SwingStates: Adding state machines to Java and the Swing toolkit

C. Appert* and M. Beaudouin-Lafon

LRI (Université Paris-Sud & CNRS), INRIA, bât 490, 91405 Orsay, France.

SUMMARY

This article describes *SwingStates*, a Java toolkit designed to facilitate the development of graphical user interfaces and bring advanced interaction techniques to the Java platform. *SwingStates* is based on the use of finite-state machines specified directly in Java to describe the behavior of interactive systems. State machines can be used to redefine the behavior of existing Swing widgets or, in combination with a new canvas widget that features a rich graphical model, to create brand new widgets. *SwingStates* also supports arbitrary input devices to implement novel interaction techniques based, for example, on bi-manual or pressure-sensitive input. We have used *SwingStates* in several Master's level classes over the past two years and have developed a benchmark approach to evaluate the toolkit in this context. The results demonstrate that *SwingStates* can be used by non-expert developers with little training to successfully implement advanced interaction techniques.

KEY WORDS: human-computer interaction, graphical user interfaces, user interface toolkits, finite state machines, Java Swing

1. INTRODUCTION

In many application areas, software systems have evolved from purely sequential systems to reactive systems: their behavior is controlled by events that can arrive at any time. For example, user interfaces react to events sent by the mouse and keyboard when the user operates them. Distributed systems react to events notifying the arrival of packets from the network. Even standalone applications may use events for inter-thread communication or simply to signal that a timer has expired.

*Correspondence to: LRI (Université Paris-Sud & CNRS), INRIA, bât 490, 91405 Orsay, France.

State machines are a well-known formalism for describing such event-driven systems: by identifying distinct states and the system's reactions to the various possible events in each state, they provide a simple yet powerful model for modeling the behavior of reactive and interactive systems. Indeed, state machines are widely used in the design stages of a computer system. For example, the well-known Unified Modeling Language (UML) [1] uses state machines to describe the dynamic behavior of individual objects, use cases or entire systems.

Surprisingly however, state machines are not integrated as a native programming construct of general-purpose programming languages such as Java or C++. As a result, developers resort to workarounds in order to implement state machines. The most frequent pattern is the use of *event handlers*, called *listeners* in Java. First, the programmer specifies a callback function (or method) that will be called when the event occurs. Second, the programmer binds this callback function or method to a specific event type, e.g. a mouse click. Since this programming pattern does not explicitly represent states, the code of the callback typically contains a series of tests on application's global variables in order to trigger the correct behavior. For example, in a user interface, the effect of a mouse click will be different if the cursor is on a menu item or in a drawing area and, in a drawing area, it will depend on which tool is currently selected.

The first main contribution of the *SwingStates* toolkit described in this paper is to use the native Java syntax to describe arbitrary state machines, almost as if they were a native language construct. Our goal was to make *SwingStates* widely available and appealing to the large community of Java developers. Therefore we did not want to customize the language, use a preprocessor or other external tools to specify state machines. Instead, we wanted developers to use the tools they are familiar with (editor, debugger, IDE, etc.) while taking advantage of the flexibility of state machines.

User interfaces are arguably the most widespread type of event-driven application and the one that suffers the most from the lack of higher-level abstractions to specify event-based behaviors. All widely used user interface toolkits such as Gtk [2], Java Swing and the native toolkits of MacOS X and Windows are based on the notion of *widget*. A widget is an interactive component that represents a button, a menu, a scrollbar, a dialog box, etc. Building a user interface consists of assembling those predefined widgets and linking them to the rest of the application through event handlers (listeners in Java). While widgets typically provide a first level of event dispatching based on the position of the cursor, i.e., a mouse click event is dispatched to whatever widget is under the cursor, the use of callbacks nevertheless creates what Myers calls a "spaghetti" of callbacks [3] with intricate dependencies. With typical real-life applications containing hundreds of widgets and thousands of event handlers, this makes the code brittle and hard to maintain.

Moreover, the widget sets of these user interface toolkits are closed: it is very difficult to create new widget types that would be more adequate for particular interactions. This is unfortunate because the human-computer interaction research community has a long history of creating novel and effective interaction techniques (see the proceedings of the ACM CHI and ACM UIST series of conferences for instance). For example, circular menus (also known as pie menus) have been known for many years to be a lot more efficient than linear menus [4]. Yet few user interface toolkits feature them. So current user interface toolkits are actually an impediment to the dissemination of advanced interaction techniques that would improve the usability of applications used by millions of users.

