quence, this sequence is concatenated onto the current action request sequence being tracked.

When a child interactor returns several different action request sequences *c1*, *c2…,cn* (representing *n* different interpretations), the current action request sequence must now branch into *n* new action request sequences (Figure 5.4 step 4). Each of these branches now continues along with the dispatch process independently of other branches, accumulating action requests accordingly. Figure 5.4 illustrates how action requests are accumulated for a single (interface alternative, event sample) pair. The JULIA toolkit provides several extensible container implementations, which we describe below.

The most basic container dispatches input to all children in depth-first order (after first checking for in focus elements). Many of the examples described in this thesis use this container.

Another container type is a scrollable container, containing a single immediate child whose contents may be scrolled. This is the container used in the example in Figure 5.3. When responding to touch input, this scroll container returns action request sequences generated by child elements, as well as a possible sequence where the scroll container itself handles the event (scrolling the contents). The likelihood of a container scrolling its contents relative to not scrolling and instead allowing children to handle events is based off of a simple heuristic (amount of vertical motion), but more sophisticated algorithms are possible.

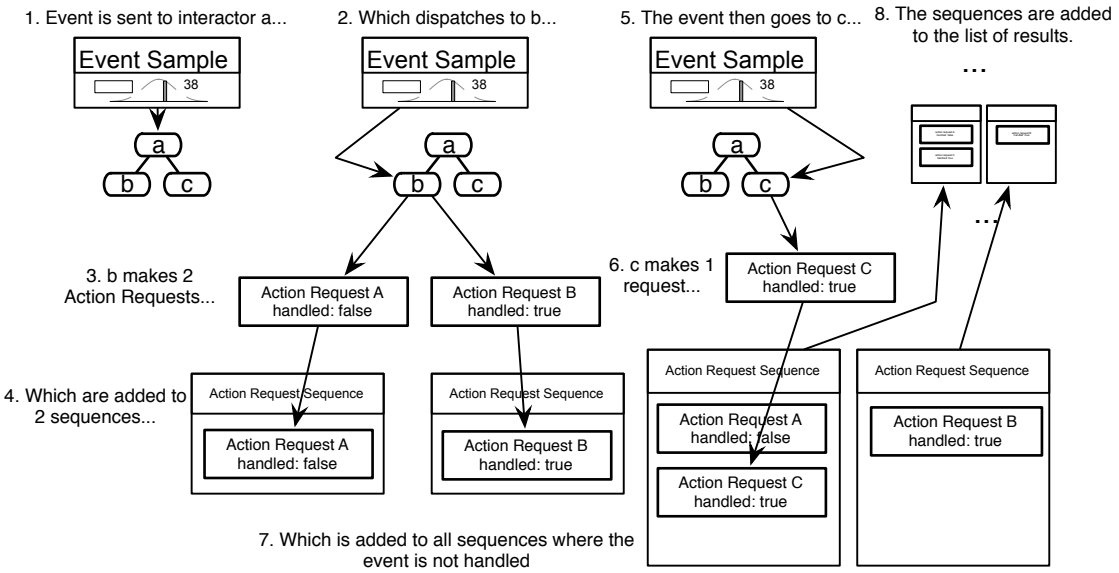


Figure 5.4: Diagram illustrating the sequence of steps used to dispatch events and accumulate action requests.

A final container type implemented in the JULIA toolkit is a container that dispatches an event to its immediate children with identical probability. In other words, it dispatches an event as if all children were at the same 'level'. Every immediate child is given a chance to handle the event, rather than prioritizing the first child. This is especially useful when building a canvas where multiple possible interpretations are possible, *e.g.* a user may be gesturing or drawing a line. This container allows each interactor (the gesture recognizer and line) to be considered independently of one another.

The three container types described allow for a large range of interactive capabilities, but of course many other containers implementing different dispatch policies are also possible, and may be implemented by extending one of the three containers provided.

5.4 Computing Likelihood of Action Requests

After dispatch finishes, each interface alternative yields zero or more action request sequences, each of which represents an alternate possible new state. The likelihood of each action request sequence is multiplied by the likelihood of the event sample and interface sample that it originated from:

*L(ActionRequestSeq') = L(ActionRequestSeq) * L(EventSample) * L(InterfaceAlternative)*

Where *L(x)* represents the likelihood of X, and *ActionRequestSeq*' represents the new action request sequence.

This in turn yields a likelihood distribution of possible state update operations, which is then sent to a mediation process (described in Chapter 7) to determine which state update operations to execute, reject, or defer.

5.5 Example

For example, consider the interface presented in Figure 5.5. This interface has two buttons (A, B), and a set of diagnostic text fields in the corner communicating the number of touches, touch locations, touch state. For the sake of example, assume each is a separate interactor. Figure 5.5A shows the interactor tree. Consider when one event sample overlapping button A gets dispatched. The event is first dispatched to the diagnostics container, which in turn dispatches to each diagnostic. Each diagnostic returns a feedback request, but does not handle the event. The last diagnostic element, however, does return two alternatives, one with the update text "foo" and the other with the update text "bar" the other representing that the user intends to gesture. The diagnostics container view then appropriately zips up these requests (Figure 5.5C) and returns the list of requests to the root container. The root container then sends the down event to button A, which re-

turns a feedback request and DOES handle the event, in both conditions. Button B does not get to see the event. The resulting request sequences contain feedback requests to update the diagnostic fields and change the state of the button.

One final difference to note in event handling is that dispatch of every input event generates some state update operation for an interface alternative. This means that even if no interactors in an interface respond to an event, an update operation representing a self transition (with no variable update) is generated for the given alternative, event pair. This is necessary to avoid killing of interface alternatives that do not respond to a particular event.

5.6 Summary

The dispatch process in the JULIA toolkit has several components which enable the probabilistic input architecture to track interface distribution. First, it tracks a distribution of possible interface update operations while still ensuring that each individual update operation (or sequence of updates) is entirely deterministic. Second, instead of immediately executing update operations, it aggregates all operations and their likelihoods. Finally, all update operations (action request sequences) are independent of one another, ensuring that no alternate interface state affects the state of another interface alternative. In combination, these components allow for proper tracking of possible interface update operations.

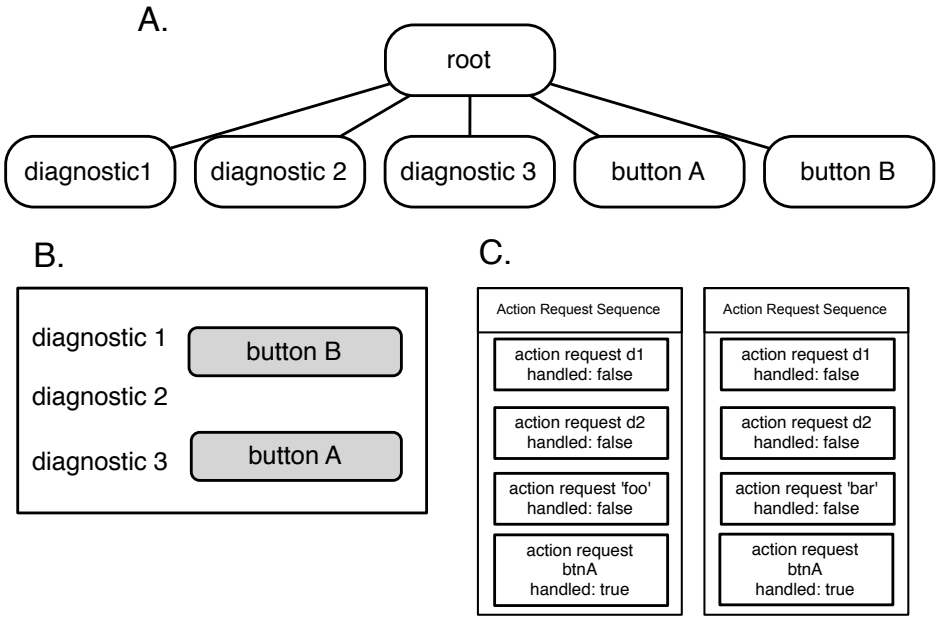


Figure 5.5 Example of dispatch process. A: interactor hierarchy of interface presented in B. C: resulting Action Requests from dispatching an event sample over button B.

Left out of this description was regarding how update operations are actually generated. In other words, how do interactors determine what operations they wish to execute? In the next section I will describe the probabilistic state machine abstraction used in JULIA to simplify interactor development.

6 DESCRIBING INTERACTORS USING PROBABILISTIC STATE MACHINES

When an interactor receives an input event, it is responsible for replying with an action request containing state update operations that it may want to execute. The JULIA toolkit provides a convenient mechanism for tracking state and sending update requests—probabilistic state machines.

6.1 Background

Deterministic finite state machines are a convenient abstraction for responding to input events and tracking interactive state in conventional interfaces. The specific notion and notation of deterministic automata was first introduced by McCulloch and Pitts in 1943 (McCulloch & Pitts 1943) and applied to interaction by Newman (Newman 1968), along with Wasserman (Wasserman 1985). In many cases, the deterministic automata cannot perform sufficiently complex computation to describe the behavior of an interactor. Therefore, machine description languages similar to Augmented Transition Networks (ATNs) (Woods 1970) are used. Though Augmented Transition Networks were introduced to parse natural language, two features make them convenient for describing interfaces. First, transitions in ATNs are conditioned not only on input state, but also on the output of other finite automata (the transitions are recursive). Second, ATNs may read to and write from a register. These two properties, in addition to making the language ATNs describe Turing complete, making them very convenient for specifying interactor behavior.

Many formal languages for describing interactor behavior have spun off from the basic ATN. For example, in (Abowd & Dix 1994), Abowd distinguishes between *input events* which signify an event that should be responded to (*e.g.* a mouse click) from *status* which communicate the change of a variable (*e.g.* mouse position). Abowd then presents a language for specifying the when to change variable constraints in response to input events. Jacob *et al.*(Jacob *et al*. 1999) present a similar architecture where constraints are updated in response to changes in a state machine. In (Abowd 1992), Abowd provides a full overview of formal languages for describing interactor and interface behavior.

While many of the proposed models simplify the description of interactors (and in some cases interfaces) in many scenarios, few have been broadly adopted in commercial user interface toolkits. While the true cause may never be known, one can speculate that mo-

mentum plays a large role. Therefore, the description of probabilistic state machines used in the JULIA toolkit was designed to be as similar to ATNs as possible.

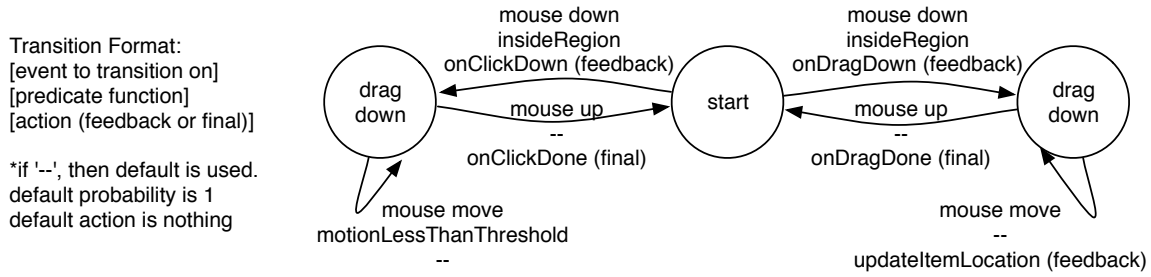
6.2 State Machine Description

The state machines used in the JULIA toolkit are a probabilistic version of ATNs. As with ATNs, transitions are conditioned not on just on input properties but also on the result of functions called predicate functions. Additionally, transition functions may access interactor or interface properties. The JULIA toolkit also introduced a few probabilistic augmentations. First, multiple transitions may exist for identical inputs. When multiple transitions are specified for the same input, multiple action requests (representing alternate representations) are generated for each transition. Second, transition predicates return a score indicating the likelihood that this transition should be taken given the input and current interface state. Third, transitions contain update functions which indicate how program or interface state would be updated in response to the transition. These update functions are then turned into action requests. In the JULIA toolkit, developers specify the state machine for an interactor as a JSON object. Figure 6.1 provides an example. As will be seen shortly, state machine specification is simple and allows for the construction of complex features.

One of the differences between a traditional ATN and the probabilistic machine is that, like a nondeterministic state machine, a single input may lead to multiple resulting transitions. For example, for the state machine in Figure 6.1, several mouse down transitions emanate from the start state. When a mouse down event occurs over an interactor that is in the start state, two transitions are valid: one where the interactor goes to the click_down state, one where the interactor goes to the drag_down state. In this case, the state machine generates two action requests representing two alternate interpretations. In this case the likelihood for both alternatives will be the same, as `is_in_region` will return the same likelihood (recall that predicate functions return the likelihood of taking a transition). However, if the predicate function were different, the likelihood of alternatives may change.

A second difference is the presence of predicate functions. A transition predicate is a condition that must be met for a state machine to take a transition. It is represented as a JavaScript function which returns a number between 0 and 1, representing the likelihood that a transition should be taken. For example, in the state machine in Figure XYZ, the `is_in_region` predicate returns 1 if the event is within the interactor region, and 0 otherwise. The predicate function may inspect properties of the interactor or interface alternative in which the request belongs. In summary, predicate functions are a convenient way to determine whether to take transitions. When combined with nondeterminism and likelihood weights, they can become very powerful, enabling features

such as adapting interactor behavior based on past user actions as well as appropriately responding to recognition likelihoods.




```
{
  start: [
   {
     to_state: "drag_down",
     input_type: "mousedown",
     predicate_function: is_in_region,
     feedback_function: on_drag_down_feedback
   },
   {
     to_state: "click_down",
     input_type: "mousedown",
     predicate_function: is_in_region,
     feedback_function: on_click_down_feedback
   }
  ],
  drag_down: [
   {
     to_state: "drag_down",
     input_type: "mousemove",
     predicate_function: return_true,
     feedback_function: update_item_location
   },
   {
     to_state: "start",
     input_type: "mouseup",
     predicate_function: return_true,
     action_function: on_drag_done
   }
  ],
  click_down: [
   {
     to_state: "start",
     input_type: "mouseup",
     predicate_function: check_motion_less_than_threshold,
     feedback_function: empty
   }
  ]
}
```

Figure 6.1 Sample description for an interactor that is both draggable and clickable. Top: graphical depiction. Bottom: description using JSON format.

A third difference between ATNs and probabilistic state machines is the specification of feedback and final action functions. These feedback and final action functions later get encapsulated into feedback and final action requests. As a reminder, action requests represent computation to update the interface and possible program state. Some action requests update the state of just the user interface, however other action requests may update program state outside of the interface. Action requests that only update interactor state are called *feedback requests*, requests that also update program state are called *final requests*. A transition may contain one feedback function and/or one final function. A transition may also contain neither an action nor feedback function. In this case, only the current state machine state of the interactor is updated when the transition is taken.

6.2.1 A More Complex Probabilistic State Machine Description

Figure 6.2 demonstrates the state machine for a line brush: a brush that can draw either horizontal, vertical, or unconstrained lines. The likelihood of horizontal and vertical lines changes based on the amount of vertical/horizontal motion when a user is dragging. This is the state machine for the line brush in the graphical object editor application presented in section 10.3.2.

This example has two interesting features. First, the start state has several transitions on the mouse down event, each of which represents a different interpretation (line, horizontal line, vertical line). Second, the likelihood of the horizontal and vertical states change based how much horizontal/vertical motion the user is exhibiting. These likelihoods are specified by the functions probabilityIsHorizontal and probabilityIsVertical. For example, if a user is moving their mouse vertically, the likelihood of the horizontal state decreases. As with the scroll view example described in previous sections, this likelihood is implement-

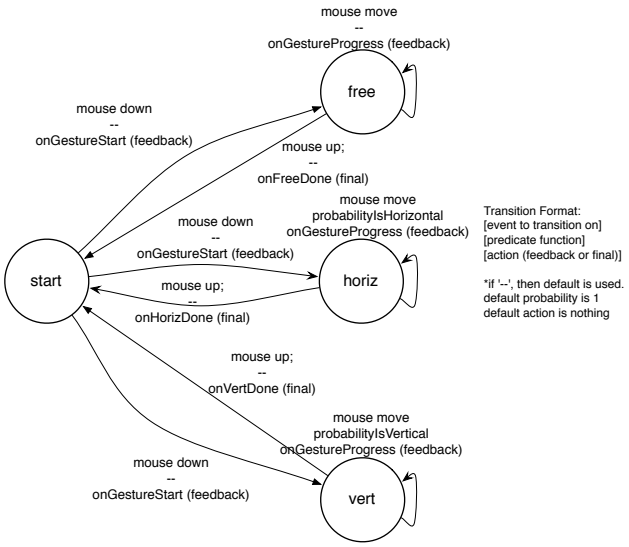


Figure 6.2: Example of state machine description for a line brush. Feedback and final actions are not labeled for sake of clarity. The 'free' state represents an unconstratined line.

ed using a simple heuristic (amount of vertical/horizontal motion), however more complex models trained on user behavior (e.g. SVMs, regression) are also possible.

