associated with the input. A record of other values that might modify the meaning of the input. A conventional example is the state of the modifier keyboard keys (ctrl, alt, etc.).

When an input is less certain, conventional frameworks are unable to model this uncertainty. As described above, Figure 1 demonstrates a touch-based interaction involving uncertainty. Here the user is pointing to a screen location very close to both an icon and the edge of a window partially covering it. The user's intent may be to move the icon or to resize the window. Although the user has a specific intent, it may not be obvious what is intended at the moment the screen is touched. With conventional input, the user's action will produce a *press* event that is represented as a single point (usually the center of the touch area). As a result, very small variations in location of the user's finger have an effect that is difficult to predict.

To accurately model uncertainty, input event properties need to be expanded from a single fact to estimates representing a range of possibilities, which we will represent using a *probability mass function* (PMF). A PMF is a function which describes the relative likelihood that some discrete random variable has a particular value or that a continuous random variable falls within a finite range. For example, a Boolean variable can be represented as a PMF with probability of 0.2 of being true and 0.8 of being



(a)



(b)

Figure 2: Overview of how a probabilistic event is handled in our framework. **(a)** When an event arrives, a selection process is used to assign a custom probability to it for each interactor. Interactors then decide whether to take action on it. Possible actions (or temporary actions such as feedback) are sent to a mediator **(b)** At some point (either when there is a clear winner in terms of probability, or a finalization request is made) the mediator picks an action, based on probability. All other actions are canceled.

false and the PMF for a position indicates for each pixel the probability that the true position falls within the pixel. Since a PMF is derived from an underlying probability distribution, the integral over all possible values of the random variable must sum to 1. In our framework, all properties of an event may be represented by a PMF instead of a single certain value. In addition, the input event as a whole is assigned a probability, indicating the likelihood that it (as opposed to a different input event) is what happened. The event probability is often 100%; however something like a recognizer that produces multiple possible alternatives might assign differing likelihoods to each alternative, each of which would be represented by a separate input event.

Event Dispatch

Conventional event dispatch consists of *selecting* an interactor (or, occasionally, a set of interactors) to receive each event, then delivering that event. In a conventional framework, selection is a cooperative process between the *dispatch* subsystem and the interactors it might deliver input to. An ordered set of candidate interactors is selected as potential recipients for an event based on factors such as location and type of the event. The dispatch system then queries each one, in order, to find out if it will use the event. An interactor may determine, based on its internal state (for example, if it is disabled), that it does not want an event, or may consume the event (take action on it), in which case dispatch is typically complete (no further interactors are queried). Each interactor implements a standard interface that encapsulates this process, facilitating the selection of which interactor should receive an event and then the actual delivery of that event.

Typical strategies for selecting interactors include *positional selection*, where the on-screen position of the primary locator determines which interactors receive events (*e.g.*, how a menu receives press events), and *focus-based selection*, where one interactor is granted exclusive access to input, regardless of position (*e.g.*, how a text box receives keystroke events). In the case of Figure 1, positional selection would be used. Conventional systems make use of a *picking* mechanism to generate an ordered list of interactors that should receive the input at a position. Each interactor is then queried as to whether it can handle the event, and the event is delivered to the first one to say yes. A typical picking strategy would select the topmost interactor within whose bounds an event is located. For example, if the centroid of a press event falls on the icon in Figure 1, the icon would receive the event.

To account for ambiguity and uncertainty these processes need to be handled probabilistically – we need to provide a representation of the certainty that an event should be delivered to a given interactor. As shown in Figure 2, probabilistic events are delivered to *all of the interactors* in the candidate selection list instead of just one.

Interpretation and Action

Once an interactor receives an event, it is responsible for responding to that event. Each interactor tracks its *interactive state*, including where it is currently drawn, what configuration it is in (for example, a check box knows if it is checked or unchecked) and where within an interactive input sequence it currently is (for example, the icon in Figure 1 may know that it has received a press event and move into a “drag” state). A finite state machine is the traditional way to represent how an interactor should respond to different input events [17, 21].

It is the responsibility of the interactor to display feedback in response to events (which is often done by updating its visible location on the screen, transparency, *etc.*). At appropriate points in the interaction (*e.g.*, when the icon receives a *release* event), the interactor is responsible for requesting action from the application on behalf of the user (*e.g.*, if the icon is dropped into a folder, requesting a file system update).