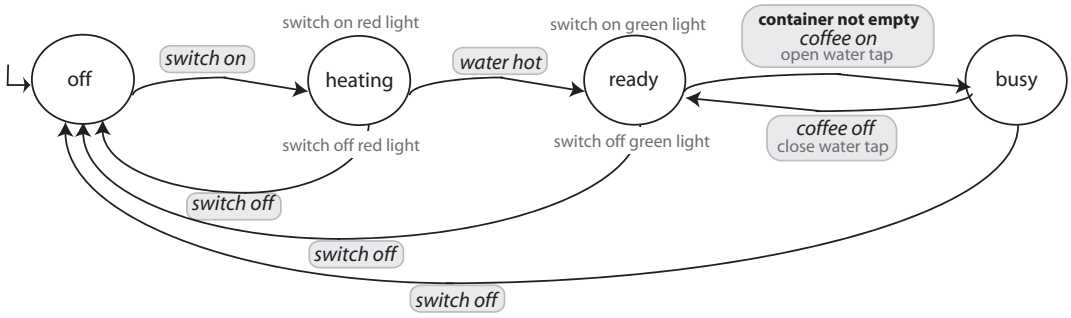

Figure 1. A graphical depiction of the state machine for a coffee maker

The second main contribution of the *SwingStates* toolkit is to extend the Java Swing widget toolkit with state machines and a new canvas widget. The new canvas widget is based on a rich structured graphical model and is designed to help create new advanced widgets such as the circular menus mentioned above. State machines are used for programming user interactions and replace listeners. They can be used with existing widgets to reprogram their behavior, as well as with the new canvas widget to program complex interactions.

The rest of this article presents *SwingStates* in detail. We describe our integration of state machines into the Java environment and illustrate how state machines differ from traditional event handlers for programming event-based applications. We then present the new canvas widget and the use of state machines both to create new widgets and to reprogram existing Swing widgets. We complete this tour of *SwingStates* by describing its support for non-standard input devices, including bi-manual input. One well-known shortcoming of state machines is the potential explosion of the number of states and transitions. We explain how to control this explosion by combining multiple state machines in various way. Finally, we report on our evaluation of *SwingStates* by showing how it follows a set of design principles and by describing our experience with using *SwingStates* for teaching and conducting a set of benchmarks on advanced interaction techniques.

2. PROGRAMMING STATE MACHINES IN JAVA

This section describes the type of state machines used in *SwingStates* and its implementation. In particular, we show how we took advantage of Java inner classes to integrate state machines as a control structure in the Java language.

SwingStates implements event-driven finite-state machines, simply called *state machines* from here on. A state machine is a model of computation consisting of a set of *states*, an *event* alphabet, and a set of *transitions* labeled with events. Exactly one state, called the *current state*, is active at any one time. Initially, the *initial state* is the current state. Figure 1 shows

a graphical representation of a state machine that models the behavior of a coffee maker. States are depicted by circles and transitions by arrows. There are four states: `off`, `heating`, `ready` and `busy`. The initial state (labeled by an L-shaped arrow) is `off`. Each transition connects an *input state* to an *output state* and may have an associated *guard* and *action*. A transition is active only when its input state is the current state and its guard evaluates to true (a guard that is omitted evaluates to true). For example, making coffee is allowed only when in the `ready` state and the water container is not empty. Each transition may have a *transition action*, which is executed when the transition is *fired*. For example, making coffee (event `coffee` from state `ready`) opens the water tap. A state may have an *enter action* and a *leave action*. Enter and leave actions are executed when a state becomes active or inactive. For example, entering the `ready` state turns the green light on while leaving it turns it off. The transitions that lead to a given state are called its *input transitions* while those that leave a state are called its *output transitions*.

When an event occurs, the first[†] output transition of the active state that is labeled with that event and whose guard evaluates to true is *fired*. If no such transition exists, the event is simply ignored. Firing a transition executes three actions in the following order:

(i) the *leave action* of the current state;
(ii) the *transition action* associated with the transition being fired;
(iii) the *enter action* of the transition's output state.

Once a transition is fired, its output state becomes the new current state.

Our goal was to seamlessly integrate state machines into Java programs in order to build on the developers' existing skills in using the Java language and its comprehensive set of tools. While a graphical format is very appealing for describing state machines and is often used for teaching and designing state machines, it did not meet our goals. Graphical editors such as StateWORKS [5] require specific tools to execute, debug and store state machines. They also require to specify the body of the actions separately from the state machine itself. This results in a poor integration with the programmers' workflow and requires significant training to use these new tools. An alternative to graphical editors is a dedicated text format for specifying state machines. For example, with the State Machine Compiler [6], developers must define state machines using a custom text format and actions must be written in Java in a separate file. State Chart XML [7] uses a similar approach, except that the text format is XML. As with graphical editors, using a text format requires at least a specific tool for translating a state machine into an executable form (either in source or object form). Forcing developers to juggle with two languages, such as Java for the actions and XML for the state machine, complicates their tasks and is error prone. Debugging also becomes a problem: either the developer has to deal with the translated form of the state machine to identify errors, or extra debugging tools must be provided.