Now that I've provided a description of how state machine is specified, I will explain what goes on under the covers to actually implement such a state machine using this example.

6.3 State Machine Operation

When an interactor described by a probabilistic state machine receives an input event sample, it examines all outgoing transitions from its current state that match the input event. For each transition that should be taken, the probabilistic state machine generates an action request for every feedback and final action defined. Action requests are weighted according to the likelihood value returned by the predicate function specified. If a predicate function returns 0, no action request is added. As a reminder, these action requests are added to action request sequences, whose likelihoods are later multiplied by event sample and interface sample likelihood, as described in Section 5.4. When an action request is added to an action request sequence, the likelihood of the overall sequence is multiplied by the likelihood of the action request being added.

The generated action requests represent a future computation to update the state of the interface (this includes updating the interactor state as well). If accepted, the action request first updates the current state of the interactor, then executes the update functions provided in the feedback or final actions. If no function is specified, only the state of the interactor is updated.

6.3.1 An Example

In the example FSM for the line brush described earlier, consider what happens after a single mouse down event is sent. We are using a mouse event with a single sample here to focus on the alternatives generated by the nondeterministic state machine, and to demonstrate the probabilistic toolkit is valuable even for deterministic inputs such as the mouse. The line brush is initially in the start state.

All transitions going from the start state are examined, and all 3 transitions (free, horiz, vert) have their predicate function return a probability value larger than 0. For each of these transitions, a different action request is generated, with the probability specified by the predicate function. Assume that the transitions to horiz, vert, and free all return 1. In response, the state machine creates three action requests, each with a weight of 1 (remember, the likelihoods of the event sample and interface sample will be multiplied in after the requests have been returned to the dispatcher). Note that the sum of likelihoods is greater than one; these likelihoods will be normalized later. These three requests are all returned by the interactor to its parent. The requests go up the interface tree until

being sent to the dispatcher, which multiplies in event sample and interface sample likelihoods. Because all three requests are feedback requests, all requests are accepted. There are now three alternate interfaces, where the line brush is in the free, horiz, and vert state. All interfaces have the same likelihood. Subsequent mouse move events will serve to adjust the probabilities: vertical motions will cause horiz to be less likely, and vice versa. The likelihood of the 'free' state will not change much.

6.4 Incorporating User Behavior Models *via* Probabilistic Transitions

As an example of the power of probabilistic transitions, I will now describe how the mechanism of probabilistic transitions can be used to easily incorporate a predictive model of user behavior into probabilistic user interfaces. User behavior patterns are a gold mine of information that can be used to not only predict what a user is likely to do, but also perform educated guesses about user intention when their input is ambiguous. For example, if a user presses between the "File" and "Edit" buttons in the middle of working on a document, and the system knows that the user very frequently presses the Edit button in this scenario, the system should be able to guess that a user wishes to press edit, and select this option. Such predictive models are used in systems such as speech recognition and touch typing, however they have yet to move to interfaces more generally because there is no good mechanism for integrating such models. The JULIA toolkit provides a mechanism for integrating behavioral models to determine intended actions, this is implemented using the probabilistic transition mechanism.

The *user behavior model* has two methods of note: `recordTransition(interactor_id, transition_id)` to record that a particular transition for some interactor has been taken, and `likelihoodForTransition(interactor_id, transition_id)` that returns the likelihood that the user's next action is to take the given transition for a given interactor.

Behavior models can vary widely in complexity. The simplest (and default) model returns the same value regardless of input. A slightly more interesting model is to record the last transition executed and increase the likelihood of repeating the same transition. For example, if a user executes a vertical swipe gesture, this model would guess his next gesture is more likely to again be a vertical swiping gesture as opposed to any other action. An even more interesting model would be to use a markov chain based on recorded behavior. Note that here I used the word "action" instead of transition. Really what this behavior model is doing is recording the likelihood of user actions (execute gesture A, press button B). Since some actions do not always correspond to a single transition in a state machine (perhaps several are required), a little extra logic is needed to specify when multiple transitions are equivalent to the same action.

To incorporate behavioral models into the JULIA toolkit, all that is required is to augment the probabilistic state machine mechanism so that it multiplies the likelihood returned from the predicate function for a transition with the likelihood of taking that transition from the behavior model. As a proof of concept, the JULIA toolkit uses the two simple models presented above (uniform distribution and increasing the likelihood of the most recent transition) to provide a very basic form of incorporating user behavior models. Several demos presented later use these behavioral models to adjust the likelihood of actions (see Section 10.1.9).

Note that the models presented use a simplifying assumption: they assume that the likelihood of a user taking a transition from the model is independent of the input event. Of course, more complex models may be integrated. The beauty of the approach used here is that the models may be arbitrarily complex. Developers may integrate arbitrarily complex behavioral models without needing to change any interface code—all that is needed is to change the behavior model. This example highlights the deep power of probabilistic transitions. It allows us to incorporate behavioral models at the user interface toolkit level, something that has never been done before.

6.5 Summary

Probabilistic state machines are a simple way of describing powerful interactor behavior that has yet to be seen in existing interfaces in such a general form. As was demonstrated in Sections 6.2.1 and 6.4, this abstraction enables behaviors such as tracking alternate input interpretation and incorporating Bayesian inference *via* user behavior models. The simple examples of the clickable/draggable buttons and line brush were just the tip of the iceberg: such probabilistic state machines can be used to implement gesture recognizers, complex inference, and more. The examples in sections 10.2 and 10.3 illustrate further capabilities that probabilistic state machines enable.

The next chapter describes how the action request sequences returned by the interactors and dispatch process are examined to determine which actions to execute (if any), and what happens when input is ambiguous. This process is called mediation.

7 MEDIATION

Figure 7.1: Overview of mediation phase to decide which actions should be executed, rejected, or deferred for later.

In Chapter 5 I showed how the JULIA toolkit dispatches a collection of input event samples to a collection of interface alternatives and obtains a list of actions. The next step is for the JULIA toolkit is to decide which actions, if any, to take. In a conventional toolkit, this step does not even exist because at most one action will be selected and it is simply executed. However, here we dispatch events to many potential interfaces, amassing a collection of possible actions and their likelihoods. The reason for this is because we do not know the exact value for the input nor the exact state that the user intending for the interface to be in before the input arrived. Therefore, we are unsure of the exact action. When all interactors have had their say, the JULIA toolkit needs to somehow figure out what should happen, and update program state accordingly.

Updating program and interface state will be covered in the next chapter, this chapter focuses only on mediation: the process of deciding which operations to execute. Mediation is performed by a Mediator object which takes in a list of possible action request sequences (output from the dispatch phase) and outputs a list of mediation results: a collection of new action request sequences with a decision to either accept, reject, or defer each action. This component is entirely modular, meaning different mediation algorithms can be used. The mediation algorithms and strategies build off of the original concepts of mediation provided by Mankoff in (Mankoff, 2001). There are two steps to the mediation process: aggregation of actions, and action resolution.

7.1 Action Aggregation

In many cases, the dispatch process will result in multiple action request sequences that represent similar or identical update operations. For example, when 100 event samples (located at slightly different locations) are all sent to a button, the button may return 100

action requests that change the button's state to down and update the button's appearance to reflect that it is depressed. In some cases, the update operation corresponding to the transition may update interface state according to properties in the event sample (for example, moving an item according to the location of the event sample), but in this example the update operation only served to change the button's background color. Therefore, all action request sequences would, after update, result in the same interface in the end. These action requests represent requests that can be combined into a single request whose likelihood is the sum of all 100 original requests.

To identify action requests that can be aggregated, the mediator examines the update code in the action request to determine whether two action requests would, after update, result in sufficiently similar resulting interfaces. For action requests stemming from probabilistic state machines, two action requests are not sufficiently similar if they do not result in the same state machine state. The mediator also analyzes an action request's update code to determine whether properties of the input event are used. Action requests whose code is identical, that do not use event properties, and whose code does not result in different interactor states, may be aggregated. If two action requests have identical code but use an input event's properties, then two action requests may be aggregated only if their event samples are equal. Action requests are aggregated by adding likelihoods (we can add samples because event samples are assumed to be independent of one another), all other properties remain the same. Moreover, the input event attached to each action request does not change when two requests are aggregated—the input event from the first action request is used. Right now input events for similar requests are either not used or identical, so this naïve removal of input events when aggre-

Callback	Target	Likelihood
OnPress	A	0.17
OnPress	A	0.17
OnPress	A	0.17
OnPress	A	0.17
OnPress	B	0.17
OnPress	B	0.17

Callback	Target	Likelihood
OnPress	A	0.67
OnPress	B	0.33

Figure 7.2 Example Action Request aggregation from example in Figure 1.1. For the sake of simplicity, we assuming that both A and B are in the start state with likelihood 1. Top: six touch down events are sent to interactors. Middle: action samples are generated as a result of state machine transitions. Each sample includes information about relevant variables (locations of event), callback, and target. Bottom: Action Requests are aggregated according to target and callback.

gating action requests is sufficient. More sophisticated input event aggregation methods are possible, and are left for future work. Fortunately, action aggregation is implemented *via* a pluggable abstraction that can be replaced with other, more nuanced aggregation methods. For example, an aggregation method could aggregate all action requests that yield the same end state and update the event being operated on to be the mean value of all sample probabilistic events in candidate action request sequences. This would have the effect of generating a 'mean action request' across a list of action request sequences.

Figure 7.2 illustrates the action sample aggregation process for a simple button press. In this example, assume that both buttons are in the start state with likelihood 1, and thus that each interactor has only one sample representing its state distribution. If each interactor had more samples, then the six alternate events could have caused more action requests to spawn (since every event gets sent to every state sample), and then the example would not fit comfortably on the page. Additionally, assume that only six touch down event samples are generated from a touch. Four of the touch down events overlap Button A, which would cause four separate transitions from the start to down state in a button (a separate transition for each of the four event samples that overlap Button A). Each transition spawns an action request to handle the OnPress update function. We will further assume that the OnPress action does not actually use any properties of the input event. Similarly, two of the touch down events overlap Button B, causing two transitions. Once again, each of these two transitions spawns an action request. A total of six action requests get sent to the mediation subsystem. For the sake of simplicity, assume that all event and state samples have equal likelihood. Since all requests originate from the same transition, two resulting requests are aggregated: one for Button A and one for Button B, with A being the more likely request.

7.2 Action Resolution

Since these aggregated actions cannot all be executed, the mediation system must now decide to accept zero or more aggregated actions. All remaining actions are either rejected, or, if the system cannot yet make a final decision, deferred. As we will discuss in the next section, when an action is deferred, the mediator typically shows feedback to seek more information from the user. As with action aggregation, mediation algorithms are pluggable. Developers can use one of a library of provided mediators, or make their own. In the next paragraph I will describe the default mediation heuristic used in my examples.

The default mediation algorithm implemented in the current version of the JULIA toolkit runs as follows: First, the algorithm rejects requests below some developer-specified minimum likelihood. Next, the algorithm determines the likelihood of the most probable final action. It then considers all actions requests with likelihood within a developer specified delta of that maximum. I will refer to this region of within a small delta of the maximum as the "equivalence region". If there is one final action within the equivalence region, that

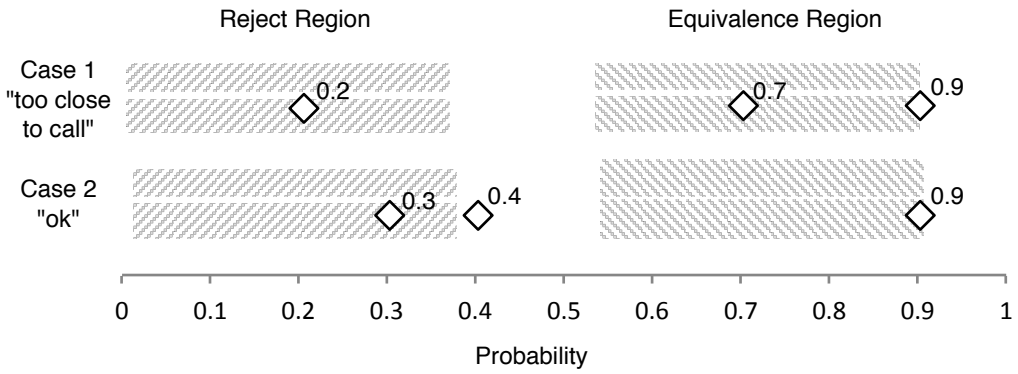


Figure 7.3: Given a set of Action Requests, the mediator sorts Action Requests (diamonds) according to likelihood, then checks to see if any requests are within some small amount (in this case, 0.4) from the maximum value ("equivalence region"). If an Action Request is within the equivalence region, the set is too close to call, and the mediator does not execute action ("deferral"). All requests in this example are final Action Requests.

action is accepted and all other actions are rejected (Figure 7.3). If there are multiple final actions within the equivalence region, the interaction is considered "too close to call". In this case, the mediator rejects all actions below the equivalence region, defers the final actions within the equivalence region, and accepts all feedback actions. If no final actions are accepted, and no final actions are deferred, then all feedback actions are always accepted.

7.3 Deferral

When actions are non final (feedback actions), the resulting interfaces from all update operations are probabilistically tracked until a user either disambiguates explicitly or implicitly through their behavior. However, final actions update program state, and thus need to be resolved. When multiple final actions are too close to call, the mediator defers this action. Deferral is handled in a separate logic path from the regular input loop, as illustrated in Figure 7.4.

The goal of deferral is to give the user an opportunity to disambiguate their intent before a decision is made. For example, if two final action requests are both very likely, it would be best if a user could provide input about their intent before the system decides for them. To accomplish this, the system provides a disambiguation interface rendering each of the deferred alternatives. This disambiguation interface is rendered using the same feedback system as will be discussed in Chapter 9, Figure 7.5 shows an example. Once again, the JULIA toolkit provides an extensible and customizable collection of mediation methods. The default mediation method is to display an "N-best list" choice dialog to the user, a common approach as described in (Mankoff 2001). To communicate the results of

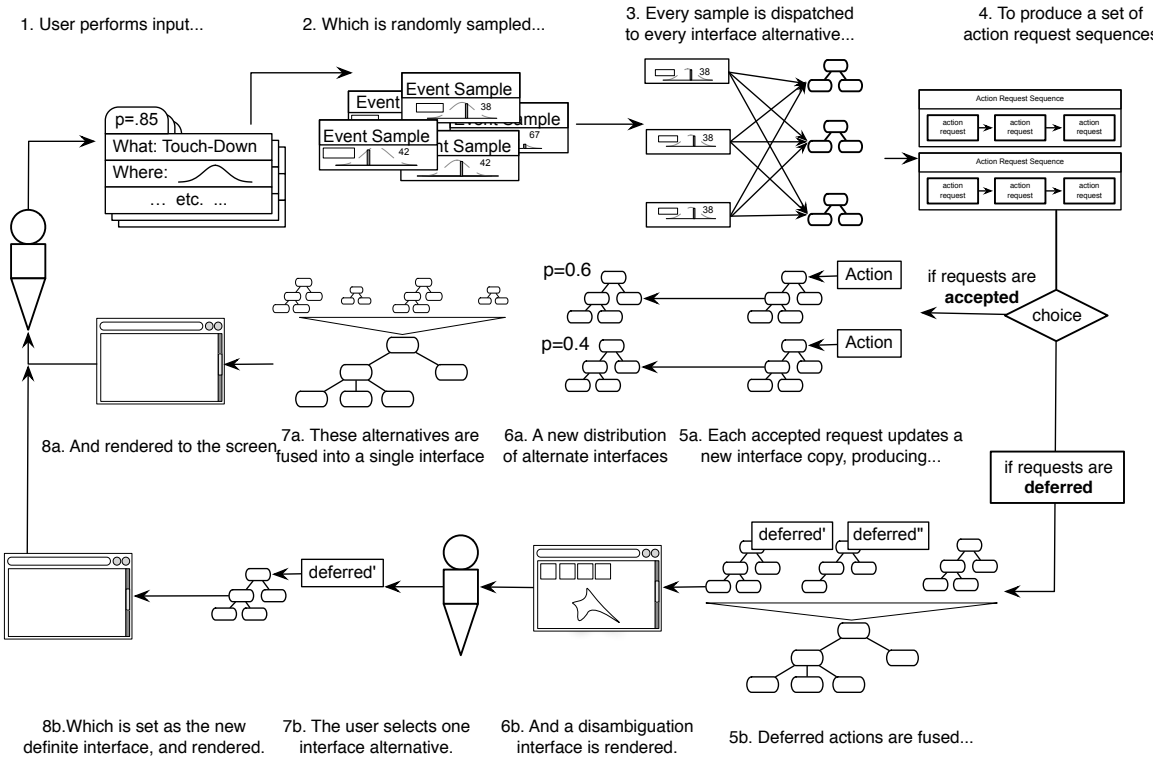


Figure 7.4 Diagram describing input-feedback-action loop, along with separate deferral loop (steps 5b -8b).

final actions, final action requests may optionally include a special draw function, which is called after the alternative is rendered to give a user further information about this action alternative. For example, if a final action may close an application, the draw method for the final action may render the text “close” onto the resulting interface alternative. This does not update the interface in any way (it is a temporary change).