In an uncertain world, each interactor also (probabilistically) interprets events. As with the conventional input handling framework, each interactor object is responsible for maintaining its own internal state. To properly handle uncertainty between alternative inputs, interactor state needs to be handled probabilistically. This can be done in an ad-hoc manner (as with the implementation described later in this paper), or using a more structured approach. One possibility is the *Probabilistic Finite State Machine* (PFSM) approach developed in our previous work [10]. A PFSM tracks the current state of an interactor as a PMF across the state space of a finite state machine and updates that state based on probabilistic events.

When an interactor changes state this may imply that an action needs to be performed, such as updating the file system to represent its new location. If we assume that applications should not execute multiple potentially conflicting actions (each implied by different interpretations of the same uncertain input), this leads to a new requirement: At some point, it is necessary to choose between the different possible actions – to resolve the ambiguity.

There is good evidence that pushing off this decision as long as possible is better for the user than making it too soon [8]. To enable this delayed resolution of uncertainty, we introduce a refinement on the notion of actions, breaking them into two categories: *temporary actions* and *final actions*. Temporary actions – which most typically provide feedback – are always fully reversible. For example, if a touch position is uncertain (*e.g.*, overlaps two buttons) we can, and probably should, provide feedback to the user indicating that both possibilities are still being considered by the system. Final actions on the other hand may not be reversible, and may not be compatible with each other. For example, it is not typically safe to ask the application to execute the final action of a "cancel" and "ok" button simultaneously, so we must make a choice between them.

In our framework, the resolution of uncertainty is handled in a lazy fashion – it is pushed off until absolutely required. If an interactor reaches a state in which it must act, it makes a finalization request. As illustrated in Figure 2, finalization requests are handled by a system component called a *mediator*. The mediator may either choose one action, to resolve ambiguity, or override the interactor by deferring mediation if more information is needed. This part of the framework is directly adapted from [14] and the reader in encouraged to refer to [13] for a discussion of interesting mechanisms for interactive mediation.

Probabilistic Input Handling Example

Figure 3 illustrates a specific scenario for the use of probabilistic input. Here touch input is being used to select a tiny square button smaller than the fingertip, packed tightly with two other small buttons. The button on the left in this example is disabled. There is clearly some ambiguity here as the finger is at least in part "over" all three buttons.

To give an overview of how our framework operates, we consider what happens when the finger is first pressed over these buttons. This input is modeled as a probabilistic event. Since we are fairly certain that a touch occurred, a single event (with 100% probability) is generated by the system. The location of that event is represented as a PMF derived from a 2d Gaussian function centered over the touch center with standard deviation of ¼ the width and height of the touch area (illustrated with the shading under

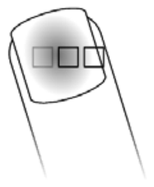


Figure 3: A probabilistic input scenario involving a touch over three very small square buttons. The button on left in this example is disabled.

the finger outline). This PMF is used during dispatch to determine how probable it is that the event is intended for each of the buttons. In this case, the *selection score* is the integration of the touch PMF over the button area. As is intuitively correct from a visual inspection of Figure 3, this *selection* score is higher for the left and center buttons than the right button. Interactors may optionally zero out their selection scores if they can determine that a particular input is definitely unsuitable for them, or may reduce the score if the input is unlikely to be suitable. For example, a disabled interactor would always return a 0 selection score and a button might return 0 for all text inputs.

Based on this, the leftmost button would return a 0 selection score and be eliminated from consideration. The event is then delivered to the two remaining viable candidates. Since the center button had a higher selection score than the right button, it generates a possible action with a higher score than the right button. In this simple example, the mediator selects that action over the other (less likely) alternative. The center button's action is executed and the right button's action is canceled. Other mediation strategies or new information might lead to other outcomes.

HANDLING INPUTS WITH UNCERTAINTY: DETAILS AND IMPLEMENTATION

While the previous section provided an overview of the key differences between conventional and uncertain input, this section explains in more depth how our framework actually handles event *modeling*, *dispatch*, *interpretation* and *action*.

A proof of concept implementation of the ideas presented here has been implemented in C# on top of the .NET framework. Relevant implementation details are discussed within each subsection. Although our implementation is not a full toolkit, it constitutes a working demonstration of the ideas discussed below.

Modeling Probabilistic Events

Each uncertain piece of information in a probabilistic event is modeled as a PMF. These PMFs can be instantiated in one of several forms, abstracted into an API. For example PMFs can be implemented as a table or histogram, can be implemented using an analytical function, or as a Monte Carlo sampling of the underling distribution.