Our solution was to use plain Java to specify state machines. Figure 2 shows the *SwingStates* code for the state machine in figure 1. The complete state machine is described by a single

[†] transitions are naturally ordered in textual representations of state machines, however this order is not specified in graphical representations such as Figure 1.

```
1  StateMachine coffeeMaker = new StateMachine() {

2    public State off = new State() {
3      Transition switchOn = new Event("switch on", ">> heating") {  };
4      Transition switchOff = new Event("switch off", ">> off") {  };
5    };

6    public State heating = new State() {
7      public void enter() {
8        // switch on red light
9      }
10     Transition waterHot = new Event("water hot", ">> ready") {  };
11     Transition switchOff = new Event("switch off", ">> off") {  };
12     public void leave() {
13       // switch off red light
14     }
15   };

16   public State ready = new State() {
17     public void enter() {
18       // switch on green light
19     }
20     Transition coffeeOn = new Event("coffee on", ">> busy") {
21       public boolean guard() { return ! containerEmpty(); }
22       public void action() { openWaterTap() }
23     };
24     Transition switchOff = new Event("switch off", ">> off") {  };
25     public void leave() {
26       // switch off green light
27     }
28   };

29   public State busy = new State() {
30     Transition coffeeOff = new Event("coffee off", ">> ready") {
31       public void action() { closeWaterTap() }
32     };
33     Transition switchOff = new Event("switch off", ">> off") {  };
34   };

35 };
```

Figure 2. The coffee maker state machine in *SwingStates*

block of Java code that can readily be run and debugged. As with most dedicated text formats, a state machine is described in *SwingStates* by a set of states and each state is described by the set of its output transitions. In *SwingStates* however, states and transitions are Java objects, declared using anonymous inner classes[‡]. In our example, we see that `coffeeMaker` is initialized with an instance of an anonymous subclass of `StateMachine` (line 1) that contains four variables representing its states: `off` (lines 2-5), `heating` (lines 6-15), `ready` (lines 16-28) and `busy` (lines 29-34). By convention, the first state of a state machine is its initial state.

Each state is itself an instance of an anonymous subclass of `State`. States can override the `enter` and `leave` methods in order to specify the state's enter and leave actions. For example, lines 17-19 and 25-27 specify that the green light is turned on/off when entering/leaving the state `ready`. States declare their output transitions using anonymous inner classes. For example, in state `ready`, the variable `coffeeOn` (line20) is initialized with an object representing the transition for the `coffee on` event. Transitions are specified as follows:

```
Transition <trans> = new <event>(<params...>, <ostate>) { ... }
```

where:

- `<trans>` is the name of the variable holding the transition;
- `<event>` is the name of a subclass of `Transition` that represents the class of event labeling this transition. Here we use `Event`, a transition that can be triggered by simple events labeled by text strings;
- `<params>` is a (possibly empty) list of parameters characterizing the event. Here, we use the label of the event we want to match, i.e., `"coffee on"`;
- `<ostate>` is the name of the output state of the transition (if omitted, the output state is the same as the input state, i.e. the state in which the transition is declared). Note that the output state is specified using a text string rather than the name of the output state itself. This is a consequence of Java's rules for initializing variables: when a transition object is instantiated, such as `coffeeOn` on line 20, the parent object is not yet created and its variables, in particular those holding states such as `off` or `busy`, are still `null`. Our solution is to specify output states by a string holding the name of the state variable and to resolve these strings to the corresponding states using Java's reflection package[§] the first time the state machine is used, i.e. only once. Leading non-alphabetic characters are ignored, which allows developers to make output states stand out in their code (we use `" >> "` in our examples).

Transitions, i.e. the anonymous inner classes that specify transitions, can override two methods: `guard` to specify a boolean guard (e.g., line 21 for testing that there is water in the container) and `action` to specify the transition action (e.g., line 22 for opening the water tap).

[‡]in Java, an inner class is a class declared within another class. An instance of an inner class only exists within the scope of an instance of the parent class, and the inner class instance has access to the parent object's methods and variables. If `A` is the name of a class, the construct `new A() {...}` creates a new anonymous subclass of A *and* a single instance of that anonymous subclass. The anonymous subclass can specify arbitrary methods and variables within the curly braces, as with a normal class.

[§]a side effect of this solution is that states must be declared as `public`, otherwise they are not visible to the reflection API.

Of course, transitions, as well as states and state machines, can specify additional methods and variables for their own use. For example, the `coffeeMaker` state machine could store its current temperature in a variable. The natural nesting of scopes provided by anonymous inner classes means that any state and any transition could access the temperature. Similarly, a method or variable declared within a state can only be accessed from inside that state, including from its transitions.