Once the disambiguation interface is rendered, users must choose an alternative. Continuing to interact with the main interface implies selecting the most likely alternative, as this is displayed as the main interface on the screen. The details of ‘continuing to interact with the main interface’ is up to the developer of mediation mechanism to decide, we define this as a click on the main interface. Once the user chooses an alternative, the update code associated with this action request gets executed, the selected interface gets set as the new certain interface, and the regular dispatch process resumes.

The approach currently implemented has a few drawbacks, which we discuss here. Improvements to this approach are possible and we leave this for future work. The primary drawback of this approach is that mediation is forced immediately whenever any ambiguity arises. In other words, this does not allow mediation to incorporate the possibility of alternative interpretations that don’t arise until after mediation is immediately invoked. For example, if a final action arises and another action may arise a little bit later, the se-

Figure 7.5 Example of deferral. User executes a circle gesture (A-C), several interpretations are likely. The most likely interface is displayed by default, and possible alternatives are displayed as an n best list (D). The user may then select one of the alternate interpretations, or continue interacting using the most likely interpretation.

cond interaction will never be displayed to the user because the first action ambiguity needs to be immediately resolved. This specific area is an area ripe for future work.

7.4 Summary

The mediation process is responsible for determining which operations to execute. The component is pluggable, allowing for alternative, developer-specified mediation to easily be used without interrupting the rest of the interface. When final actions must be resolved, the mediator either resolves these itself, or asks the user for assistance (if it is too close to call). This process of waiting for as long as possible (via feedback actions) for further input has significant advantages over conventional dispatch, which immediately acts on all input events as they occur. By considering multiple interpretations and allowing decision to be deferred until a later time, the mediator allows for the JULIA toolkit to consider far more information than is available in conventional input toolkits when making a decision about interface action.

8 INTERFACE UPDATE

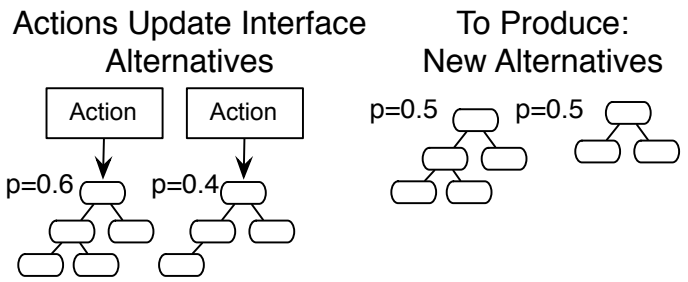


Figure 8.1: Overview of interface update.

After the mediator chooses which operations to execute, the JULIA toolkit needs to update its distribution over interface alternatives to reflect the new update operations. Every accepted action request represents one potential new interface state. The likelihood of an accepted request implies the likelihood of the resulting interface alternative. For example, if two action requests are accepted, this means that two interface states are possible. If only one action request is accepted, only one interface state is possible (this is always the case when a final action is accepted). Cancelled action requests represent an invalid new state, therefore cancelled requests simply do not get executed (and the resulting interface sample does not get put into the new distribution). The interface update process is responsible for updating the distribution over interface states.

The interface update phase can be broken into three parts: cloning interfaces, updating clones, and resampling alternate interfaces. Accepted action requests contain a reference to the alternate interface that generated the request, as well as an action function to update this interface. Interface update is performed by executing the interface update operation on the alternative. Because multiple action requests may refer to the same interface alternative, the first phase of interface update is to clone interface alternatives for every action request.

Note that it is possible for two action requests to come from the same transition from the same interface alternative, where one is a feedback action and one is a final action. To ensure that both are executed on the same interface alternative, one option is to combine requests like this during the action aggregation step (Section 7.1), turning the update function into a function that first executes the feedback update code, then the final update code. This will ensure that cases where action requests from the same transition on the same alternative are updating only one new interface alternative, not two.

8.1 Cloning Interfaces

Cloning interface alternatives is a fairly straightforward process. Every interactor implements a clone method that executes a deep copy of all properties and returns a copy of itself. Interfaces are cloned recursively according to the interactor hierarchy. While this current method is fairly inefficient, the area is ripe for optimization. The sequential nature of the interface alternatives makes the problem suitable for compression using, for example, incremental data structures (Tanin *et al*. 1996). More sophisticated lossless compression methods are also possible (Ziv & Lempel 1977).

One challenge that arises in dealing with interface clones is referring to interface elements. For example, if a developer has a variable pointing to a button in her original interface, this variable will no longer point to the proper button (or rather buttons, since likely there are now several alternate buttons that the variable could point to). If the developer wishes to update properties of other interactors (say change the text of a button) as part of an interface update operation, how does she accomplish this?

One approach is to assign each interactor an ID and refer to the interactor by ID. Every Container View (including the root view) implements a `findViewById()` method which can be used to find and modify views in an update function. Thus, for any update operation that involves other interactors, the root view for that interface alternative first finds the view of interest by ID, then updates its values accordingly.

Another option is to store references to objects themselves directly in the JULIA object that manages alternate interfaces, and modify these references appropriately during the update function call. Other approaches are also possible, including modifying the implementation language or binding the variables to different values according to the interface alternatives being updated. For our example implementation, however, we found that the simple method of tracking interactor IDs worked well.

8.2 Updating Cloned Interfaces

After interfaces are cloned, they must be updated with the appropriate update functions. The process for this is actually fairly simple: every action request is simply executed within the context of the newly cloned interface. Because the update function is executed within the context of an interface copy, no other interface alternatives are modified during this process, only the particular alternative interface corresponding to the new action request is modified. As a result, every alternate interface after the update function is complete represents an alternate representation unaffected by any others. The likelihood of this alternative is simply equal to the likelihood of the corresponding action request (which is in turn computed based on the likelihood of event samples and the result of action request aggregation).

Note that this model of tracking alternate interfaces is very convenient—if at any point the user wishes to select a particular alternate representation as a final representation (such as with final action requests, or through some other mediation process, some of which will be discussed in chapter 9), updating the interface distribution to reflect this is merely a matter of throwing out all other alternative interpretations.

On the other hand, a large number of action requests can lead to too many interface alternatives to manage, causing the number of alternate interfaces to explode exponentially. To solve this problem, the JULIA toolkit reduces the number of interface alternatives, which will be discussed in the next section.

8.3 Interface Resampling

After interface alternatives have been cloned and updated, it is possible that there will be too many alternatives to reasonably track. For example, a simple touch event with 100 samples could lead to 10,000 or more new interface alternatives. The final step that must be taken is resampling to reduce the total sample count. If left unchecked, over time the number of alternate interface samples will grow exponentially. This increase occurs each time multiple event samples arrive and are combined with interface alternatives. However, state distributions can generally be adequately approximated by a limited number of samples.

Before performing resampling, the JULIA toolkit reduces the number of interfaces by identifying sufficiently similar interfaces. The heuristics to identify identical action requests do not always catch all update operations that yield identical (or sufficiently similar) updates. For example, consider an update operation that sets the x position of an interactor to touchEvent.x modulo 10. Now consider two event samples whose x coordinates are 11 and 21. Both update operations will set the x position of an interactor to 1, however the action requests would be considered different because they used a different event sample as a parameter. As a result, an additional reduction step is needed.

8.3.1 Interface Reduction

Interface reduction takes a list of alternate interfaces and reduces this set to a smaller set of interfaces. In contrast to resampling, the aim of reduction is merely to combine similar or identical interfaces. Interface reduction is designed as a pluggable component, the JULIA toolkit provides a library of several reduction strategies. These reduction components are actually re-used in a later phase of feedback, which I will discuss in section 9.2.

To assist with interface reduction, every interactor implements an `isSimilarTo()` method which is used to compare interfaces. For example, two container interactors are similar if all of their children are identical, and if the two containers have

similar properties. Two interactors governed by probabilistic state machines are equal if their current state machine states are equal, and if all other properties are equal. The default interface reduction implementation uses this `equals()` method to determine if two interface alternatives may be combined.

The default interface reduction algorithm in the JULIA toolkit combines similar alternatives as determined by the `isSimilarTo()` method, however other reduction strategies are also possible. For example, if interface samples differ only by the position of one interactor, this set of interface alternatives may be combined into a single interface with that interactor positioned at the mean of the positions of all samples. This reduction strategy is implemented in the JULIA toolkit and is discussed in section 8.3.1. Once again, because this reduction stage is entirely pluggable, reduction strategies may be interchanged and developed as interface developers see fit.

8.3.2 Resampling Possible Interfaces

To avoid an explosion of possible interfaces, the JULIA toolkit ensures that the number of interface alternatives after each input processing step is no larger than a configurable threshold. I will refer to this threshold as MAX_POSSIBLE_INTERFACES. After reduction, it is likely that the number of possible interfaces is still larger than MAX_POSSIBLE_INTERFACES. Therefore, JULIA needs to resample and renormalize the interface distribution.

The resampling algorithm in the JULIA toolkit is pluggable, and can be replaced with any number of algorithms. The algorithm in the JULIA toolkit is a naïve algorithm to perform resampling: it simply takes the most likely MAX_POSSIBLE_INTERFACES interface samples. This approach works well for the applications of moderate complexity presented in this thesis, however has the disadvantage of removing the most unlikely samples. In the future we hope to provide more sophisticated resampling methods leveraging importance resampling. These methods adjust the probability distribution of the samples being drawn to ensure that likely unlikely (but important) samples are drawn, and then adjusts the weights of these drawn samples according to their likelihood in the original distribution. A common technique used in particle filters is sequential importance resampling, introduced by Gordon (Gordon *et al*. 1993). Because of the modular nature of resampling, algorithms such as sequential importance resampling may easily be integrated without disturbing the toolkit architecture at large. After resampling, the resulting set of state samples is a reduced set representing the new state distribution of the interactor.

8.4 Updating to a Certain State

The point of tracking interface alternatives to better infer a user's actual intent. In other words, the higher level goal of the input system is to correctly determine the actual cer-

tain interface state. When the mediation system has sufficient information to disambiguate input, when input is actually unambiguous, or when a user manually disambiguates input *via* interaction (*e.g.* select an item in an n-best list), the toolkit should update its state distribution to reflect having a single, certain state.

In section 3.2.2 I mentioned that the state of interface alternatives is represented by a list of interface alternatives as well as a *last_certain* interface representing the last certain state the interface was in. The JULIA toolkit has several ways of updating this *last_certain* view. First, when a mediator returns a single action request as being accepted, only one interface update operation is possible (all others were deemed invalid). Thus, the particular update operation is executed and the resulting interface is set as the *last_certain* interface, with probability 1. Second, when a user selects a particular alternative as 'correct', he/she is explicitly telling the input systems what the deterministic state is, and thus the particular alternative that should be set as the *last_certain* interface. Correcting for this is merely a matter of setting the alternative selected by the user as the *last_certain* interface and throwing away all other interpretations.

In contrast to other systems that have complex processes for resolving actions (Mankoff 2001, Schwarz 2010), the JULIA toolkit's mechanism is simple and elegant. Selecting a definite alternative is just a matter of updating the last certain interface.

8.5 Summary

The interface update phase is responsible for updating interface samples and downsampling this set of updated interfaces to appropriately reflect the new distribution of interface state. By cloning interface alternatives and executing update operations in the context of these cloned alternatives, the update phase is able to generate new state distributions without altering the state of other alternate interfaces. Downsampling is a key component that makes the operation of the JULIA toolkit feasible, without this the list of alternatives would grow to be far too large. By maintaining a reasonable number of interface alternatives at each step, the update phase ensures a reasonable approximation of the distribution of interface state.

The final phase in the JULIA toolkit is feedback, which is responsible for providing a meaningful rendering of interface state to the user, and allowing them optionally disambiguate their intent. The feedback phase is covered in the next chapter.

9 PRESENTING FEEDBACK

One of the key components of any user interface is a visual representation of the system state to the user. Users must be in a perpetual input-action-feedback loop in order to effectively interact with a computer. The above chapters have covered how input representation, state update, and action change when input is uncertain. This chapter provides an overview of how feedback changes under uncertainty, and provides an architecture for easily making these changes while requiring minimal developer effort.

The primary difference uncertainty brings is that rather than having a single, certain interface, the system is now dealing with a probability distribution over possible interfaces. The feedback system presented in this chapter provides the following facilities to deal with this new challenge:

- **Reduction** – reducing a large set of alternatives to a smaller set of 'salient' alternatives.
- **Identifying Differences Between Interface Alternatives** – an algorithm for identifying differences between interface alternatives.
- **Visualizing Uncertainty and Interacting with Feedback-** an API for building feedback objects that combine interface alternatives (and their likelihoods) into a single interface that communicates system state and allows users to specify which alternative is the 'correct' alternative.
- **Adjusting feedback based on context** – an API allowing developers to dynamically determine what feedback to show.

This architecture allows developers to easily experiment with different feedback techniques by allowing developers to easily switch between feedback methods as well as develop new feedback techniques. Just as with subjunctive interfaces (Lunzer & Hornbæk 2008), developers are able to change the method of rendering alternatives, as well as how alternatives are fused. To demonstrate the power and generality of our architecture, this chapter will also cover 12 feedback techniques already implemented using this system.

9.1 Overview

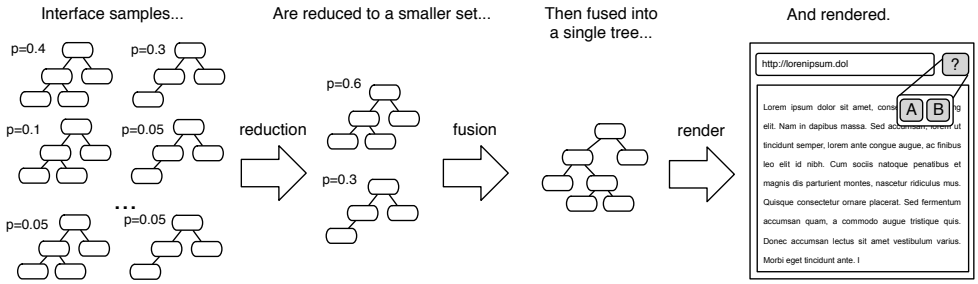


Figure 9.1: Overview of feedback process.

In a conventional interface, an interactor is made up of a set of properties; every time an interface property changes the interface system renders the new interface onto the screen. In the probabilistic input architecture, we now have not a single interface, but a distribution (PMF) over possible interfaces. The goals of the feedback system are three-fold. First, the feedback system should support rendering and combination of interface alternatives in a manner that is understandable to users. Second, this rendering should be efficient. Finally, the feedback system should be modular and extensible: developers should be able to pick between a number of different feedback strategies as well as develop their own.

To support this, I designed a modular feedback system whose components may be easily switched out and modified, along with a library of feedback techniques to allow developers to generate understandable feedback of uncertain interfaces. The feedback system has four stages (Figure 9.1). First, the system obtains a collection of interface samples from the result of interface update as discussed in Chapter 8. Next, the system may reduce a potentially large number of alternate interfaces into a representative subset. This subset is then fused into a single combined interface using one of many possible feedback algorithms. Finally, the fused interface is rendered to the screen.