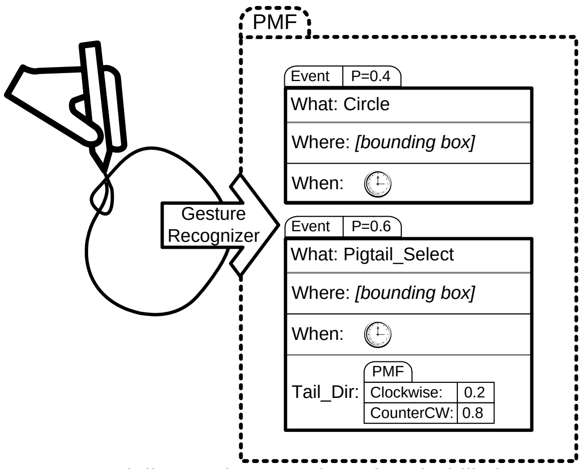


Figure 4: Modeling and generation of probabilistic events. A gesture recognizer generates two event alternatives: a circle with a probability of 0.4 and pigtail selection with probability 0.6

The type of the event (*what* it is) is a special case. Since the event type may determine the detailed structure of the overall event record, we implement the PMF for event type as a collection of separate event objects, one for each type, each with an associated probability. The collection of alternative events and their associated probabilities then serves as the PMF over possible event types. Figure 4 illustrates a sample uncertain event.

Event Dispatch

As described earlier, the goal of event dispatch is to select a candidate set of interactors that may represent the proper recipient(s) of an event, and then deliver that event. Although position or interactor focus are most commonly used, other interesting policies are possible, see [9]. For example, input could be dispatched based on its proximity to a target or whether it surrounds a target. Probabilistic selection should be equally flexible, with the added requirement that events are delivered to *all* interactors that may be plausible recipients.

Selecting which interactor should receive an event

We support a probabilistic notion of dispatch in which events are delivered to all interactors which are candidates for input. Candidacy is determined using a scoring mechanism that considers event properties and interactor state. These scores are provided by querying each interactor. The resulting probabilistic selection list can be seen as a PMF. Past tools have typically not provided structured support for uncertainty about who should receive an event (termed *target ambiguity* [14]).

Conventional dispatch usually determines candidacy based on one of two algorithms focusing on event type (focused dispatch) or position (positional dispatch). Both are handled in the same way in our framework. Each interactor examines the type, position, or any other property of an event to determine a *selection score* between 0 and 1 inclusive. Selection may be based on event types, as well as measures of "overlap," "nearness" or other more radical spatial relationships. By loosening the requirements for "overlap" and using logic and contextual information to determine "nearness", we can enable *inexact* interaction [3]. A nice property of the framework is that more exotic styles of dispatch can be supported simultaneously with more conventional styles because of the uniform way in which uncertain input is delivered.

As a simple example of this process, consider the buttons in Figure 3, which are smaller than the touch area. They are designed to calculate a selection score based on the integration of the location PMF over the button bounds. Because buttons look at the integration over the non-uniform location PMF, if the finger completely covers two or more buttons, the buttons closest to the center of the finger (where probability density is higher) will have the highest score. In contrast, an interactor used in a pen based system might return a high selection score for a circling gesture event which *encloses* it, rather than requiring overlap. Alternatively we might support underlining in a pen system using a combined measure of nearness (without overlap) and parallelism to calculate the selection score.

The result of the selection process is a *candidate list* of interactors with associated probabilities (*selection scores*) indicating the likelihood that they are the intended target of the user's input. These selection scores are normalized across all interactors so that they sum to 1.

Finally, for each interactor, the normalized selection score, and probability that the event actually occurred are multiplied together to determine a final probability to be associated with dispatch of the event to that specific interactor.

Interpretation, Feedback and Action

At this point, it is up to the interactor to interpret the meaning of the event based on its internal state, update that state, and decide whether there is a possible action it might take as a result. As indicated earlier, it is often valuable to provide immediate feedback to expresses a system's current understanding of the user's input – ambiguous or otherwise. As a result, feedback or other temporary, fully reversible actions, are modeled separately from actions that have permanent and/or irreversible consequences. In our framework we only allow interactors to modify their own appearance to avoid conflicting representations of feedback.

The interactor informs the system of any temporary actions that it will take using a *temporary action* object that encapsulates key information about the action useful for later reversing it. Similarly, possible (final) actions are encapsulated in a *possible action* object (also referred to as a final action request), which has an associated probability that this interactor is really in a state compatible with that possible action. This possible action is passed to the system along with a Boolean indicating whether the request must be finalized immediately, or can be handled in a lazy fashion.