Transitions are triggered by sending events to the state machine. In our example, the "`switch on`" event can be sent to the coffee maker as follows:

```
coffeeMaker.processEvent( new VirtualEvent("switch on") );
```

`VirtualEvent` is a subclass of Java's standard `EventObject` that contains the event's label to be matched with an `Event` transitions, e.g., line 3 in Figure 2.

In summary, anonymous inner classes provide a natural syntax where states are nested within state machines, transitions are nested within their input states and guards and actions are specified by overriding methods. State machines are specified in plain Java so that programmers can use their usual tools. For example, debugging can be done by setting a breakpoint on an `enter` method to break when the corresponding state becomes current. Note that this approach does not preclude the use of specific tools. For example, *SwingStates* provides facilities for graphically displaying state machines at run-time (see figure 20 for an example). However, such tools are not mandatory and our experience shows that the Java syntax is easily adopted even by novice programmers (see section 8).

SwingStates implements different kinds of state machines. The simplest and most general kind, `StateMachine`, features two types of basic events: events identified by a text string, as we have seen above, and `Timeout` events that are fired after a delay specified by the programmer. The other kinds of state machines are tailored to programming user interfaces with the Java Swing toolkit and are described in the following sections.

3. PROGRAMMING USER INTERFACES WITH STATE MACHINES

While *SwingStates* can be used to program any general state machine, it has been especially designed to facilitate the implementation of interaction techniques and user interfaces. This section reviews the state-of-the-art in state machines for describing interaction and then argues for the use of state machines in user interface toolkits.

Since Newman's seminal work [8], state machines have often been used to describe user interaction (see [9] and [10] for early examples) but rarely to actually directly program it [11]. For example, Buxton [12] introduced the 3-state model to describe typical press-drag-release interactions and Hinckley et al. [13] extended this model to describe more complex bi-manual interactions. While they both use state machines to model interaction, neither use them for their implementation. In 1990, Myers introduced Interactors [14], which are objects that intercept events and translate them into operations on a target item. The Garnet [15] and Amulet [16] toolkits implement Interactors as press-drag-release state machines but only provide developers with a small set of predefined behaviors, such as a *move and grow* interactor that translates or scales objects. Unfortunately, many interactions cannot be described by these simple predefined state machines. For example, displaying a tooltip or opening a folder using

a spring-loaded interaction as with Mac OS X require more complex state machines that use time as input.

Jacob et al. [17] proposed a model and a language based on state machines to capture the formal structure of post-WIMP interactions. Using this model requires a dedicated environment: the developer specifies the machines using a visual editor and enters the actions as C++ code by clicking on the relevant machine parts. The hybrid representation is then translated into an executable C++ program. HsmTk [18] is a more recent C++ toolkit that also uses state machines to program interaction. It uses the W3C Scalable Vector Graphics format (SVG) to describe the graphical rendering of interactive objects and a textual format for the hierarchical state machines that describe their behavior. HsmTk uses a preprocessor to translate state machines and rendering code into C++. In both cases, the lack of integration with existing user interface tools and the requirement to use specific tools and formats seriously hinder the adoption of these systems by developers.

Developers of user interfaces instead have adopted widely available toolkits such as Java Swing, Gtk [2], Qt [19] or the native tools provided by Mac OS X and Microsoft Windows. Programming interaction techniques with these toolkits consists in assembling standard widgets such as buttons, checkboxes or linear menus and linking them to the rest of the application through callbacks or listeners. Such interactive applications are notoriously difficult to debug and maintain due to the difficulty of following the flow of control [3]. Moreover, creating new widget classes is difficult because it requires an intimate knowledge of the inner workings of the toolkit. As a result, developers usually stick to the predefined widget set and applications rarely feature novel interaction techniques.

We claim that using state machines instead of event handlers (callbacks or listeners) leads to cleaner and easier to maintain user interface code and can facilitate the implementation of novel and advanced interaction techniques in mainstream applications. The main reason for the "spaghetti of callbacks" [3] is the fact that a single interaction is spread among several handlers and that a single handler may contain code from several interactions. For example, consider a press-drag-release interaction that moves an object if started on that object and moves the whole window content (called panning) otherwise. This requires three handlers for the press, drag and release events. Each event must be declared and bound to its event type. The body of each handler must test whether it is moving an object or panning the window, requiring a global variable to coordinate state among the handlers. With a state machine, we need three states: an initial state that awaits for a press and determines whether it is a move or a pan, a state that will be active while moving an object, and a state that will be active while panning the window. Each state only specifies the events it is interested in, and the whole specification is localized in a single piece of code. Figure 3 compares these two approaches. While the code length is similar, it is important to realize that the logic of the interaction is better represented in the state machine. For example, adding a constrained move when the user holds the Shift key down translates into adding a state to the state machine while it requires changing all three handlers and adding a global variable to the event handler approach.