9.2 Reducing the Number of Alternatives to Show

In many cases, the number of interface alternatives exceeds what could reasonably be communicated to the user, even when considering only the most likely interfaces. Therefore, our input system must reduce the number of interface alternatives in an efficient manner, while still representing the initial distribution as closely as possible. Several possibilities present themselves.

One method is simply to pick the N most likely interfaces and show then using one of the feedback techniques described below. While simple, this has the disadvantage of not rep-

resenting a large number of interfaces. For example, if there were 100 alternative interfaces, all of which had a particular interactor in a different location, showing only the top 3 versions would fail to communicate the possibilities. An alternate approach is to reduce the number of interfaces using one of the interface reduction techniques provided in the interface reduction step of interface update (Section 8.3.1).

For example, one such interface reduction algorithm that would be particularly useful is to aggregate all possible values from interface alternatives into a single interface. This reduction algorithm turns each property of an interactor into a PMF over all values the property takes on across all interface alternatives. In other words, now each property (for example, the x coordinate of a location) is not a single value, but a list of values. Interfaces that have new interactors in them would have these interactors added in the final interface.

This reduced interface (or interfaces) can then be fed into different feedback algorithms. As will be seen shortly, these algorithms can then take this list of properties and render the mean, standard deviation, or other aggregated properties as they see fit. One final note about this particular aggregation approach is that instead of combining all values, interfaces may first be clustered and then combined (although we leave the implementation of this in a library for future work).

One major advantage of this reduction step is that it is often doing the same operation as interface reduction in Section 8.3.2. This means that all reduction algorithms implemented for feedback may be used in interface update, and vice versa. Developers may write an aggregation algorithm once and reap the benefits in multiple parts of the toolkit.

9.3 Identifying Differences Between Interface Alternatives

When giving feedback about interface alternatives to the user, it is important to identify which interface components are different from a 'ground truth' (in this case, the last certain state). As mentioned in sections 3.2.2 and 8.4, the JULIA toolkit tracks a *last_certain* interface representing the last certain interface state. Rather than comparing all alternatives to each other, JULIA compares alternative interfaces to this last interface state to determine which components have changed.

Several methods for determining differences are possible. First, all interactors already implement an `isSimilarTo()` method to identify similar interfaces during action aggregation (section 8.3), which can be used to similarly identify specific differences in interfaces. Interactors are marked as 'dirty' when they differ from the *last_certain* interface. Any interactor is dirty if it does not exist in the *last_certain* interface, or if its state has

changed between interfaces. Similarly, containers with different numbers of children (either added or removed) or children of different orderings are marked dirty. These dirty bits can also be set during state update, since state update by definition modifies properties. Rather than comparing all interfaces to the *last_certain* interface, our implementation modifies the dirty bit during state and container update, assuming that interactors can not be changed at other times.

Once differences have been identified, different feedback objects can examine these differences to provide more meaningful feedback to the user, for example highlighting specific objects that have changed.

9.4 The Feedback Object: Specifying What Type of Feedback to Show

In conventional interfaces, interactor developers need to implement their own feedback rendering. Implementing this in the context of probabilistic interfaces leads to extra work, inconsistent visual styles, and errors. One of the aims of the work presented here is to abstract communication of uncertain system state away from the developer in order to facilitate a consistent and reusable way to communicate ambiguity. Importantly, it should be very easy for developers to specify what form of feedback to show, and for developers to write their own forms of feedback either by extending existing feedback algorithms or writing their own.

Rendering feedback in the JULIA toolkit is handled through a feedback object which is responsible for taking interface alternatives and generating a resulting fused interface which is then rendered to the screen. This involves performing reduction, identifying differences, and finally fusing this information into a single interface which will be displayed on the screen. Note that some of these steps, *e.g.* interface reduction and identifying differences are themselves implemented as interchangeable objects, allowing for maximum code reuse. Also note that a feedback object only needs to implement a draw method that takes a list of alternatives, meaning feedback can be arbitrarily simple or complex. For example, later on we will introduce meta-feedback objects, which are feedback objects that contain other feedback objects, and choose which feedback to display based on contextual information.

Developers specify which feedback to show by specifying which feedback object to use in their interface. For example, one form of feedback is to simply render the most likely interface to the screen by calling `draw()` on the root of the most likely interface. This is the default feedback method used in our input system and is what conventional interfaces do.

Figure 9.2 Example of Octopocus-inspired feedback technique of overlaying multiple alternate interpretations using opacity (left), blur (middle) and saturation (right) to reflect probability. The dot is the location of the touch position. The user is drawing an arrow pointing right.

Feedback objects implement a very simple interface: They have a single draw() method which takes a canvas to draw on along with a list of interface alternatives (the result of interface updates). This simple interface allows for a surprisingly complex set of opportunities, which we describe below. Developers may implement their own feedback methods, use one of the many feedback methods provided in the JULIA toolkit, or extend an existing library feedback method.

9.5 Visualizing Uncertainty

As seen in step 3 of Figure 9.1, the fusion step takes a list of interactor trees (each tree represents an interface alternative) and fuses these trees. The simplest option, what in practice conventional interfaces do, is to render the most likely interface. In fact, this is a good option in many cases. However, when several interfaces are highly likely, or when the user indicates indecision (*e.g. via* a dwell), it is worthwhile to communicate alternate possible interface states by visualizing the uncertainty to the user. Below we describe several feedback methods we have implemented to demonstrate the generality of our approach.

9.5.1 Overlay Multiple Versions

Inspired by the feedback techniques presented in Octopocus (Bau & Mackay, 2008), we observed that it may occasionaly be beneficial to overlay interface alternatives on the interface, passing alternatives through various filters. here, all alternatives of an interactor are grouped into a single container, which can render these alterantives as it sees fit. The overlay container can adjust the size or appearance of child interactors (alternate versions of a single interactor) based on probability. For example, it may use alpha compositing, adjust contrast or blur based on likelihood values. Figure 9.2 illustrates our explorations of different forms of feedback for the Octopocus interaction

Figure 9.3 Example of animation feedback. Text fields fade through alternate values.

technique. For background, the Octopocus interaction technique aims to help users learn gestures by overlaying all possible gesture completions on the screen, so that users may better understand how to move their finger to complete a certain action. The feedback visualization techniques explored for communicating probability included adjusting sharpness, opacity, scale, contrast, and rendering separate visualizations (*e.g.* progress bars and text). Another example of overlay in existing work is to draw a 'dashed' or 'impermanent' version of a possible alternative, as shown in Igarashi's interactive Beautification (Igarashi & Matsuoka 1997). To support this, interactors expose a '`drawAmbiguous()`' method which can render an interactor using dashed lines as with Igarashi's work, or in any other form desired.

9.5.2 Animate Between Alternatives

Animation can be used to switch between input alternatives or convey number of alternative interfaces. For example, individual interface elements may jitter or pulse with a frequency according to the number of alternative representations (Figure 9.3).

More practically, individual interface elements may fade between possible values. To the best of our knowledge this form of feedback has not yet been explored in the literature, and provides a new area of feedback to be evaluated and explored by research practitioners.

The advantage of our toolkit is that it allows for these feedback methods to be easily implemented and explored, without requiring any effort on behalf of the interface developer. Developers of feedback may work entirely independently of the interface implementer.

9.5.3 N-Best Lists

The goal of the N-Best list is to quickly show a small number (typically less than 5) of alternatives near the area of interaction. This family of feedback methods is similar to the work presented in Side Views and more generally the work on Subjunctive Interfaces cov-

ered in Section 2.2.3. The most likely interface is rendered in the main interaction area, and interface alternatives are rendered near the point of interaction, or at some other developer-specified location (Figure 9.4). These alternatives can be rendered in several different ways, which we describe below.

Figure 9.4 Examples of N-Best lists. Left: when it is unclear which folder a user intends to drag, several alternatives pop up. Center: when several gesture recognition results are possible, show alternatives. Right: It is unclear whether a user wishes to drag the rectangle or resize, show alternative options.

First, scaled versions of the entire interface may be rendered (Figure 9.4, right). While simple to implement, this has the disadvantage of being difficult to interpret. One alternative is to render only the interface components that have changed (Figure 9.4, Left, Center). This is done by comparing each interface alternative to the most likely view, marking which interactors are different (have different properties or do not exist in the original), and rendering these to the screen. However, rendering only the changed views removes the interface context and can also be difficult to interpret.

A third approach is to render a scaled version of every interface alternative and highlight the changed regions (Figure 9.5, left). Developers can dynamically switch between any of these feedback methods merely by changing the type of feedback. Many other approaches are possible, for example summarizing changes as text, or adding animation. The existing N-Best Lists can easily be extended to add these changes with minimal developer effort.

Figure 9.5 Examples of alternate N Best list renderings. Left: Highlight changed regions, Center: Communicate probability using opacity. Right: Communicate probability using text.

Alternatives are sorted by probability and displayed left to right (though of course other display strategies are also possible), and various filters or effects may be applied to alternatives to communicate the likelihood of every alternative, for example opacity, size, and contrast. We also implemented a simple thermometer visualization comparing likelihood (a completely full thermometer indicated probability 1).

9.5.4 Rendering Aggregate Properties

As mentioned in Section 9.2, the results of different interface reduction algorithms may be fed into different types of feedback to create interesting forms of aggregate feedback. For example, we can take the output of the property aggregator discussed in Section 9.2 to render the mean, mode, median, and standard deviation of different interface properties. Figure 9.5 shows an example of rendering the mean and standard deviation of a cursor's position, with a simple overlay of all possible values on the left.

There is a wide body of work on visualization of uncertain information which this aggregate feedback mechanism can implement. Some examples are explored in (MacEachren 1992) and (Pang *et al*. 1997). The beauty of the approach in this toolkit is that it now enables the rendering and exploration of these visualization techniques in the domain of user interfaces, something that has not yet been demonstrated before.

9.6 Interacting With Feedback

In many cases, visualization of uncertainty naturally leads to a desire to resolve this uncertainty. The concept of mediation is covered in Section 6, however because feedback and mediation are so closely linked, a treatment of mediation in this section is warranted here.

As an example, we have implemented several methods for selecting interface alternatives during and after interaction. While most of our examples are implemented using some

Figure 9.6 Example of interactive feedback. After a user executs a gesture, several alternatives are possible. If she selects the rectangle, this alternative will be selected. If she ignores the n-best list, the circle will remain on the screen.

form of N-Best list, alternative selection may also be possible using different feedback methods.

As discussed in Section 8.4, the implementation for selecting alternatives is very simple: when a user selects an alternative, this alternative is marked as 'certain' and all other alternatives are removed. For example, in Figure 9.6, after a user executes a gesture, several interface alternatives are presented, representing alternate interpretations of her gesture. She selects one of them and this is set as the certain interface.

Additionally, in Figure 9.7 the user wishes to drag a box, but the system interprets her actions as resizing the box. She wishes to change to an alternate interpretation. This is accomplished by dragging his finger/mouse through the proper alternative, as with crossing interfaces. Afterward, the correct alternative is selected. Note that a limitation of this interaction is that the square in the selected alternative may become dragged into a position that is unintended. A second limitation is that an n-best list item may obscure the location where a user wishes to drag the item, causing an n-best list item to be unintentionally selected.

A similar effect can be achieved by pressing a key mid-interaction, or by using voice input. Alternatives disappear after a timeout or after a user completes a gesture. Other error correction strategies such as those reviewed by Bourguet (Bourguet 2006) are possible and easily added in our architecture.

Figure 9.7 Example of interactive feedback. A user wishes to drag the left box, however it is being resized instead (A). *While keeping the mouse down*, she moves over to the leftmost interface alternative, which represents a drag (A dotted line, B, C). Once she crosses over the desired alternative, it is selected.

9.7 Feedback About Future Events

In addition to providing feedback on the current state of an interface (and its alternatives), it is possible to emulate potential future events and render the possible future interfaces resulting from them. For example, input prediction models might predict future pointer locations or future events given past behavior. These input event predictions (and their likelihoods) can be simulated through our system as probabilistic events, and the resulting alternatives presented as feedback to the user and make them available to be selected as shown in the previous section. This could both provide prompting indicating to novices what follow-on possibilities there are, and provide an automated and generalized mechanism for the system to produce shortcut interactions for expert users without having to explicitly build such shortcuts into every part of the interface.

Figure 9.8 shows an example of future feedback discussed in the 'predictive menus' demonstration from Section 10.1.6. This demonstration depicts a menu that seems to predict which items a user is likely to select based on their current selection and their past behavior. Note that the menu interactor is actually nothing special and does not have any prediction logic. The event prediction is handled by a separate component, while feedback is handled by the JULIA toolkit.

Here feedback about possible menu selections can be used to speed up menu selection and also educate users about common menu options. The results of prediction can be communicated using any of the techniques above. In the development of this demo, I was able to experiment with all 12 different feedback techniques presented, Figure 9.8 shows a few of the most promising options.

Surprisingly, I found the opacity overlay (Figure 9.8 bottom left) to work well in educating novices about commonly used menu items, something unexpected that came from easily being able to select different feedback techniques.

There are several ways that future event prediction can be implemented, here we cover two alternate methods. First, after each mouse or key event is dispatched, a special 'event predictor' event source may separately dispatch a new 'prediction' event representing alternate future follow-on inputs. As a result, all future interface states caused by prediction are fully reversible, meaning the resulting alternate interfaces always get generated and rendered. A distribution over possible interfaces is generated from this predicted input event, and results are rendered to the screen. This is the method used to implement prediction in predictive menus. Note that this method does *not* require the development of special interactors.

A second approach is to generate event prediction in response to transitions on interactor states. For example, when the menu interactor makes a particular transition, it may generate a new 'prediction' event that needs to be dispatched to the interface. This is a simi-

lar approach to that used in generation of draggable events discussed in Section 10.3.2. Again, all predictions generate feedback actions, which in turn generates a list of alternate interfaces that are rendered to the screen using the same feedback mechanisms used for current ambiguous alternatives.

Presenting feedback about possible future events is a strong and unexpected capability that comes as a result of the architecture being general and extensible. This approach will become especially useful as event prediction becomes more popular. Our feedback approach makes it easy to communicate possible future events (and their likelihoods) without requiring any additional work.

Figure 9.8. Several examples of feedback about future events. The user has moved his mouse over the Edit menu, the likelihood of each subsequent menu item selection is presented in the bottom right and is based on past behavior. Top Left: An n best list of the 4 most likely alternatives is displayed. Top Right: A more compressed n best list is displayed, this time showing a custom rendering of menu commands. Bottom Left: Every interface alternative is overlaid, with opacity reflecting likelihood.

9.8 Meta-Feedback: Adjusting Feedback Based on Context

In many cases developers may wish to alter the feedback method used based on the likelihoods of alternatives, input types used, or even (within an interface) the interactors that have ambiguity.

To support this and allow for maximal flexibility, we introduce the notion of meta-feedback objects: feedback objects which decide what form of feedback to display given system state. Meta-Feedback objects are just feedback objects that contain other feedback objects. Meta-Feedback objects analyze the inputs or alternatives and select which feedback object to use for rendering based on context. Because meta-feedback objects implement the feedback interface, meta-feedback can be combined in any order, allowing for arbitrarily complex feedback.

Below we give examples of several meta-feedback objects implemented in our system as examples.

9.8.1 Inspecting Interface Likelihoods

When several alternatives are likely, developers may want to render another form of feedback, for example an N-Best list (see section 7.5.1). The `DeltaLikelihood` feedback object allows developers to specify a threshold and feedback types indicating what feedback is shown at which threshold levels. For example, if feedback should be shown when there is more than one interface within probability 0.5 of the most likely interface, the developer would specify a delta of 0.5, and two feedback types `MostLikelyFeedback` and `OpacityFeedback`. When the next most likely interface is within delta > 0.5, the most likely interface is shown, otherwise all interfaces within 0.5 of the most likely interface are overlaid using opacity.

9.8.2 User Hesitation

Another scenario when developers may want to show feedback is when users indicate indecision, for example when a cursor is dwelling on the screen. The `Dwell Feedback` object shows the most likely interface until a user dwells on the screen, at which point it displays a new feedback type specified by the developer. This feedback can be combined with the `DeltaLikelihood` object to only show feedback when the user has dwelled and interface alternatives are within some threshold of one another.