Mediation

A mediator's job is to choose between competing (and potentially conflicting) actions. Typically, the mediator will make this choice when it receives a *finalize* request. In that case, the mediator will make one of three choices: (1) it

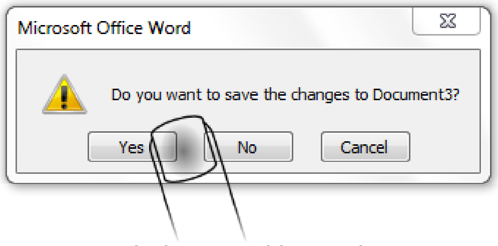


Figure 5: Scenario for an ambiguous button press. Buttons in our framework look at the integral of the location PMF over button area to determine selection.

may select an action (in our implementation simply the most probable one) (2) it may decide that no action is probable enough to execute and cancel all actions (3) it may decide that it needs additional information and request some from the user, deferring mediation until the user responds *e.g.*, while an *N-best list* style mediation dialog is offered to the user. Many other sophisticated interactive mediators are also possible [13]. In some cases, the mediator *may* decide to finalize even without a request to finalize. For example, if there is one possible action that is significantly more likely than any others (*i.e.*, ambiguity is very low), the mediator can choose to automatically finalize that action.

Once a possible action is selected, the associated interactor is notified that the action has been finalized (and can act on it) and any other interactors with possible or temporary action requests are notified that their actions are canceled.

One nice property of the temporary/possible action split is that feedback is provided early, and may be visible up to and through the mediation process. This can help a user to adjust their input to implicitly mediate, or provide the user with needed context during more explicit cases of mediation. When interactors are notified that an action has been canceled, feedback can be removed. Any pending actions (temporary or possible) registered since the last time an action was finalized are canceled. The interactor associated with each action is expected to roll back its state and undo any temporary changes such as displaying feedback.

A Complete Example

To illustrate the complete flow of input under our framework, we use the interface shown in Figure 5. This example adds realism to the example associated with Figure 3. Although both examples are fairly basic, our implementation enables something very powerful: By allowing users to select among very small buttons using a finger, we open the door to mobile applications that can take advantage of increasing screen quality rather than being limited by finger size. Following this we will present a number of additional examples of the interactions enabled by our tool.

1. The user touches the screen and a probabilistic touch event is generated and passed to the dispatcher. As described before, a single event with 100% probability is generated. The position information within the event is uncertain and is provided as a PMF derived from a 2d Gaussian function centered about the touch center with standard deviation equal to the width/height of the touch area divided by 4.
2. Each interactor in the interface is asked for a selection score which indicates whether it should be in the candidate dispatch list. It is obvious in Figure 3 that this includes all three buttons, but note that in the case of Figure 5, this includes the “yes”, “no”, “cancel” and “close” (“X”) buttons, as well as the dialog box title bar. Each interactor computes its selection score by integrating the PMF over the button area. After receiving the selection scores, the dispatcher delivers a touch down event to the ‘yes’ and ‘no’ buttons, as these are the only buttons which have returned nonzero selection scores.
3. Our button class acts as soon as it receives the initial touch (without waiting for a release event), so each button’s tracking of state and interpretation of the input is trivial. The response of the “yes” and “no” buttons is to each emit a possible action and request finalization. Since the interpretation of events is very simple in this example, the incoming event score is simply passed along as the action probability.
4. When the mediator receives the possible actions, it notes the presence of a finalization request and immediately makes a decision among the possible actions. The mediator we implemented selects the action with the highest probability to go forward. In the case of Figure 5, this means that it calls *finalized()* on the “yes” button, passing its action back so that it can respond, and calls *cancel()* on the “no” button. All other buttons are ignored, since they did not submit possible or temporary actions.
5. The selected button notifies the application that it should execute its action. If any other button had displayed any feedback, at this point it would remove that feedback.

This example has the practical effect of making the effective target size larger. This is one of the central ways in which one can make it easier to select a target (“beat” Fitts’ law [1]) and so it is not surprising that many advanced interaction techniques take a similar approach – starting from Sutherland’s earliest work with gravity fields in SketchPad [23] through [3, 15, 2, 16]. Our framework’s approach to identifying candidate targets makes adaptive versions of techniques for beating Fitts’ law essentially the default (having an effect analogous to adaptive area cursors [11] or bubble cursors [6]).

Integrating Conventional Interactors

Our approach allows different interactors taken from a library to be used largely without regard to what other (probabilistic or conventional) interactors are being used in the same interface. In this section we will outline how our framework could integrate conventional interactors, although we have not yet implemented this.