Indeed, the developer must follow a different cognitive process when programming with state machines vs. event handlers. Event handlers require an *event-centered* approach: for each event type, the developer must ask herself "What are the different cases in which the interface must handle this event and how?" Adding or modifying interactions typically affects several handlers

```
Shape dragged = null;
boolean moveShape = false;

MouseListener listener = new MouseAdapter () {
  public void mousePressed(MouseEvent evt) {
    if(evt.getButton() == BUTTON1) return;
    dragged = pick(component, evt.getPoint());
    moveShape = (dragged != null);
    if(moveShape) {
      dragged.hilite();
    }
  }

  public void mouseReleased(MouseEvent evt) {
    if(evt.getButton() == BUTTON1) return;
    if(moveShape)
      dragged.unhilite();
    dragged = null;
  }
}

component.addMouseListener(listener);
[...]

MouseMotionListener motionListener =  new MouseMotionAdapter () {
  public void mouseDragged(MouseEvent evt) {
    if(evt.getButton() == BUTTON1) return;
    if(moveShape)
      background.move();
    else
      dragged.move();
  }
}

component.addMouseMotionListener(motionlistener);
```

```
StateMachine machine = new StateMachine() {
  Shape dragged;

  public State start = new State() {
    Transition t1 = new Press(BUTTON1, "--> move_shape") {
      public boolean guard() {
        dragged = pick(component, evt.getPoint());
        return dragged != null;
      }
    }
    Transition t2 = new Press(BUTTON1, "--> move_bg") { }
  }

  public State move_shape = new State() {
    public void enter() { dragged.hilite(); }
    Transition t1 = new Drag(BUTTON1) {
      public void action() {
        dragged.move();
      }
    }
    Transition t2 = new Release(BUTTON1, "--> start") { }
    public void leave() { dragged.unhilite(); }
  }

  public State move_bg = new State() {
    Transition t1 = new Drag(BUTTON1) {
      public void action() {
        background.move();
      }
    }
    Transition t2 = new Release(BUTTON1, "--> start") { }
  }
}

machine.attachTo(component);
```

Figure 3. Code for a "drag" interaction with event handlers (left) and with a state machine (right)

and requires new global variables, so event handlers are more compact for interfaces that have only a few states or for interfaces whose behavior is very similar among states. State machines require a *state-centered* approach instead: for each state, the developer must ask herself "Which events are relevant to this state, i.e., which events cause a significant change, and which states (existing or new) do they lead to?". Adding or modifying interactions typically result in fairly localized changes, although it may be difficult to control the growing number of states.

The potential advantages of using state machines to program interactions (as opposed to just describing them) and the fact that Java developers are used to the Swing user interface toolkit has led us to *SwingStates*: the first user interface toolkit to use state machines for programming interaction directly in Java. Other advanced Java toolkits, such as Piccolo [20]

and Zoomable Visual Transformation Machine (ZVTM) [21] for zoomable interfaces or InfoVis [22] and Prefuse [23] for Information Visualization replace rather than extend Swing while keeping an event-based model for handling input. A notable exception is SubArctic [24], which builds on AWT (the predecessor of Swing) and Java2D, but manages input through a flexible mechanism based on agents and dispatch event policies.

The following sections describe the two main uses of *SwingStates*:

- developing new widgets: *SwingStates* features (i) a canvas, which is a Swing widget (a subclass of `JComponent`) with a rich graphical model, and (ii) state machines with a comprehensive set of transitions dedicated to this canvas;
- customizing existing widgets: *SwingStates* features (i) state machines that can be used to replace or extend the listeners of any Swing widget and (ii) state machines that can be used to define interactions across several widgets.

4. PROGRAMMING NEW WIDGETS WITH THE SWINGSTATES CANVAS

Most user interface toolkits, including Swing, make it difficult for developers to create new widget classes. Consequently, interactive applications do not take advantage of novel interaction techniques such as the control menus [25] that we use as example in this section. *SwingStates* features a canvas widget class, `Canvas`, designed specifically to create new widgets. A canvas contains a set of shapes defined by geometrical and graphical attributes, as well as attributes dedicated to interaction. Interaction with the canvas and its shapes is defined using a special class of state machines, `CStateMachine`, that contains transitions for input events that take into account these interaction attributes. The rest of this section presents the canvas' graphical model and shows how to define interactions with the canvas' content.

4.1. Graphical model

A *SwingStates* canvas contains a set of graphical objects called *shapes* stored in a display list. Unlike other Java canvases such as AWT's `Canvas` and Swing's `JPanel` where developers have to paint the content of the canvas directly using the Java2D API for 2D graphics, the *SwingStates* canvas is declarative: its on-screen display is updated automatically when the content of its display list is changed, e.g. when adding or removing shapes or when changing their attributes. This is a similar approach to the scene graphs often used in 3D graphics, such as Inventor [26].