9.8.3 Input Type

The `InputTypeMetaFeedback` object allows developers to adjust the feedback rendered based on the type of input event. For example, one could render the mean interface when users are using mouse, and change the feedback method to show only the most likely interface when the keyboard is used. Developers may adjust this using a special feedback type, `InputTypeMetaFeedback`, specifying a mapping from input type to feedback.

9.8.4 Number of alternatives

Certain types of feedback, for example N-Best lists, may not be appropriate when the number of alternatives is large. Therefore, it may be desirable to adjust the type of feedback based on the number of interface alternatives present. Developers may use the `NAlternativesMetaFeedback` feedback type to adjust the feedback based on the number of alternatives.

9.8.5 Which interactors are ambiguous

Not all types of feedback are appropriate for all interactors. For example, a feedback to render the mean interface is appropriate for a cursor but not a text field. Developers may specify a mapping between interactor id (or interactor type) to a feedback object *via* an `InteractorMetaFeedback` object, adjusting feedback accordingly. This feedback type first splits an interface according to the filters specified for a feedback object. It then renders feedback appropriately according to each filter type, using the feedback object specified for that filter type. The resulting renderings are then layered on top of one another appropriately.

9.9 Summary

In this chapter, I have proposed building a rendering system that renders the uncertain state of an interface while allowing developers to stick to a conventional and familiar method for rendering interfaces. Five general forms of feedback were presented, each of which can be easily configured in several ways leading to dozens of different feedback techniques that may be implemented.

10 VALIDATION BY DEMONSTRATION

The aim of the JULIA toolkit is to provide a robust mechanism for handling uncertainty across input sensing, interpretation, and application action in a general, re-usable fashion. To demonstrate the viability of the JULIA toolkit, I explore a total of 17 case studies, in the form of interaction techniques and applications. The 17 case studies discussed in this chapter illustrate that the probabilistic input architecture handles many different kinds of ambiguity, enables rendering of a wide variety of feedback, is easy to work with, can be used to implement traditional deterministic user interfaces, and can be implemented on a wide variety of platforms. Additionally, the demonstrations illustrate that even without any optimizations the architecture performs well enough to be usable in fairly complex interfaces.

This chapter is organized as follows: first, I present nine new interaction techniques developed using the probabilistic input architecture that demonstrate not only that this new architecture provides a foundation for a multiplicity of new interaction techniques, but also that the architecture can handle a wide range of ambiguity types. Next, I present seven feedback techniques that demonstrate the versatility of the probabilistic input architecture. Third, I present three more complex applications to demonstrate that the architecture scales well to full interfaces, and is backwards compatible. Finally, I describe the platforms that successive versions of the toolkit were implemented on, demonstrating that the architecture works well in a desktop, mobile and web environment. Table 8.1 gives an overview of the work presented in this chapter. The demonstrations discussed in these chapters were built using different iterations of the JULIA toolkit with somewhat differing architectures, ambiguity representations and state update methods (see Section 10.4). While in actuality they were built using different iterations, it will become evident that each of these demonstrations could have been built using just the final version of the toolkit. The breadth and code simplicity of these demonstration applications show that this toolkit is not only powerful, but is also simple to work with and can be made to perform reasonably well within the near future.

10.1 Interaction Techniques

In this section I will present nine new interaction techniques developed using the probabilistic input architecture. These techniques demonstrate not only that this new architecture provides a foundation for a range of new interaction techniques, but also that the architecture can handle a wide range of ambiguity types.

		Sensor Ambiguity	Interpretation Ambiguity	Action Ambiguity	Provides Feed-back	Input Modality
Interaction Technique	Two button dialog	✓			✓	Touch
	Interacting With Multiple Sliders		✓			Touch
	Smart Window Resizing		✓			Touch
	Predictive Menus			✓	✓	Mouse
	Bayesian Draw		✓	✓	✓	Touch
	N Best List for Ges-tures		✓		✓	Touch
	Ninja Cursors		✓	✓	✓	Mouse
	Smarter Text Entry	✓	✓			Keyboard/ Speech
Feedback Technique	Octopocus				✓	Touch
	Side Views				✓	Any
	Hover Widgets	✓	✓	✓	✓	Touch
	Future Feedback	✓	✓	✓	✓	Any
	Animation Between Alternatives	✓	✓	✓	✓	Any
	Mean Feedback	✓			✓	Any
	Highlight Only Changed Regions	✓	✓	✓	✓	Any
Application	GesturePaint	✓	✓	✓	✓	Touch
	Interactive Beautifi-cation	✓	✓	✓	✓	Touch

Table 10.1: Overview of completed validation presented in this chapter. Work is organized by type (Interaction Technique vs. Application), and then criteria are presented.

10.1.1 **Probabilistic Buttons**

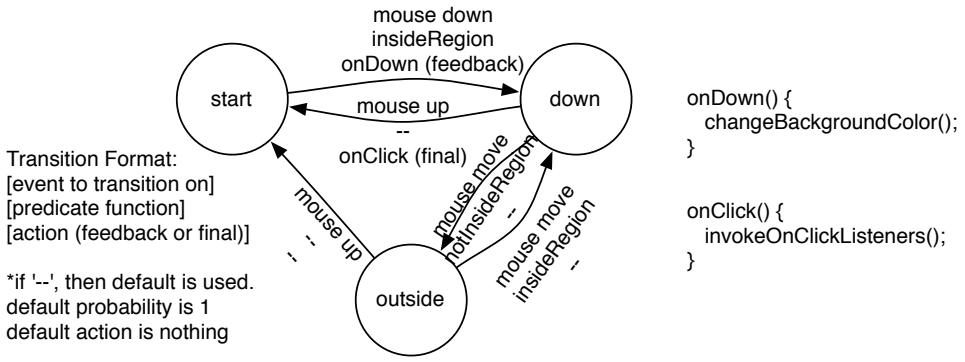


Figure 10.1. Top: state machine for a button. Feedback and final actions are present on the start → down transition and down → up transition. Bottom: Example of interface distribution (bottom right) and feedback (bottom left) from a touch event.

The example of the two-button dialogue runs throughout this dissertation, and this example demonstrates handling uncertainty at the sensor level. Specifically, the touch sensor in this example introduces sensor imprecision, as it is unclear which specific location the user intends to target when he or she is touching the screen. By representing the touch location as a distribution of possible touches, then aggregating all possible update operations, the JULIA toolkit is able to integrate the likelihood of possible button press states over a set of possible locations and determine the button that the user most likely hit. If two buttons are equally likely, then the toolkit does not execute action and waits for further user disambiguation. Note that it is also possible, through the use of probabilistic transitions, to incorporate a prior belief about the likelihood of a button being pressed. The two button dialog demonstrates how a simple button can be made to respond intelligently to user behavior.

As Figure 10.1 illustrates, the state machine logic for these buttons is very similar to that of a standard button. The probabilistic toolkit does all of the heavy lifting regarding tracking of alternatives and likelihoods, so the developer does not need to worry about it. As can be seen from the simple state machine description, this example demonstrates how the logic for one of the most basic interactors remains simple under the JULIA toolkit because the system takes care of tracking likelihoods and alternatives.

The demo illustrated in Figure 10.1 was written in JavaScript for the third iteration of the toolkit (see section 10.4). In this demo, 20 event samples are generated for each touch event, and at most 10 alternative interfaces are maintained. The button demonstration, including the specification of the button interactor, totaled to about 150 lines of uncompressed JavaScript code.

10.1.2 Ambiguous Sliders

Sliders (Figure 10.2) demonstrate handling of uncertainty at the input interpretation and sensor level. At the input interpretation level, one form of ambiguity is target ambiguity: it is sometimes unclear which particular interactor a user intends to interact with. In this demonstration, users can actuate sliders at a distance, meaning they do not need to be directly over the sliders to manipulate them. To determine the likelihood of actuation, each slider computes the distance between it and the current move event, as well as the direction of motion of the finger. As shown in Figure 10.2, all viable target sliders show feedback. This example demonstrates how the JULIA toolkit sends inputs to multiple interactors, computes action likelihoods, and delays action when input is uncertain, allowing instead for the interactors to provide application feedback.

This demo was written for desktop, for the first iteration of the toolkit. This particular iteration did not use event samples (see Schwarz 2010), however I estimate that about 20 event samples per touch event, down sampling to 10 alternate interfaces at each step would do a more than adequate job of supporting this interaction technique. The demonstration code totaled about 200 lines of C# code.

Figure 10.2: Sliders provide feedback when a user moves in between two sliders.

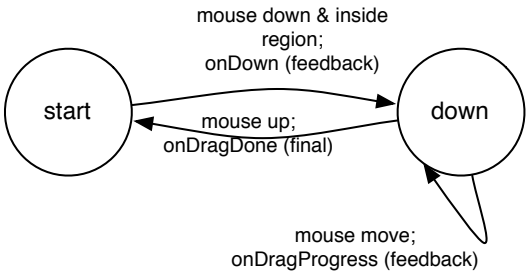
10.1.3 Smart Window Resizing

Smart window resizing demonstrates how my toolkit handles target ambiguity. In this example (Figure 10.3), when a user presses down, it is unclear whether she intends to resize a window by moving horizontally, or move the icon beneath the window. Subsequent motions serve to disambiguate this: horizontal motion means a window resize, while more vertical motion indicates movement of the icon.

Figure 10.3: Illustration of implicit disambiguation for touch input. **Left:** User 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.

Figure 10.4 shows the state machine for both the resizable window (left), and the icon (right). When a user initially touches down (Figure 10.3, left), her finger overlaps both the icon and the resizable window, making both the icon and window possible interpretations. As the user moves her finger vertically, the transition weight on the resize transition in Figure 10.4 decreases, and the interpretation that the window is being resized becomes less likely. Appropriate feedback is drawn to the screen. When the user releases her finger, the interpretation for icon movement is sufficiently more likely than a window resize, that this final action gets accepted and a new deterministic interface is set. This demo is implemented in JavaScript, with 20 samples per input event and at most 10 interface samples at the end of each dispatch loop.

Icon Probabilistic State Machine

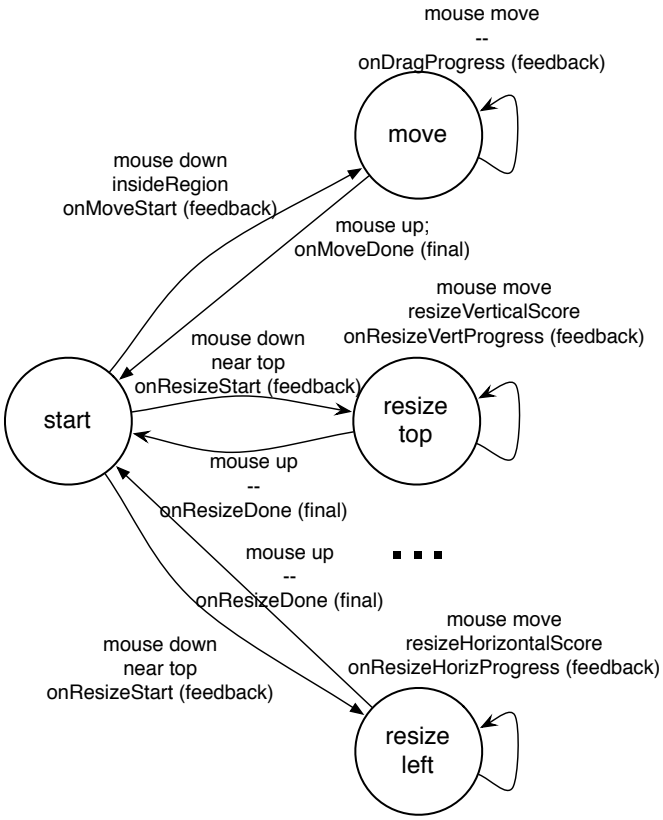



```
onDown() {
  downMousePos = getMousePos();
  downIconPos = getIconPos();
  sendDragBeganEvent();
}

onDragProgress() {
  positionChange = getMousePos -
                   downMousePos;
  iconPos = downIconPos + positionChange;
  sendDragProgressEvent();
}


onDragDone() {
  // send a new event in
  sendDragCompletedEvent();
}
```

Resizable Window Probabilistic State Machine



Predicate Functions:

```
insideRegion() {
  if(mouseInsideBoundingBox()) {
    return 1;
  }
  return 0;
}
resizeVerticalScore() {
  angle = absoluteValue(getAngleOfDrag());
  // remapValue(min_in, max_in, min_out, max_ou
  return remapValue(45, 90, 0, 1, angle);
}

resizeHorizontalScore() {
  angle = absoluteValue(getAngleOfDrag());
  // remapValue(min_in, max_in, min_out, max_ou
  return remapValue(90, 45, 0, 1, angle);
}
```

Action Functions

```
onMoveStart() {
  downMousePos = getMousePos();
  downWindowPos = getWindowPos();
}
onResizeStart() {
  downMousePos = getMousePos();
  downWindowSize = getWindowSize();
}

onResizeStart() {
  downMousePos = getMousePos();
  downWindowSize = getWindowSize();
}

onResizeHorizProgress() {
  positionChange = getMousePos -
                   downMousePos;
  windowSize = downWindowSize + positionCha
}

onDragProgress() {
  positionChange = getMousePos -
                   downMousePos;
  windowPos = downWindowPos + positionChai
}
```

Figure 10.4 Probabilistic state machine description for resizeable window and icon interactors

10.1.4 Text Entry in a Form

Figure 10.5 Behavior of probabilistic text entry. A user is typing his phone number. Initially (left), email, ssn and phone are possible inputs. Once the user has typed more than 9 characters (middle) ssn is no longer an option. Finally the user clicks the 'phone' field, and this is selected as ground truth (right).

Smarter text entry illustrates how target ambiguity is resolved during the feedback phase. 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 first selecting a text box, and those that do respond often do so by selecting 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.

The JULIA toolkit can address this problem with text fields that change their action likelihood based on what content they expect to receive. If several text boxes can receive input, the JULIA toolkit delays action and provides feedback accordingly.

I implemented two examples – smart text delivery and speech text entry. Both examples deliver text to a form with several different fields, *e.g.* name, phone number, and email. Because it is initially unclear which text field should handle input, the JULIA toolkit ends up tracking *n* alternate interfaces when a voice or key event is dispatched, where *n* is the number of form fields. This creates *n* alternate interfaces, each of which has a different text box in focus. Subsequent key events are dispatched to the text field in focus, likelihoods of alternatives are adjusted based on how well the entered text matches what the field expected. As the input becomes less likely, the transition likelihood decreases (or becomes zero). Once the transition likelihood is zero, no interfaces are generated as a result of input and the alternative is removed.

Figure 10.5 demonstrates the behavior of these new text fields. When a user types text without having first selected a textbox, all text boxes which have not yet been filled in show the typed text in gray. A user can then continue typing or select the correct textbox. The textboxes have a special behavior attached to them: If a text box is selected, the al-

ternative corresponding to this text box is set to the *last_certain*, or correct view. Note that this is all handled by the toolkit, the text box merely sends a request to finalize interface ambiguity. When a user presses enter, all text fields send a final action requests with a corresponding likelihood.

For speech input, recognition results often contain multiple uncertain interpretations, such as “2” and “q”. All uncertain interpretations are sent to all text fields and shows feedback appropriately, as before. The likelihood of a recognition result is propagated to the likelihood of different interface alternatives. When a user says the word ‘next’ or pauses long enough, the text fields send final action requests, and the most likely interface alternative (corresponding to the most likely text field) is selected.

The form demo also incorporates a simple Bayesian behavioral model to further disambiguate results. Since people tend to fill out forms top to bottom, the topmost entry is initially more likely than others. Also, given the index of the last entered element, the next highest element is more likely than other elements.

The probabilistic input form demonstrates three features of the JULIA toolkit. First, the example shows how the likelihood scores of recognition events are directly translated to interface likelihoods rather than immediately being thrown out. This example also demonstrates that the JULIA toolkit works well for new input types such as voice input. Finally, the example demonstrates how voice recognition uncertainty can be combined with a behavioral model to further reduce uncertainty.

For this demonstration, 20 event samples are tracked, and interfaces are down sampled to 10 alternatives at each step. This demo was implemented using the third iteration of the toolkit, and totals about 200 lines of JavaScript code.