As discussed earlier, ordinarily, a conventional interactor upon receiving input will either discard it (allowing another interactor to use it instead) or act on it in an irreversible fashion. A simple solution to this problem is to embed the conventional interactor in a container that will act as a broker between it and our framework, ensuring that it only ever receives input that it can assume is certain.

To accommodate this, the wrapper container handles selection scoring in a way that matches the typical behavior of conventional interactors: input must fall *inside* the interactors bounds, and non-zero scores are provided only for input types that the interactor has registered interest in. To facilitate this, a developer must specify the type of input the interactor handles.

If an event is then dispatched to the wrapper container, it immediately creates a possible action and submits a request to finalize it to the mediator. If the mediator selects the possible action, the original event is passed on to the conventional interactor by the wrapper.

One special case must be handled for this to work: It is possible that the conventional interactor will discard the event (rather than acting on it). In this case, the proper behavior of the mediator would be to select the next most likely possible action instead. To facilitate this, the mediator does not cancel any actions until *after* the wrapper container notifies it that the conventional interactor did indeed consume the event. With these changes, conventional interactors may be integrated into our framework in a seamless fashion.

Note that like every other aspect of our framework, this approach is pluggable. For example, the wrapper could: support inexact interaction, finalize later in the interaction process, or convert between input event types to make them match the expectations of the conventional interactor.

EXPLORATION OF RESULTS

We provide a set of demonstrations illustrating the flexibility and expressiveness of our framework. These examples help to illustrate the space of interactions that our framework should be able to facilitate. Our examples include new and existing interaction techniques – demonstrating that our framework is expressive enough to create new possibilities and support the state of the art. We show how our framework can support feedback of uncertainty, provide a firm basis for making probabilistically sound decisions, and avoid premature decisions. This helps to avoid brittle dialogs and allows interactions to make use of multiple and richer sources of information about the interaction to resolve ambiguity (such as the shape of a movement that occurs *after* an initial click, touch, or other start of an interaction).

Our validation consists of six case studies. Taken together they demonstrate our framework's ability to address new types of uncertainty, expand on or straightforwardly support prior novel interaction techniques from the research literature, and introduce new interaction techniques. These examples are presented in the subsections below, and the reader is also referred to the accompanying video figure for running demonstrations.

Improving Touch Interaction (Examples 1, 2 and 3)

Pointing activities are an integral part of interaction in today's graphical user interfaces. Perhaps because they are such a basic part of what we do, we have stopped noticing that not all pointing activities are equally straightforward and error free. For example, to save screen space, many windows have very small borders. This is good, except when the user is trying to resize a window and must grab its border (and not something under it or next to it). Similarly, small slider and scrollbar thumbs are often difficult to manipulate (or far from the current locator position). These and other interactions are made even worse in the face of touch input, where the user's finger may be larger than the very targets he or she is attempting to select. For small devices such as mobile phones, the result can be limits on the number of interactive elements that can reasonably be interacted with, despite high screen resolution.

Implementation

We have implemented three examples: (1) smart window resizing (shown in Figure 1); (2) ambiguous and remote sliders (shown in Figure 6); and (3) tiny buttons for touch input (shown in Figure 3). Some key implementation details are discussed here.

All of these examples boil down to a simple question: Which interactive element should be handling the current set of locator-related events? Since these events are touch based, they are modeled as a single probabilistic event record, with a PMF describing the location as an area approximately the size of the user's finger (derived from the 2d Gaussian mentioned earlier). The question of who should handle each event is easily addressed in our framework by modifying the selection scores of interactors. Based on this, each interactor is given an event with a score weight indicating the likelihood that this is the correct interactor to consume that event. At that point, all of the interactors proceed in the identical way: They decide whether to show any feedback, create a possible action (representing what they would do with the event), and pass it to the mediator. This process repeats as each additional touch event arrives, until the mediator decides that ambiguity is low enough to finalize, or an interactor reaches a state in which it must finalize (such as when the user raises his or her finger). Note that mediation always considers the most recent possible actions only, but will cancel all past actions as well when completed.

Factors to consider for selection include: (1) direction of motion for move events (appropriate for smart window resizing, which assigns scores to targets differently for horizontal motion and vertical motion) (2) overlap or nearness (overlap is appropriate for touch, but nearness creates the equivalent of an area cursor [16, 12]). Selection is implemented in the same way for all of the interactors in our examples. However, it would be straightforward to implement special cases such as buttons with dangerous actions that artificially reduce their selection scores to require that they be more unambiguously selected before they can be activated.