Each shape in a canvas is a specialization of the `CShape` class. All shapes have a geometry, defined by a Java2D `Shape` and an affine transformation, and visual attributes, defined by Java2D `Paint` (fill and border color) and `Stroke` (border style). *SwingStates* comes with six predefined shape classes (figure 4): `CRectangle`, `CEllipse`, `CSegment`, `CText`, `CImage` and `CWidget`. `CWidget` is a special class that allows any Swing widget to be treated and manipulated like a shape while keeping it interactive. For example, a `JButton` can be scaled and rotated and still be clicked. All shapes have setters, e.g. `setPaint(Paint)` for any `CShape` or `setText(String)` for `CText`, that trigger a redisplay of the canvas to keep the internal and

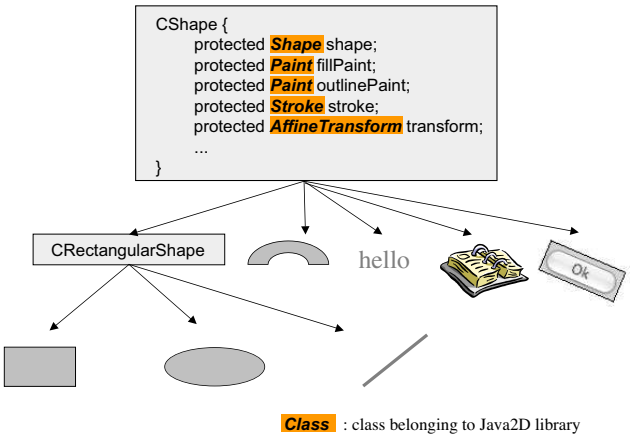

Figure 4. *SwingStates*' predefined canvas shapes

visual states consistent. While the predefined shape classes cover most needs, developers can create their own shapes by extending `CShape` or any of the predefined shape classes.

The shapes in the canvas are displayed in the order of the display list, i.e. the last shape in the list is on top of the others. Shapes can be added or removed from the display list and their order in the list can be changed. Shapes can also be hidden and then shown without removing and re-adding them to the list by using the `drawable` attribute. This is useful to improve performance and simplify programming for interfaces that show shapes temporarily, such as a popup menu.

By default, the coordinates of a shape are relative to the canvas' upper-left corner. These coordinates can be transformed by an affine transform that combines scaling, translation and rotation. The transformation can be relative, i.e. it is combined with the current shape's transform, or absolute, i.e. it replaces it. By default, the center of the scale and rotation transforms is the center of the transformed shape but developers can set it to any point. The coordinates of a shape can also be relative to those of another shape, called its *parent*. In this case, the shape's transform is combined with its parent's. As a result, applying a transform to a shape affects all its descendants. Finally, a shape s can be *clipped* by another shape c so that only the parts of s that intersect c are displayed. As in the GmlCanvas toolkit [27], these two relationships among shapes (hierarchy and clipping) are independent from each other and from the order in the display list. For example, a parent shape can also be a clipping shape, and a parent shape can be displayed above or below its child. Such independence provides a lot of flexibility to describe complex displays and interact with them.

As an example, we describe the implementation of a *control menu*. A control menu [25] is a circular menu that works as follows: the user invokes the menu by pressing a mouse button, selects a menu item by crossing it and then continues to drag the mouse to control a parameter