10.1.5 Rapid Disambiguation Between Text Selection and Scrolling for Touch Interfaces

Selecting text can be very cumbersome on mobile devices, requiring a press and hold interaction. The reason for this is that when the finger touches down it is unclear whether a user intends to interact with the scrollable content or the scroll view itself. Touch-based operating systems take a seemingly ad-hoc approach of requiring a press and hold to trigger a text selection. Our probabilistic approach easily circumvents this issue by simulating both alternatives and waiting for future finger movements to disambiguate the interaction. Figure 10.6 illustrates this disambiguation. The algorithm uses a simple heuristic: the total amount of vertical or horizontal motion to adjust the likelihood of selecting text vs. scrolling. The most likely interface is presented on screen. When a user pauses, the alternate interpretation is shown on screen. Users can switch to this interpretation simply by tapping on it.

Figure 10.6 Demonstration of disambiguation between text selection and scrolling. Both alternatives are tracked with likelihoods (bottom row, center, bottom row, right). The most recent certain state is also tracked (bottom row, left). In this case the system believes the user is scrolling with likelihood 0.94. A user may switch interpretations by selecting the alternative in the top left corner.

This demo was implemented in JavaScript. 20 event samples are generated for every key event, and interfaces are downsampled to 10 alternatives at each step.

10.1.6 Predictive Menus

Menus in desktop applications can often be complicated and difficult to learn. Furthermore, with the exception of the 'recent items' list many applications support, along with the ill-fated adaptive menu feature provided in Microsoft Office 2003 (Microsoft 2003), menus rarely adapt to our behavior. In fact, it has been shown that changing the menu

items themselves to *e.g.* put more commonly used items up top is actually less efficient (Findlater 2004). Nevertheless, it would be nice if menus could take advantage of usage behavior to predict what users will select and accelerate menu selection. Additionally, for novice users, it would be beneficial to visualize commonly used menu items.

To solve both of these problems, I built a predictive menu system which supports both of these requirements without rearranging or modifying the underlying menu. Figure 10.7 shows an example. Menu items have different selection likelihoods given the currently selected menu item. These likelihoods are built using the Markov assumption, and based on user history. Given the likelihood of menu items, the predictive menu shows one of several developer or user-specified interfaces.

First, the menu can show the most likely interface, corresponding to a conventional menu system. Second, the menu can overlay possible future menu selections given current input (Figure 10.7), adjusting the opacity based on the likelihood of selecting a particular

Figure 10.7 Illustration of predicted menus. A user has moved his mouse over the Edit menu, the likelihood of each subsequent menu item selection is presented in the bottom right and is based on past behavior. Top Left: An n best list of the 4 most likely alternatives is displayed. Top Right: A more compressed n best list is displayed, this time showing a custom rendering of menu commands. Bottom Left: Every interface alternative is overlaid, with opacity reflecting likelihood.

item. Third, the menu can show an n-best list of possible menu item selections, ordered by likelihood. Thus, when the user selects Edit…, the most likely menu item, Gaussian Blur, is shown as the most likely completion. Users can then either select the completion or press the corresponding key to jump right to that option. Several renderings of n best lists are possible, as shown in Figure 10.7.

Although the capabilities of this demo are fairly complex, the implementation is no more complex than that of a normal hierarchical menu. In fact, more complexity is devoted to the implementation of the hierarchical menu itself than to rendering predictions. Menus respond to mouse events, behaving in the same manner as standard hierarchical menus. Menus also respond to special "select menu item" events, which specify a specific menu item to select. When a menu item is selected, the menu interactor determines a list of predicted future events (and likelihoods) and dispatches a new probabilistic event, which generates a distribution of events samples according to the prediction likelihoods. Each of these probabilistic events is then handled by the menu interactor, and interface alternatives representing possible future selections are generated. These alternatives are then displayed as interactive n-best lists or as overlaid feedback.

This example demonstrates several things. First, it shows how the JULIA toolkit supports prediction of future events. Second, it demonstrates how prediction of future events can accelerate interactions. More importantly, it demonstrates how complex interactions such as predicting future actions and showing completion can be added to interfaces with minimal developer effort. The JULIA toolkit handles all complexity regarding tracking interface alternatives, rendering feedback, and disambiguation.

10.1.7 Ninja Cursors

To further demonstrate how the probabilistic input architecture enables complex interaction techniques can be built using simple logic, we re-implemented a version of Ninja Cursors (Kobayashi 2008). Ninja cursors is an interaction technique that uses multiple cursors to reduce target selection time. Rather than manipulating a single cursor, users manipulate an array of cursors distributed evenly across the screen. The original technique used a clever set of heuristics to ensure that no more than a single cursor was present inside a target at a time. Figure 10.8 demonstrates the behavior of this interaction technique. Multiple evenly spaced cursors are generated as samples from a special Probabilistic Ninja Cursor event. To disambiguate multiple inputs, Kobayashi changed the behavior of cursors to ensure that no more than one cursor was inside a button at a time. Our approach is slightly different: Button press likelihood is proportional to the proximity of a cursor to the center of the button. When button press likelihoods are too close, the mediator defers action, and an n-best list is displayed to the user, allowing him to disambiguate (Figure 10.7).

Figure 10.8 Demonstration of Ninja cursors. Rather than a single cursor, the user has a grid of cursors. The user wishes to select the target outlined in green (circle with solid line). Several other cursors overlap this target (dashed and dotted lines). After clicking, the toolkit selects the most likely interface, and shows the three most likely alternatives (dashed lines). The least likely alternative (dotted line) is not shown.

Note that this implementation allows for likelihoods to be adjusted by more than just proximity to button center. For example, eye gaze can be used to further adjust likelihoods. Other context such as previous actions may also be used.

Once again, while the resulting interaction is complex, the implementation of this interaction technique is fairly simple, requiring only a small modification to the default button interactor to add a likelihood to the transition of the mouse click event, and the creation of a new NinjaCursor mouse event. This demonstration is yet another example of how novel and fairly complex interaction techniques can be implemented using very little extra work. Once again, developers do not need to worry about the actual tracking of alternatives, rendering alternatives, or disambiguation. This is all handled by the toolkit.

This demonstration was implemented using the third iteration of the JULIA toolkit. For Ninja cursors, the number of event samples equaled the number of Ninja Cursors desired. In our case, this was 25 evenly spaced cursors. Interface samples were reduced to 10. In total, the demonstration was written in about 350 lines of JavaScript code.

10.1.8 N Best List for Gestures

We also re-implemented the disambiguation method for gestures mentioned in (Mankoff 2001) to further show the generality of our approach. Specifically, we built an n-best list for a gesture recognizer. Gesture recognizers can be notoriously inaccurate, and when these gestures perform actions that are difficult to back out of, this can be very frustrating. To avoid this, we built an n-best list for gesture recognizers which allows users to disambiguate their gesture if several gesture interpretations are likely. Figure 10.9 demonstrates the behavior.

We implemented a simple shape drawing application where users draw shapes *via* gesture (circle, triangle square). When a user draws a gesture, the most likely shape is drawn to the screen. If multiple interpretations are likely, alternate interfaces containing results

Figure 10.9 Example of N Best list for gestures. User executes a circle gesture (A-C), several interpretations are likely. The most likely interface is displayed by default, and possible alternatives are displayed as an n best list (D). The user may then select one of the alternate interpretations, or continue interacting using the most likely interpretation.

of other gestures (in our application, drawing different shapes) are displayed. The user may then switch to an alternate interpretation, or continue their interaction if the current shape is correct.

A JavaScript implementation of the $1 recognizer is used (Wobbrock 2007). When a gesture is recognized, a final action request is sent. The mediator used here executes the final action in an alternate interface and presents the result. This example was implemented with 20 samples per event and at most 10 interface samples, and demonstrates once again the versatility of the toolkit.

10.1.9 Bayesian Drawing: Leveraging Prior Probabilities to Adjust Future Action Likelihood

As a final demonstration of how prior actions can be used to adjust the likelihood of future actions, I discuss a behavior implemented in the flagship application (graphical object editor) in Section 10.3.2. Specifically, this application guesses the shape a user is drawing based on their motion and past actions

Users draw shapes or lines by dragging out the bounding box of the shape. Horizontal, vertical, and unconstrained lines may be drawn, in addition to rectangles and ellipses. If the user dwells in the middle of an interaction, an n-best list pops up, allowing the user to disambiguate. Likelihood of shapes and lines drawn depends on motion: horizontal motions reduce the likelihood of vertical lines, and vice versa. Additionally, a user's last action adjusts the likelihood for their next actions. If a user drew an ellipse last, an ellipse is more likely.

The implementation again for this interaction is fairly simple. Lines and shapes are separate interactors, they are placed in a container that dispatches to each child with equal likelihood (this is part of JULIA's standard library of interactors). The line recognizer adjusts its likelihood for horizontal/vertical lines according to line motion. Both the line and shape recognizers have probabilistic transitions to the down state, which examine the last action and adjust likelihood accordingly.

This demonstration was implemented in JavaScript using the third iteration of the toolkit. 20 samples per event were generated, leaving at most 10 alternatives at the end of each dispatch step. This demonstration shows how to incorporate priors into application logic. Once again, the framework does all of the heavy lifting.

10.2 Feedback Techniques

In addition to implementing several new interaction techniques, I have also implemented six feedback techniques: three existing, and three new, to demonstrate the versatility of the feedback system implemented.

10.2.1 Side Views

The N-Best list fusion technique discussed in Section 7.5.2 provides an implementation of a Side Views-like interaction. Side View is a technique described by Terry *et al.*(Terry & Mynatt 2002) for providing persistent, on-demand previews of commands. One improvement my approach provides over side views is the ability to communicate the likelihood of alternatives, as can be seen in Figure 10.10. In particular, the feedback techniques use opacity, scale and blur to communicate the likelihood of alternatives.

10.2.2 Octopocus

To demonstrate the versatility of the Overlay feedback method, we implemented an Octopocus-like feedback technique published by Bau *et al*. A similar interaction technique is also described in Bennet *et al.*'s paper on SimpleFlow (Bennett *et al*. 2011). For background, the Octopocus interaction technique aims to help users learn gestures by overlaying all possible gesture completions on the screen as a user is performing a gesture, so that users may better understand how to move their finger to complete a

Figure 10.10 Examples of N-Best lists presented throughout this thesis, aggregated for clarity. N-Best lists provide a similar functionality as Side Views.

certain action. The state machine for the Octopocus interactor is presented in Figure 10.11. The likelihood of each gesture is adjusted based on the likelihood of each gesture at each point in time. This likelihood is in turn used to adjust the likelihood of transition probabilities for each of the possible gesture interpretations.

In addition to the reimplementation of the above existing techniques, several novel feedback techniques are presented in Chapter 9, including rendering possible future states, animating between alternatives, and having side views highlight only the changed interface states.

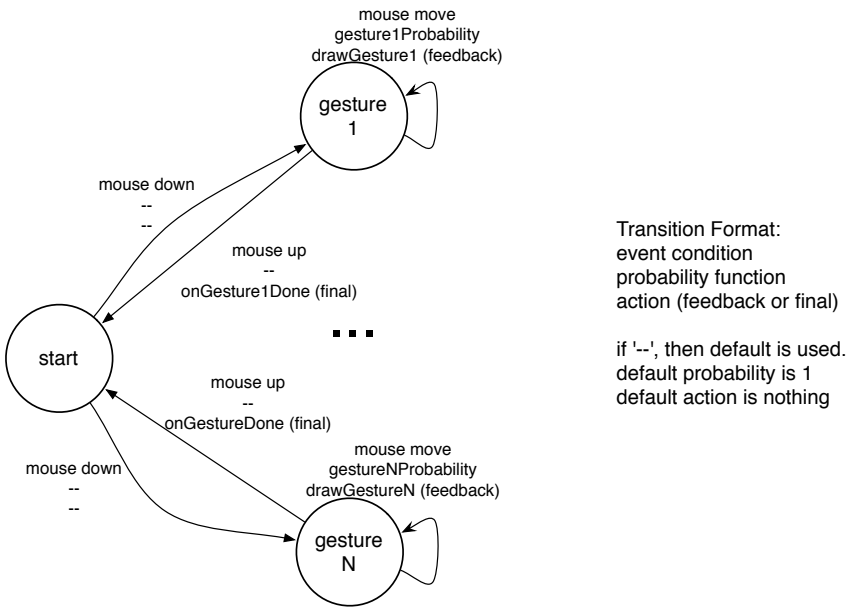


Figure 10.11 Example of Octopocus-inspired interaction technique of overlaying multiple alternate interpretations using opacity (left), blur (middle) and saturation (right) to reflect probability. Bottom: state machine for Octopocus-inspired interaction technique.

10.3 Applications

In addition to the interaction techniques presented above, I implemented three more complex applications using the probabilistic input architecture. These applications demonstrate the scalability of the approach in terms of interface complexity. Not only is the probabilistic architecture able to appropriately track interface alternatives and likelihoods, but the development burden on developers is similar to what would be required for conventional applications.

10.3.1 GesturePaint

GesturePaint was developed for the second iteration of the JULIA toolkit. This painting application demonstrates carrying of uncertainty from the sensor, to input interpretation, and finally action level. Additionally, the application delays action until it is certain of an interpretation. Below I briefly describe several of the interactors used in this application.

All of my example interactors were written in very much the same form as a conventional interface, without having to think or code probabilistically. The programmer doesn't have to be concerned with probabilities because the system does it for them in a nice, transparent way. The GesturePaint application itself is written entirely without regard to uncertain events, and is a relatively simple application consisting of roughly 400 lines of code with the same setup code and logic as any standard paint application.

My paint application allows users to "stamp" (add onto the canvas) images onto their painting. To support this I developed a stamp interactor that can be both moved and pressed. The stamp interactor uses three states to accomplish this. The state machine has two state properties – the drag start position and the touch ID (to support multitouch).

Figure 10.12: Screenshots of feedback provided by stamp interactors. The dotted circles have been added to indicate the position of the user's finger, and the numeric probability displays at the left are for illustration, but would not normally appear in such an interface. Left: Press feedback: A: Stamp not pressed. B: Stamp depressed completely when press is unambiguous. C: Stamp depressed partially when press is ambiguous. Right: Move feedback. A: User pressed in between the house and tree when beginning their drag, overlapping the tree more than the house. B: Both the house and tree are shown, with the tree being less transparent than the house reflecting overlap difference.

Stamps provide feedback to indicate the likelihood that they are being pressed vs. moved. Stamps provide press feedback by manipulating the shadow to make the interactor appear depressed in proportion to the likelihood that they are pressed (Figure 10.12, left). To provide feedback about moving, stamps show a 'ghost' version of the moved stamp. The transparency of the ghost stamp is based on the stamp's move likelihood (Figure 10.12, right). Sometimes multiple stamps might be selected or moved ambiguously (because the initial touch overlapped both interactors). To accommodate this, each stamp uses an alpha value corresponding to its move likelihood, which helps the user to see what the system thinks is happening (Figure 10.12, right).

This has the advantage that the user can back out of an incorrect interpretation before any final actions are invoked (by moving the stamp away from and then back to its original position). Importantly, I do not explicitly support this escape mechanism: it is a natural solution that arises from an understanding that dragging and pressing are both possible.

The specific feedback used by the stamp class is different for different transitions. However, all of them use likelihood as a drawing parameter (for shadow size, transparency, etc.). Outside of this parameter, the stamp has no other code that uses probabilities.

Stamps demonstrate the impact of uncertainty about which exact screen location the user intends to touch, what direction the user is moving, what interactor is being targeted, and so on. In these examples uncertainty arises directly from the properties of individual input events (at the sensor level). Another source of uncertainty is recognized input, as in the case of gesture recognition. Gestures that are prefixes of one another (such as 'c', 'g' and circle gestures) are especially problematic. Many applications are designed to avoid common prefixes or provide sophisticated feedback because of the resulting high degree of ambiguity.

My framework handles this sort of uncertainty without requiring any special effort by the developer. By default, the recognizer is called repeatedly after each new input event (*e.g.*, TOUCH_MOVE) arrives. Each time, it generates a probabilistic gesture event which contains a distribution specifying the probability that each possible gesture is the correct in-

Figure 10.13: Screenshots of the canvas interactor as user draws a figure 8. The "c" and circle gesture share a common prefix. Dotted line indicates finger location. Left: Initially a 'c' gesture is most likely (probability 0.47). Feedback indicates that c is the most likely interpretation, though a circle is possible. Middle: The system becomes confident that the gesture is a circle, reflecting this in feedback (also evident in state distribution). Right: As the user completes the figure 8, the canvas believes the user is painting and removes all gesture feedback.

terpretation. During event sampling, this event is divided into event samples for individual gestures (*e.g.*, a circle gesture, a 'c' gesture, etc.), weighted by likelihood.