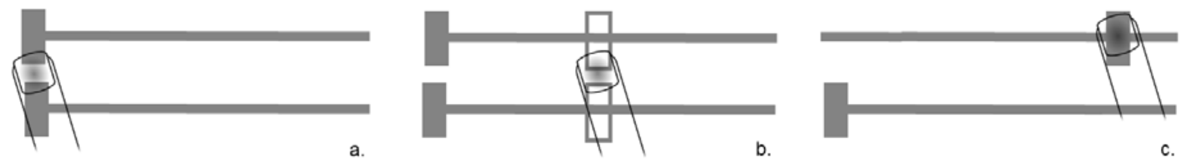


Figure 6: Sliders use temporary actions to provide feedback when a user moves in between two sliders.

Given these decisions, we implemented three sample applications:

1) Smart Window Resizing (Figure 1)

For this interaction horizontal motion should drive the selection score for the window up, and vertical motion should drive it down. Direction of motion is calculated by comparing the location of the current move event to the original press event. The icon's selection score is computed based on the integral of the location PMF over the icon area, multiplied by a factor accounting for its direction.

2) Ambiguous and Remote Sliders (Figure 6)

A key feature of these sliders is interaction at a distance. Thus, the selection score depends on the distance between the current *move* event and each slider on the screen in addition to the direction of motion. As shown in Figure 6b all viable target sliders show feedback.

3) Tiny buttons for touch input (Figure 3)

Buttons integrate the location PMF over the button area to calculate a selection score. Details on this application of our framework are provided in the two concrete examples discussed earlier in the text.

Discussion

We have discussed the implementation of three demonstration applications. In all three cases, there is ambiguity in determining what the user intends to interact with. This is further complicated by the wish to interact at a distance (sliders) or support inexact interaction (buttons). An important property of our framework is the fact that all of these different styles of interaction are enabled simply by varying the method for determining the selection probability. The interactors involved do not directly communicate. Rather, our framework helps to handle the process of deciding among them.

In using these example applications, we found it easy to adapt to the flexibility of the interface. For example, in the case of small buttons, it is very easy to pick a spot near one button but not near others. This effectively increases target size for that button.

Clearly, our framework has the potential to enable entirely new forms of interaction. Interactors could adjust their responses based on how likely they are to be pressed (given a users previous actions), based on the severity of any dangerous consequences they might have, or other forms of context [3]. By enabling feedback without committing to action, we can improve the user's ability to make corrections. This may help mitigate a key limitation of touch interaction – the inability to hover over the interface to position a locator without affecting what happens in the interface.

Just as clearly, not all of these may be usable. As with the GUI interfaces of the past, the interaction techniques made possible by our framework will need to be studied and refined progressively over time to match the user's needs as fluently as windows, menus, icons, and so on.

Smarter text entry (Examples 4 and 5)

The previous examples focused on touch input, but our framework is equally appropriate for other types of input. Consider the problem of entering text into a form.

Although the text a user is typing is certain, *where* this text should go may be ambiguous. Many forms don't respond when a user starts typing text without having first selected a field, and those that do respond often simply select the topmost text box. Things get even more complicated when a speech interface is in use, especially given that speech is often used because of physical or visual impairments that eliminate the ability to use a pointer.

Our framework can address this problem with text fields that return selection scores based on what content they expect to receive, show feedback, and make final action requests when they think their text fields are complete. If several text boxes can receive input, our system easily allows these fields to provide feedback and allow the user to disambiguate.

Implementation

We have implemented two examples – smart text delivery and speech text entry. Both examples deliver text to a form with several different fields: name, phone number, email, and URL. We built a text box interactor that matches input against a regular expression, which we re-used for each field. During the *selection phase*, a text box returns a selection score based on the match between the incoming character and its regular expression. For example, if a text box has already been filled out it returns a selection score of 0, if a phone number field sees an incoming alphabetic character, it also returns 0, whereas if a phone number field sees an incoming number it returns a selection score of 1.

When a text box receives an event, it produces a *temporary action* and shows the text it's seen so far in gray. Text boxes send a finalize request if a user clicks on the textbox (to explicitly disambiguate which field should receive the input), or if the text in the textbox matches some finalization criteria expressed by a regular expression (for instance ends in ".com" in the case of a URL field).

4) Smart Text Delivery

When a user types text without having first selected a text-box, all text boxes which have not yet been filled out and that can receive input (i.e. which return a selection score > 0) show the typed text in gray. A user can then continue typing or select the correct textbox. If a user continues typing, his text may match some finalization criterion for a text box, causing that text box to send a finalization request with a possible action score of 1 and to become selected if no other textboxes send similar finalization requests.