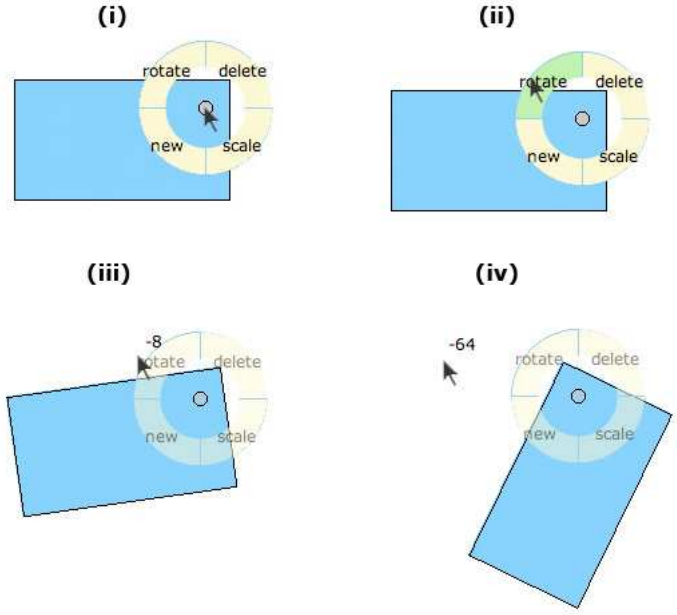

Figure 5. Setting rotation angle using a control menu

of the selected command. For example, figure 5 shows an interaction to set the rotation angle of a shape: (i) the user invokes the menu by pressing the mouse button ; (ii) the user visits the "rotate" item ; (iii and iv) the user controls the rotation angle by dragging the mouse after having crossed the item: dragging towards the left increases the angle while dragging towards the right decreases it. Figure 6 shows the relationships among the shapes to implement a control menu. Note that the parent of all the menu shapes is the last shape in the display list. Displaying the control menu at a given location therefore simply consists of applying an absolute translation to this shape. Showing and hiding the whole menu however requires another feature called *tags*.

Each shape in a canvas can be *tagged*. A tag is a collection of objects that can be enumerated using an iterator. Each object can have several tags and a given tag can be attached to several objects. Tags are typically used to represent groups of shapes and to manipulate all the shapes in a group at once. To that effect, the `CTag` class defines most of the methods of the `CShape` class so that developers can manipulate all objects with a given tag as if it were a single object. For example, the instruction below translates all the shapes tagged by `aTag`:

```
aTag.translateBy(10, 20);
```

SwingStates' tags were inspired by the Tk toolkit [28]. In Tk, a tag is simply a text string that can be attached to one or more graphical shapes. The Tk canvas includes a set of commands

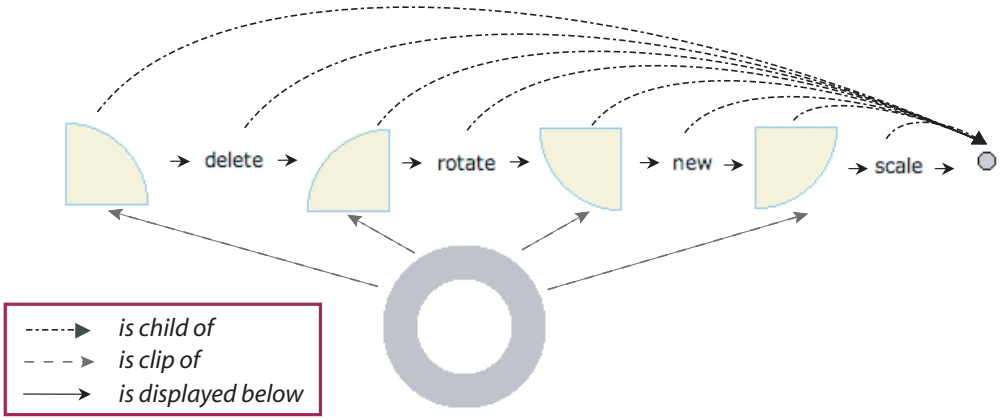

Figure 6. Relations between shapes in a control menu

that take a tag name as parameter to modify all the objects that have this tag at once. While Tk tags are implemented as simple lists associated to the tag's name, *SwingStates*' tags are full-fledge objects and therefore are more powerful. *SwingStates* features two types of tags: *extensional tags*, which are explicitly added to or removed from a shape, as in Tk, and *intentional tags*, which are specified by a predicate that tests whether or not a shape has this tag. Extensional tags can be *active*: each time an extensional tag is added to or removed from a shape, the tag's `added(CShape)` or `removed(CShape)` method is called.

For example, to implement a selection mechanism, we define a tag `selected` that we add to selected shapes. In order to provide a visual feedback of selected objects to the user, we override the `added(CShape)` and `removed(CShape)` methods so that selected shapes have a 2-pixels outline while non selected shapes have a 1-pixel outline:

```
CExtensionalTag selected = new CExtensionalTag(canvas) {
    public void added(CShape s) {
        s.setStroke(new BasicStroke(2));
    }
    public void removed(CShape s) {
        s.setStroke(new BasicStroke(1));
    }
};
...
// create three objects, select two of them and translate them:
CRectangle rect = canvas.newRectangle(100, 100, 50, 150);
CEllipse circle = canvas.newEllipse (300, 300, 350, 350);
CText text = canvas.newText(250, 250, "Hello world", new Font("verdana", Font.PLAIN, 12));
rect.addTag(selected);
text.addTag(selected);
selected.translateBy(10, 20);
```

Intentional tags are used to define a set of shapes according to a given criterion. Each time an intentional tag is used, the set of tagged shapes is recomputed. Because this can be computationally expensive, e.g., when the canvas has many shapes, intentional tags should be used wisely. The following example shows an intentional tag that turns all the red shapes of a canvas into green shapes:

```
CIntentionalTag redShapes = new CIntentionalTag(canvas) {
    public boolean criterion(CShape s) {
        return s.getFillPaint() == Color.RED;
    }
};
...
redShapes.setFillPaint(Color.GREEN);
```