I developed a canvas interactor for my paint application that handles both 'moving' (for painting) and gesture events. Figure 10.14 shows the gestures the canvas recognizes. The canvas interactor has 9 states (one for each recognized gesture as well as a start and moving state), and 37 transitions. The canvas provides feedback about the canvas state (paint and interpreted gestures) using transparency. Figure 10.13 illustrates what happens when the user paints an 8 shape on the canvas. Because 8 shares a common prefix with 'c' and 'circle', the top hypothesis shifts from 'c' to 'circle' to painting on the canvas as the user draws. The canvas interactor demonstrates the toolkit's support for recognition ambiguity: this interactor supports recognition of gestures with common prefixes without any additional complex logic to handle ambiguity.

Developers working in a conventional input handling framework could certainly implement this canvas, however they would need to track not only the gesture probabilities, but also would need to include logic to determine when to decide whether the user is painting or gesturing. In my framework, the developer handles all of this simply by including both gestures and raw down/move/up events on transitions in the canvas's state machine. The underlying system handles all logic relating to tracking probabilities and deciding between paint and gesture events.

Each of these interactors is interesting individually, but the interactions become even more interesting when the interactors are combined in an application. In addition to ambiguity about touch location and gesture, for any user action it is always unclear whether a user intends to paint on the canvas, execute a gesture, click on a button, or move a stamp. The success of the paint application hinges on its ability to manage multiple alternative interpretations across multiple interactors for as long as possible (i.e., until the user lifts his/her finger, at which point the system needs to act). The framework I developed ensures that the application gives appropriate feedback about each possible action to the user. When the user lifts her finger, the framework acts appropriately (either resolving input when the resulting action is clear or prompting the user to disambiguate). No changes are required to the interactors described above for this to happen: Thanks to my framework, this complex application simply works.

Figure 10.14: Gestures recognized by the Canvas Interactor

The power of my framework is clear when all of these interactors are combined together. While it might be feasible to write these interactors individually using a conventional input framework, writing an application that correctly handles uncertainty across interactors would be extremely difficult, and not reusable. For example, in a hand built solution, it would be difficult to remove one type of interactor (say, a stamp) and replace it with another, whereas my framework handles this *via* cancellation and acceptance of actions. The JULIA toolkit provides a general method for handling uncertain input, tracking interactive state and resolving ambiguous actions.

This implementation was built using the second iteration of the JULIA toolkit, and used about 20 event samples for every touch input event. The application totaled about 150 lines of application code (for interactor setup and event handling, this doesn't include the code to describe interactor behavior).

10.3.2 Graphical Object Editor

The graphical object editor is a drawing tool inspired by Igarashi's work on interactive beautification (Igarashi & Matsuoka 1997) and Zeleznik's Lineogrammer (Zeleznik *et al*. 2008). It also leverages concepts from Lunzer's subjunctive interfaces work (Lunzer & Hornbæk 2008). The Graphical Object Editor was developed for the third iteration of the JULIA toolkit. In addition to the features demonstrated by GesturePaint in section 10.3.1, this application also demonstrates incorporation of prior action into future behavior, proper maintenance of interface alternatives, more complex and nuanced interactions, generations of new events in response to existing events, and rendering of feedback to allow for disambiguation. Below I describe the application behavior and implementation.

The graphical object editor is a diagramming application that allows for drawing and manipulation of shapes and lines. Users may draw rectangles or ellipses using free form gestures (*e.g.* draw a circle to make a circle) or by dragging out a bounding box. They may also draw straight lines: unconstrained, horizontal or vertical. The application tracks a user's previous action and adjusts the likelihood of shapes accordingly such that the most recently drawn shape is most likely (all other things being equal).

Shapes and lines may be dragged and resized. When multiple actions are possible, the toolkit tracks all options and their likelihoods. A line's endpoints snap to control points on shapes. If multiple snap points are possible, all alternative shapes are tracked. Items can be removed by being dragged to the bottom of the screen. When being dragged, appropriate feedback shows up indicating the removed region. Of course, the user may also simply wish to place an item near the bottom of the screen. Once again, both alternatives are tracked. When a user pauses or dwells, an n-best list appears showing the most likely interpretations. Users can then disambiguate their intent by tapping on an alternative.

Many of the controls in this application are actually borrowed from the interaction techniques presented earlier. For example, drawing lines are the same as presented in section 10.1.9. Drawing shapes by freehand is done using the same gesture recognizer in section 10.1.8. Drawing shapes by specifying the bounding box is nearly identical to lines, the only differences being in the feedback rendered, the resulting event handlers, and in the removal of logic that considers horizontal and vertical lines.

The interactors for the ellipse and rectangle shapes are similar to the window interactor in section 10.1.3. Lines have similar logic. In addition to being resized and dragged, lines can also snap to different geometry. To do this, the line queries all targets. For every snappable point, a new action request is generated, updating the line endpoint to the snap position. After a dwell timeout, JULIA renders feedback, which is an n-best list as described in 9.5.3.

One interactor is custom for this application, however: the 'remove' button. The remove button only appears when a draggable item is selected. When a draggable item is dropped over the remove button, that item is removed from the canvas. This is implemented through the generation of new events in response to drag begin and drag end interactions. After drag interaction begins, any draggable object sends a new DragBegan event. This injecting is performed through special call in the JULIA toolkit. However, under the covers the dispatch logic for this call is almost identical to normal dispatch. The only difference is this event is not dispatched to all possible interfaces, just to the specific interface called from. The trash button then handles these drag began and drag end events. Note that this behavior does not happen on all interface alternatives, only the ones where an element is actually being dragged.

Once again, much of the complex behavior in this application: tracking of alternatives, disambiguation, is built into the toolkit. Many interactors here have already been implemented, the power comes from when everything is combined. The success of the graphical object editor hinges on its ability to manage multiple alternative interpretations for as long as possible, and to easily be able to disambiguate between alternatives. The probabilistic architecture ensures that the application gives appropriate feedback about each possible action to the user. When the user lifts her finger, the framework acts appropriately (either resolving input when the resulting action is clear or prompting the user to disambiguate). No changes are required to the interactors described above for this to happen.

Figure 10.15: Illustration of a few features of the graphical object editor. Gray dot indicates touch location. A: When a user drags on an empty area, he may create either a line, rectangle, or ellipse. The most recent shape is weighted to be more likely. B: Line endpoints may be moved, they snap to different endpoints. Multiple endpoint candidates may be chosen (hollow circles in alternative views), or the user may continue free manipulation (solid circle). C: When the intended target of a drag is ambiguous (first alternative shows an alternate line, second alternative shows creating a new line, third shows a snap), alternatives are shown. D: Lines may be dragged to the bottom to be removed, or (as the alternative shows), a user may opt to just move the line to the bottom of the screen. E: Boxes may be moved or resized, the main interface shows drag feedback, alternative shows resize feedback. F: When a box is very small, all alternatives are shown, allowing the user to disambiguate.

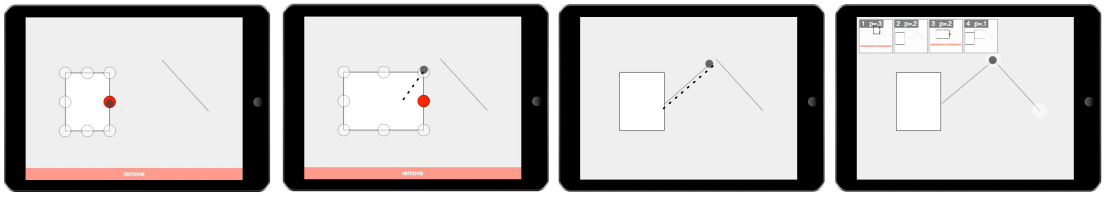


Figure 10.16: Example of user dragging a line from the edge of a box (dotted line shows drag). User begins drag, initially belief is box resize. Eventually, likelihood of box resize diminishes, line draw is next. After a pause, all alternatives are shown.

As an example, consider what happens when a user starts dragging on the edge of a box (Figure 10.16). Several interpretations are possible: dragging the rectangle, resizing the rectangle, drawing a line, drawing a rectangle, drawing an ellipse. All alternatives are considered, the most likely, a drag, is displayed. As the user continues dragging upward, the horizontal resize becomes less likely. The user pauses, and all alternatives are displayed on the screen. As the user continues, the updates dynamically update. Finally, the user selects the alternative for a line, and continues drawing, now with a line visible. When the user releases his finger, an OnLineCompleted handler is called, which adds a new line object to the screen. Note that this is the same interaction as the Window resizing example presented in section 10.1.3.

This type of interaction would be very difficult to implement in a conventional user interface framework. In fact, one would need to build something almost as complex as the probabilistic input system itself to properly handle all cases (such as, for example, the remove button appearing only when draggable items are selected). In contrast, this application was written largely without regard to probability, and its fairly complex operation was described in only a few hundred lines of JavaScript. As with other applications, this application could be run with just 20 samples per event, and up to 10 alternatives tracked at once. This application demonstrates the power of the probabilistic input architecture: it enables development of complex interfaces without requiring complex programming.

10.3.3 Backwards compatibility

The JULIA toolkit can be used to simulate traditional, deterministic input. This is merely a matter of disallowing nondeterministic state machines and only having one event sample. In this situation, only one interface alternatively is maintained and updated. In this representation, specification of the state machine, as well as dispatch, transforms into exactly the same algorithms that are used to dispatch conventional input.

10.4 Platforms

To demonstrate that the probabilistic input architecture is language and platform independent, we implemented different iterations of the architecture in three different platforms: a Windows desktop platform, the Windows Phone platform, and a web platform (Chrome, JavaScript + SVG). Additionally, the architectural concepts used were refined in every interation, so that the final iteration represents over three iterations of successive improvements.

The first iteration of the JULIA toolkit (Schwarz *et al*. 2010) was developed in C# to run on the Windows 7 platform. It was developed using the Windows Presentation Foundation (WPF) toolkit. The toolkit ran on a Dell XT2 touchscreen tablet with an Intel Core 2 Duo 1.2 GHz processors and 3GB of RAM. I chose a laptop mostly for convenience of development and the availability of touch input. Many interactions were touch interactions in a desktop environment, which is often a very challenging interface.

The second iteration of the JULIA toolkit (Schwarz *et al*. 2011) was built for the Windows Phone OS on top of the XNA game framework, which has a primitive input handling system supporting only polling for input but not events. I chose the phone because touch input is a widely used medium that contains a large amount of uncertainty (i.e., the intended location of the touch event is uncertain). The phone also illustrates that my approach works on systems with relatively low amounts of memory and processing power. The development and testing for this version of the tool was done on a phone with 512MB of RAM and a 1GHz processor, which compared to typical modern desktop or laptop machines is quite limited (and even current top of the line phones). I support touch, gesture, and accelerometer-based shake events. The framework has about 2,000 lines of C# code, and the 10 demos I wrote totaled about 1,000 lines of C# code.

The third iteration is implemented in JavaScript and the HTML5 APIs. The toolkit hooks into existing touch event handlers, overriding them with probabilistic event handlers and dispatching input *via* the JULIA toolkit. The JavaScript platform was chosen to illustrate the versatility of the toolkit and make the resulting toolkit and demos easier to share. The implementation supports mouse, keyboard, touch, and voice input. Finally, this implementation demonstrates that the probabilistic architecture can be implemented on a universally used platform: the web browser. The development and testing for the third iteration was done on a computer with 16 GB of RAM and a quad core, 2.3GHz processor. This toolkit supports touch, mouse, keyboard, and voice recognition events (from the Google Chrome Voice API). The toolkit and demos were developed and tested in the Chrome browser. The toolkit (including the library of interactors and feedback objects) totals to about 8,000 lines of uncompressed JavaScript code, and the 14 demos written total about 1,600 lines of JavaScript code. Collectively, these implementations demonstrate that the architecture is platform and language agnostic.

11 CONCLUSION

This dissertation has presented a new user interface architecture that treats user input as an uncertain process, approximates the probability distribution over possible interfaces using Monte Carlo sampling, and provides tools for interface developers to easily build probabilistic user interfaces. Importantly, alternate interfaces (and their likelihoods) are managed by the architecture itself, allowing for interface developers to reap the benefits of probabilistic interfaces without needing to think probabilistically. To recap, the contributions of this thesis are as follows:

- A new architecture for modeling and dispatching uncertain user input, using Monte Carlo sampling to approximate the probability distribution over possible interface states.

- A system for generating fluid interactive feedback which continuously communicates alternative (or future) input interpretations as a user is interacting and allows her to disambiguate intent.

- An API and library which allows user interface developers to easily build probabilistic interfaces. This includes a mechanism for specifying interactor behavior using probabilistic state machines, a rich set of classes for developing interactive feedback, and a built-in mechanism for adjusting likelihood of future actions based on past input.

- A large collection of interaction techniques and applications which demonstrate the versatility and power of the toolkit. These include demonstrations of how to leverage prediction to accelerate interaction, how to build interfaces that allow users to easily switch between multiple input interpretations, and how to perform Bayesian inference to adapt interface behavior.

This architecture provides the foundation for a new era of user interfaces which handle uncertainty gracefully to better infer user intent.

11.1 Limitations

Several limitations of the work are alluded to throughout the thesis, warranting a more careful discussion. These limitations fall into two categories. First, the current toolkit requires significant CPU and memory resources. Second, the usability of many of the techniques, along with the APIs used, has not been tested.

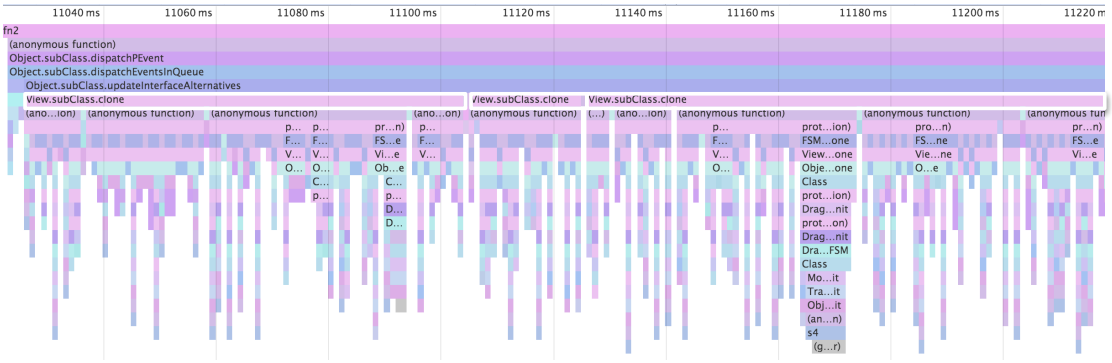


Figure 11.1: CPU profile of dispatch process obtained during dispatch of a single input event. Most time is spent in the View.subClass.clone method, cloning interface alternatives (rectangular boxes outlined in red).

One drawback of our approach is that large numbers of samples are often required to represent distributions where the state space is large: when input events have many variables, or variables that can have many values. In many cases, the state space for user interfaces is not that large, and thus tens of samples are often sufficient (as is demonstrated by our interaction techniques and applications). However, the current approach breaks down for cases with large amounts of ambiguity where hundreds or even thousands of samples are required.

This dissertation work focused on developing early concepts in using Monte Carlo techniques for managing uncertainty, and did not focus heavily on optimizations to allow for simulation of large numbers of samples. However, many optimizations are possible. A CPU profile of toolkit in operation (Figure 11.1) reveals that most time is spent cloning interfaces. Several optimizations are possible here. First, each dispatch sequence for a (event sample, interface sample) pair is independent of others, meaning we can execute update operations in parallel. Additionally, rather than copying all interface properties, we can use incremental data structures (Tanin *et al*. 1996) to store interface alternatives. This will not only reduce the time required to clone alternatives but will also reduce the memory footprint. Many other optimizations are possible and left for future work.

A second limitation of this work is that the interaction techniques and applications presented in this dissertation were not evaluated for usability. While our validation by demonstration is suitable for the contribution of this work, the practicality of the demonstrations presented remains unknown. Many interaction techniques (such as animating between alternate versions of interfaces), are interesting but questionable in terms of usability. Further investigation is needed into the usability of the interaction techniques presented.