5) Speech Text Entry

Speech recognition systems often include confidence scores for each recognized utterance. Previous research has identified and evaluated patterns of entry and correction in speech recognition systems [12], and examined how to use confidence scores to improve interaction [5]. Our framework naturally incorporates these confidence scores of recognition systems into probabilistic events, and supports the improvements studied and recommended in previous work.

We used the Windows Speech API to implement a speech-based probabilistic dispatcher on top of the text demo described above. A speech recognition event often contains multiple uncertain interpretations, such as “2” and “q”. Our speech dispatcher would turn these alternate interpretations into two alternate text entry events which it would send to each of the interactors. At an implementation level, those events are handled identically to the single character events described in example 4 above, with the caveat that an interactor will only display temporary feedback about the *most likely* event currently in consideration. From an interactive perspective the result is that a name field might show one possible interpretation of the user’s speech (such as “q”), while a phone number field might show the other (such as “2”). The user could then continue speaking or select the correct textbox.

Discussion

While it would be possible to implement an intelligent form entry system using conventional interactors, that would typically require a complicated ad-hoc solution on top of the conventional event-based model. In contrast, we were able to implement our demo by writing a single reusable probabilistic text box subclass in 130 lines of code.

The extensions to our existing text box class to add support for voice was less than 30 lines total. This shows that our framework can handle multiple interpretations of alternative events with little to no extra work on behalf of the developers writing new interactors.

Improved GUI Pointing for the Motor Impaired (Example 6)

Individuals with motor impairments may struggle greatly when using conventional GUI input devices such as a mouse and keyboard. For example, in a prior analysis of real-world pointing data by six individuals with motor impairments [10], we found a mean double click speed of 1.3 seconds [SD .3 seconds], a mean slip distance (distance traveled between the press and release of a mouse button) of almost 6 pixels [SD 20 pixels], and high rates of missed clicks, direction changes, excess distance traveled, and so on. All of these measures tended to have a high variance both within users (over time) and across users. One interpretation of this analysis is that this input is unpredictable, or *uncertain*. That is to say, the actual location, time, and even button used when a user clicks is at best an approximation of their actual intent. We hypothesized that by treating their input in a way that handles its true uncertainty well, we could increase the accuracy of user interactions.

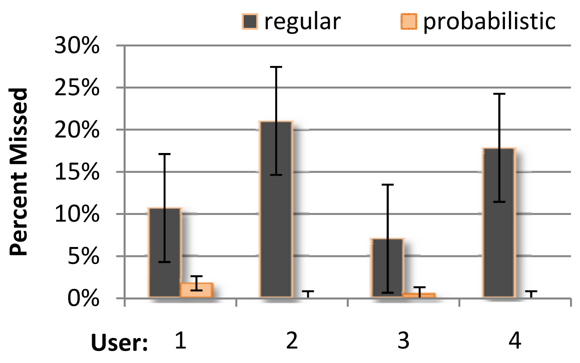


Figure 7: Reduction in errors for motor-impaired users when probabilistic input is used

Implementation

To test the validity of our hypothesis, we conducted an experimental analysis of recorded real-world interaction data of motor-impaired participants, derived from [10]. We selected a total of 3,614 click events from hour-long recordings of four (male) participants using a desktop computer. We then pruned these down to 419 clicks representative of real-world selection scenarios with known targets (many of the clicks in our data were from games which had abnormally large target sizes). For each click, we coded the event based on whether the user clicked on their intended target.

Next, we simulated the same click using our framework. For this simulation, we modeled the location of the click probabilistically using a 2d Gaussian distribution of 30x30 pixels (providing an expanding selection area analogous to bubble cursors [6]). Each potential target returned a score based solely on overlap with the probabilistic location of the event, and mediation selected the interactor with the highest score. The result of each simulation was coded along the same metric as the original interaction: Was the intended target selected?

As shown in Figure 7, in the original data set, users missed their target 14% of the time, on average (SD=6%). Our simulated interaction was *incorrect in only two cases.* This is a worst-case result, meaning that a more sophisticated model of location ambiguity or a user who learned how to leverage our solution would likely have led to even better results.

Discussion

This example demonstrates a novel application of the overall concept of uncertainty. More generally, many situations where the user's intent may not match their actions might be considered an opportunity to apply our framework.

Summary

The six examples discussed in this section illustrate how relatively straightforward applications of our tool can enable novel interaction techniques, or simplify (and expand on) the implementation of interaction techniques that had previously been applied in an ad-hoc fashion.

FUTURE WORK

Our framework presents a new way of thinking about and handling user input. As such, it opens a wide range of possible directions for future work, some of which we touched upon in our presentation thus far.