Additionally, the developer-facing API (probabilistic state machines) was not rigorously validated. While an informal validation based on number of interaction techniques and code simplicity is presented, a more formal evaluation is needed before the approach can be determined as ready for practical use. These evaluations will likely point out problems that will lead to improvements, which will in turn be their own valuable contributions.

Despite these limitations, the work presented in this thesis is a great leap forward in the domain of architectures for handling uncertain input, and the demonstrations provided show the feasibility of the Monte Carlo approach—a surprising and positive result.

11.2 Future Work

The probabilistic input architecture lays the foundation for an entirely new type of user interface architecture, however this is just the tip of the iceberg. Most of the work needed to bring this architecture into practice lies before us. Several avenues of future work present themselves: improving performance of the toolkit, developing more complex interfaces, and building more sophisticated features.

In addition to the performance improvements mentioned in the previous section, the interfaces in the demos and applications, while complex, still do not approach the complexity of most real applications. A further avenue of future work may be to track multiple alternative program states, to remove the need to decide on all final actions. Finally, the probabilistic input architecture lends itself nicely to input prediction and Bayesian simulation. A more general method that allows for systematic integration and interface improvement would be interesting to explore. In short, there's an enormous body of work to explore, and the future of probabilistic input is a promising field that I hope to see flourish in the near future.

12 REFERENCES

53Designs, 2012. Paper. Available at: http://www.fiftythree.com/paper.

Abowd, G.D., 1992. *Formal aspects of human-computer interaction*. Oxford University.

Abowd, G.D. & Dix, A.J., 1994. Integrating status and event phenomena in formal specifications of interactive systems. *ACM SIGSOFT Software Engineering Notes*, 19(5), pp.44–52.

Arvo, J. & Novins, K., 2000. Fluid Sketches : Continuous Recognition and Morphing of Simple Hand-Drawn Shapes. *UIST '00*, 2, pp.73–80.

Balakrishnan, R., 2004. "Beating" Fitts' law: virtual enhancements for pointing facilitation. *International Journal of Human-Computer Studies*, 61(6), pp.857–874.

Bau, O., 2010. *Interaction streams*. University of Paris.

Bau, O. & Mackay, W.E., 2008. OctoPocus: a dynamic guide for learning gesture-based command sets. In *UIST '08*. New York, NY, USA: ACM, pp. 37–46.

Bellegarda, J. et al., 2012. Method, device, and graphical user interface providing word recommendations for text input.

Bennett, M. et al., 2011. Simpleflow: enhancing gestural interaction with gesture prediction, abbreviation and autocompletion. In *INTERACT'11*. London, UK: Springer-Verlag, pp. 591–608.

Bier, E.A. et al., 1993. Toolglass and magic lenses. In *SIGGRAPH '93*. New York, New York, USA: ACM Press, pp. 73–80.

Bohus, D. & Horvitz, E., 2009. Learning to Predict Engagement with a Spoken Dialog System in Open-World Settings. In *SIGDIAL '09*. New York, NY, USA: ACM Press, pp. 244–252.

Bolt, R.R.A., 1980. "Put-that-there": Voice and gesture at the graphics interface. In *SIGGRAPH '80*. New York, NY, USA: ACM Press, pp. 262–270.

Bourguet, M.-L., 2006. Towards a taxonomy of error-handling strategies in recognition-based multi-modal human–computer interfaces. *Signal Processing*, 86(12), pp.3625–3643.

Bragdon, A. et al., 2009. GestureBar: improving the approachability of gesture-based interfaces. In *CHI '09*. New York, NY, USA: ACM, pp. 2269–2278.

Cohen, P.R. et al., 1997. QuickSet: multimodal interaction for simulation set-up and control. In *ANLC '97*. Morristown, NJ, USA: Association for Computational Linguistics, pp. 20–24.

Eisenstein, J. & Davis, R., 2005. *Gestural Cues for Sentence Segmentation*, Boston, MA.

Eisenstein, J. & Davis, R., 2006. Gesture improves coreference resolution. In *NAACL '06*. Stroudsburg, PA: Association for Computational Linguistics, pp. 37–40.

Eisenstein, J. & Davis, R., 2004. Visual and linguistic information in gesture classification. In *ICMI '04*. New York, New York, USA: ACM Press, pp. 113–120.

Frankish, C., Hull, R. & Morgan, P., 1995. Recognition accuracy and user acceptance of pen interfaces. In *CHI '95*. New York, NY, USA: ACM Press, pp. 503–510.

Freeman, D. et al., 2009. ShadowGuides : Visualizations for In-Situ Learning of Multi-Touch and Whole-Hand Gestures. In *ITS '09*. New York, NY, USA: ACM Press, pp. 165–172.

Goodman, N. et al., 2012. Church: a language for generative models. *Uncertainty and Artificial Intelligence*, pp.220–229.

Google, 2013. Google Glass. Available at: http://www.google.com/glass/start/.

Gordon, N.J.J., Salmond, D.J.J. & Smith, A.F.M., 1993. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEEE proceedings. Part F. Radar and signal processing*, 140(2), pp.107–113.

Grossman, T. et al., 2006. Hover Widgets : Using the Tracking State to Extend the Capabilities of Pen-Operated Devices. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*. New York, NY, USA: ACM Press, pp. 861–870.

Hammersley, J.M. & Handscomb, D.C., 1964. Monte Carlo Methods S. Kotz & N. L. Johnson, eds. *Annals of Statistics*, 44(247), pp.335–341.

Harrison, C. & Hudson, S.E., 2009. Abracadabra: wireless, high-precision, and unpowered finger input for very small mobile devices. In *UIST '09*. New York, New York, USA: ACM Press, pp. 121–124.

Harrison, C., Schwarz, J. & Hudson, S.E., 2011. TapSense: Enhancing Finger Interaction on Touch Surfaces. In *UIST '11*. New York, New York, USA: ACM Press, pp. 627–636.

Henze, N., Rukzio, E. & Boll, S., 2011. 100,000,000 taps: analysis and improvement of touch performance in the large. In *MobileHCI '11*. New York, New York, USA: ACM Press, p. 133.

Hinckley, K. et al., 2010. Pen + Touch = New Tools. In *UIST '10*. New York, New York, USA: ACM Press, pp. 27–36.

Hinckley, K. et al., 2012. Touch and stylus discrimination and rejection for contact sensitive computing devices.

Hofstadter, D.R., 1979. *Godel, Escher, Bach: An Eternal Golden Braid*, New York, NY, USA: Basic Books, Inc.

Holz, C. & Baudisch, P., 2010. The generalized perceived input point model and how to double touch accuracy by extracting fingerprints. In *CHI '10*. New York, NY, USA: ACM Press, pp. 581–590.

Holz, C. & Baudisch, P., 2011. Understanding touch. In *CHI '11*. New York, New York, USA: ACM Press, pp. 2501–2510.

Horvitz, E., 1999. Principles of mixed-initiative user interfaces. In *CHI '99*. New York, New York, USA: ACM Press, pp. 159–166.

Horvitz, E. et al., 1998. The lumière project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann Publishers Inc., pp. 256–265.

Huckauf, A. et al., 2005. What you don't look at is what you get. In *Proceedings of the 2nd symposium on Appied perception in graphics and visualization*. New York, New York, USA: ACM Press, pp. 170–170.

Igarashi, T. et al., 1998. Pegasus. In *CHI '98*. New York, New York, USA: ACM Press, pp. 24–25.

Igarashi, T. & Hughes, J.F., 2001. A suggestive interface for 3D drawing. In *Proceedings of the 14th annual ACM symposium on User interface software and technology - UIST '01*. New York, New York, USA: ACM Press, pp. 173–181.

Igarashi, T. & Matsuoka, S., 1997. Interactive beautification: a technique for rapid geometric design. In *UIST '97*. pp. 105–114.

Istance, H. et al., 2008. Snap clutch, a moded approach to solving the Midas touch problem. In *Proceedings of the 2008 symposium on Eye tracking research applications ETRA 08*. pp. 221–228.

Jacob, R., Deligiannidis, L. & Morrison, S., 1999. A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1), pp.1–46.

Kaiser, E. et al., 2003. Mutual disambiguation of 3D multimodal interaction in augmented and virtual reality. In *Proceedings of the 5th international conference on Multimodal interfaces - ICMI '03*. New York, New York, USA: ACM Press, pp. 12–19.

Kettebekov, S., Yeasin, M. & Sharma, R., 2005. Prosody based audiovisual coanalysis for coverbal gesture recognition. *IEEE Transactions on Multimedia*, 7(2), pp.234–242.

Lalanne, D. et al., 2009. Fusion engines for multimodal input: a survey. In *International Conference on Multimodal Interfaces*. New York, New York, USA: ACM Press, pp. 153–160.

Latoschik, M.E., 2005. A user interface framework for multimodal VR interactions. In *International Conference on Multimodal Interfaces*. New York, New York, USA: ACM Press, pp. 76–83.

Li, Y., 2009. Beyond Pinch and Flick : Enriching Mobile Gesture Interaction. *Computer*, 42(12), pp.87–89.

Liu, J., Wong, C.K. & Hui, K.K., 2003. An adaptive user interface based on personalized learning. *IEEE Intelligent Systems*, 18(2), pp.52–57.

Lü, H., 2013. Gesture Studio : Authoring Multi-Touch Interactions through Demonstration and Declaration. In *CHI '13*. ACM Press, pp. 257–266.

Lunzer, A. & Hornbæk, K., 2008. Subjunctive interfaces. *ACM Transactions on Computer-Human Interaction*, 14(4), pp.1–44.

MacEachren, A.M., 1992. Visualizing uncertain information. *Cartographic Perspective*, 13(13), pp.10–19.

Mankoff, J., 2001. *An Architecture and Interaction Techniques for Handling Ambiguity in Recognition-Based Input*. Georgia Institue of Technology.

Mankoff, J. et al., 2000. OOPS: a toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces. *Computers & Graphics*, 24(6), pp.819–834.

Martin, J.-C., Veldman, R. & Béroule, D., 1998. Developing Multimodal Interfaces: A Theoretical Framework and Guided Propagation Networks. In *Multimodal Human-Computer Communication, Systems, Techniques, and Experiments*. London, UK: Springer-Verlag, pp. 158–187.

McCulloch, W.S. & Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), pp.115–133.

Metropolis, N. & Ulam, S., 1949. The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247), pp.335–341.

Moscovich, T., 2009. Contact area interaction with sliding widgets. In *UIST '09*. New York, New York, USA: ACM Press, p. 13.

Murugappan, S. et al., 2012. Extended Multitouch : Recovering Touch Posture , Handedness , and User Identity using a Depth Camera. In *UIST '12*. New York, NY, USA: ACM Press, pp. 487–496.

Newman, W.M., 1968. A system for interactive graphical programming. In *AFIPS '68*. New York, New York, USA: ACM Press, pp. 47–54.

Nielsen, J., 1993. *Usability Engineering*, Morgan Kaufmann.

Olwal, A. & Feiner, S., 2003. SenseShapes : Using Statistical Geometry for Object Selection in a Multimodal Augmented Reality System. In *ISMAR '03*. Washington, DC, USA: IEEE Computer Society, pp. 300–301.

Oviatt, S., 1996. Multimodal interfaces for dynamic interactive maps. In *CHI '96*. New York, New York, USA: ACM Press, pp. 95–102.

Oviatt, S., 1999a. Mutual disambiguation of recognition errors in a multimodal architecture. In *CHI '99*. New York, New York, USA: ACM Press, pp. 576–583.

Oviatt, S., 1999b. Ten myths of multimodal interaction. *Communications of the ACM*, 42(11), pp.74–81.

Oviatt, S.L. & Cohen, P.R., 1999. Multimodal integration-a statistical view. *IEEE Transactions on Multimedia*, 1(4), pp.334–341.

Pang, A.T., Wittenbrink, C.M. & Lodha, S.K., 1997. Approaches to uncertainty visualization. *The Visual Computer*, 13(8), pp.370–390.

Rabiner, L.R., 1989. A tutorial on hidden Markov models and selected applications in speech recognition A. Waibel & K.-F. Lee, eds. *Proceedings of the IEEE*, 77(2), pp.257–286.

Rogers, S. et al., 2010. FingerCloud: uncertainty and autonomy handover in capacitive sensing. In *Imagine*. Association for Computing Machinery, pp. 577–580.

Rosenberg, I. & Perlin, K., 2009. The UnMousePad: an interpolating multi-touch force-sensing input pad. *ACM Transactions on Graphics*, 28(3), p.65.

Schwarz, J. et al., 2010. A framework for robust and flexible handling of inputs with uncertainty. In *UIST '10*. ACM Press, pp. 47–56.

Schwarz, J., Mankoff, J. & Hudson, S.E., 2011. Monte carlo methods for managing interactive state, action and feedback under uncertainty. In *UIST '11*. pp. 235–244.

Sewell, W. & Komogortsev, O., 2010. Real-time eye gaze tracking with an unmodified commodity webcam employing a neural network. In *CHI EA '10*. New York, New York, USA: ACM Press, pp. 3739–3744.

Stewart, C. & Murray-smith, R., 2011. AnglePose : Robust , Precise Capacitive Touch Tracking via 3D Orientation Estimation. In *CHI '11*. New York, NY, USA: ACM Press, pp. 2575–2584.

Tanin, E., Beigel, R. & Shneiderman, B., 1996. Incremental data structures and algorithms for dynamic query interfaces. *ACM SIGMOD Record*, 25(4), pp.21–24.

Terry, M. et al., 2004. Variation in element and action: supporting simultaneous development of alternative solutions. In *CHI '04*. New York, NY, USA: ACM Press, pp. 711–718.

Terry, M. & Mynatt, E.D., 2002. Side views: persistent, on-demand previews for open-ended tasks. In *UIST '02*. New York, NY, USA: ACM, pp. 71–80.

Vanacken, D. et al., 2008. Ghosts in the interface: Meta-user interface visualizations as guides for multi-touch interaction. In *2008 3rd IEEE International Workshop on Horizontal Interactive Human Computer Systems*. IEEE, pp. 81–84.

Vogel, D. & Baudisch, P., 2007. Shift: a technique for operating pen-based interfaces using touch. In *CHI '07*. New York, NY, USA: ACM Press, pp. 657–666.

Wasserman, A.I., 1985. Extending State Transition Diagrams for the Specification of Human-Computer Interaction. *IEEE Transactions on Software Engineering*, SE-11(8), pp.699–713.

Weir, D. et al., 2012. A user-specific machine learning approach for improving touch accuracy on mobile devices. In *UIST '12*. New York, NY, USA: ACM Press, pp. 465–476.

Weir, D., 2012. Machine learning models for uncertain interaction. In *UIST Adjunct Proceedings '12*. New York, New York, USA: ACM Press, pp. 31–34.

Welch, G. & Bishop, G., 2006. An Introduction to the Kalman Filter. *In Practice*, 7(1), pp.1–16.

Wensveen, S.A.G., Djajadiningrat, J.P. & Overbeeke, C.J., 2004. Interaction Frogger : a Design Framework to couple action and function through feedback and feedforward. In *Proceedings of the 5th conference on Designing interactive systems DIS '04*. New York, NY, USA: ACM Press, pp. 177–184.

Wigdor, D. et al., 2009. Ripples: utilizing per-contact visualizations to improve user interaction with touch displays. In *UIST '09*. ACM, pp. 3–12.

Williamson, J., 2006. *Continuous Uncertain Interaction*. University of Glasgow.

Williamson, J., 2003. Hex: Dynamics and probabilistic text entry. In *Switching and Learning in Feedback Systems*. Berlin, Heidelberg: Springer-Verlag, pp. 333–342.

Williamson, J. & Murray-Smith, R., 2002. *Audio feedback for gesture recognition*, Department of Computing Science, University of Glasgow.

Wilson, A. & Shafer, S., 2003. XWand: UI for intelligent spaces. In *CHI '03*. New York, NY, USA: ACM, pp. 545–552.

Wobbrock, J.O., Wilson, A.D. & Li, Y., 2007. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *UIST '07*. New York, NY, USA: ACM, pp. 159–168.

Yeh, S.-T. & Chen, J.-M., 2011. Method and device for palm rejection.

Zeleznik, R. et al., 2010. Hands-on math. In *UIST '10*. New York, NY, USA: ACM Press, pp. 17–26.

Zeleznik, R.C. et al., 2008. Lineogrammer: creating diagrams by drawing. In *UIST '08*. New York, New York, USA: ACM Press, pp. 161–170.

Ziv, J. & Lempel, A., 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3), pp.337–343.