Specifically, we mentioned that during the *interpretation phase*, interactors could probabilistically track their state based on probabilistic inputs and act accordingly. The details of this are fairly complex and are a topic we plan to cover in future work. Finally, in future work we plan to integrate our contributions into a toolkit which developers can use to handle inputs with uncertainty.

CONCLUSION

The advent of new input technologies introduces a new type of input: input with uncertainty, which conventional input frameworks deal with poorly. We presented a framework for the robust and flexible handling of inputs with uncertainty as an extension of the conventional event-based input handling framework, and showed how our framework supports existing interaction improvements, enables new interactions, and can be used to create techniques improving user experience for people with motor impairments. Our paper highlights the importance of properly dealing with uncertain input and presents a robust and flexible framework for handling inputs with uncertainty.

ACKNOWLEDGMENTS

This work was funded in part by grants IIS-0713509, IIS-0803733, and IIS-0840766 from the National Science Foundation and a grant from the Intel Research Council. This project was also supported by a National Science Foundation Graduate Research Fellowship and an ARCS Foundation Fellowship.

REFERENCES

1. Balakrishnan, R. "Beating" Fitts' law: virtual enhancements for pointing facilitation. IJHCS **61**(6), 2004, 857-874.
2. Bederson, B. B. Fisheye menus. In *Proc. UIST '00*, 217-225.
3. Blanch, R., Guiard, Y., and Beaudouin-Lafon, M. Semantic pointing: improving target acquisition with control-display ratio adaptation. In *Proc. CHI '04*. 519-526.
4. Cao, X., Wilson, A.D., Balakrishnan, R., Hinckley, K., and Hudson, S., Shapetouch: Leveraging contact shape on interactive surfaces. In *Proc. TABLETOP '08* 129-136.
5. Feng, J. and Sears, A. Using confidence scores to improve hands-free speech based navigation in continuous dictation systems. *TOCHI* **11(**4):329-356, 2004.
6. Grossman, T. and Balakrishnan, R. The bubble cursor: enhancing target acquisition by dynamic resizing of the cursor's activation area. In *Proc. CHI '05*, 281-290.
7. Hong, J., Landay, J., Long, A.C., and Mankoff, J., Sketch recognizers from the end-user's the designer's and the programmer's perspective. In *Proc. AAAI Spring Symposium on Sketch Understanding '02*, 73-77.
8. Hudson, S. E., Mankoff, J., and Smith, I. Extensible input handling in the subArctic toolkit. In *Proc. CHI '05*, 381-390.
9. Hudson, S. E. and Newell, G. L. Probabilistic state machines: dialog management for inputs with uncertainty. In *Proc. UIST '92,* 199-208.
10. Hurst, A., Mankoff, J., and Hudson, S. E. Understanding pointing problems in real world computing environments. In *Proc. ASSESTS '08*, 43 – 50.
11. Kabbash, P., Buxton, W., The "prince" technique: Fitts' law and selection using area cursors. *In Proc. CHI '95*, 273-279.
12. Karat, C., Halverson, C., Horn, D., and Karat, J. Patterns of entry and correction in large vocabulary continuous speech recognition systems. *In Proc. CHI '99*, 568-575.
13. Mankoff, J., Hudson, S. E., and Abowd, G. D. Interaction techniques for ambiguity resolution in recognition-based interfaces. In *Proc. UIST '00*, 11 – 20.
14. Mankoff, J., Hudson, S. E., and Abowd, G. D. Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. In *Proc. CHI '00*, 368-375.
15. McGuffin, M. and Balakrishnan, R. Acquisition of expanding targets. In *Proc. CHI '02*, 2002, 57-64.
16. Moscovich, T. Contact area interaction with sliding widgets. In *Proc. UIST '09*, 13-22.
17. Newman, W.M., A system for interactive graphical programming. In *Proc. AFIPS Spring Joint Computer Conference '68*, 47-54.
18. Oviatt, S. Mutual disambiguation of recognition errors in a multimodal architecture. In *Proc.CHI '99*, 576-583.
19. Oviatt, S. Ten myths of multimodal interaction. *CACM* **42**(11):74 – 81, 1999.
20. Sutherland, I. E. Sketchpad, A Man-Machine Graphical Communication System. Doctoral Thesis. Massachusetts Institute of Technology, 1963.
21. Wasserman, A. I. Extending State Transition Diagrams for the Specification of Human-Computer Interaction. *IEEE Trans. Softw. Eng.* **11**(8):699-713, 1985.