4.2. State machines for the *SwingStates* canvas

`CStateMachine` is a class of state machines dedicated to programming interaction with the canvas and its shapes. It provides a set of transition classes of the form `<EventType><?Context>` to handle standard input events in different contexts. For example, a `Press` transition is triggered by a mouse press anywhere in the canvas, while a `PressOnShape` transition is triggered by a mouse press only if the cursor is on a shape, and a `PressOnTag(CTag)` transition is triggered by a mouse press only if the cursor is on a shape that has the given tag. The different contexts that are recognized by *SwingStates* can be seen as predefined guards for the basic input event transitions. The implementation however is more complicated: when a state machine is attached to a canvas and its current state contains `*OnShape` or `*OnTag` transitions*, the canvas performs *picking*, i.e. it determines which shape is under the cursor and makes this shape available to the state machine. Note that a shape can be made *non pickable* if it is to be displayed but ignored for picking. The canvas state machine also implements a priority among the contexts so that an `*OnTag` transition, e.g., `PressOnTag`, has priority over an `*OnShape` transition, e.g., `PressOnShape`, which has priority over a simple input event transition, e.g., `Press`.

The code in figure 7 implements the state machine that runs our control menu. State `menuOff` (line 1) uses a `PressOnShape` transition (line 5) to display a menu of shape creation commands and a `Press` transition (line 8) to display a menu of application commands. State `menuOn` (line 12) becomes active once the mouse button is pressed and stays active until either the mouse button is released (line 13) or an item is selected (line 17). If the mouse enters a menu item (line 14), this item becomes the current one and if it then leaves the whole menu, the machine enters state `control`. In this state, the user controls a parameter of the current command (line 21) until she releases the mouse button (line 23). To detect when the mouse cursor enters a menu item and to get its associated command, we use an extensional tag of class `ControlMenuItem` that stores the associated command as a field:

*We use the notation `*<context>` to specify any input event (`Press`, `Release`, `Move`, `Drag`, `Wheel`, `Enter`, `Leave`) in the given context. Similarly, we will use `<event>*` to specify the given input event in any context (`OnShape`, `OnTag`).

```
1  CStateMachine interactionControlMenu = new CStateMachine() {

2    Command selectedCommand;

3    public State menuOff = new State() {
4      public void enter() { hideMenu(); }

5      Transition invokeOnShape = new PressOnShape(BUTTON1, ">> menuOn") {
6        public void action() { showMenuShape(getPoint()); }
7      };

8      Transition invoke = new Press(BUTTON1, ">> menuOn") {
9        public void action() { showMenuBackground(getPoint()); }
10     };
11   };

12   public State menuOn = new State() {

13     Transition stop = new Release(BUTTON1, ">> menuOff") { };

14     Transition enterItem = new EnterOnTag(ControlMenuItem.class) {
15       public void action() { selectedCommand = ((ControlMenuItem)getTag()).getCommand(); }
16     };

17     Transition selectCommand = new EnterOnTag("background", ">> control") { };

18   };

19   public State control = new State() {

20     Transition control = new Drag(BUTTON1) {
21       // apply selectedCommand with a parameter computed according to the cursor location
22     };

23     Transition stop = new Release(BUTTON1, ">> menuOff") {
24       public void action() { canvas.getTag("background").setPickable(false); }
25     };

26   };

27 };
```

Figure 7. State machine for a control menu

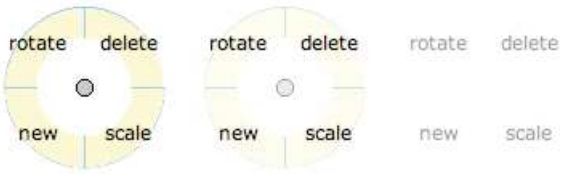

Figure 8. Fading out a menu (labels start to fade out only once the item backgrounds are completely transparent.)

```
class ControlMenuItem extends CExtensionalTag {
  private Command command;
  public ControlMenuItem(Command command) {
    super();
    this.command = command;
  }
  public Command getCommand() { return command; }
}
```

Each menu item has a different tag that holds its command, however all these tags belong to the same class. The transition `EnterOnTag(ControlMenuItem.class)` (line 14) is triggered when the mouse cursor enters a shape with a tag of this class and its action can retrieve the associated command (line 15)[¶].

This example uses an invisible object to distinguish between the cursor leaving the menu item to select it (i.e., the mouse cursor leaves the whole menu) from the cursor leaving the menu item to select another item (i.e., the mouse cursor leaves the current item but enters a new one): When displaying the menu (lines 6 and 9), we add a shape just below the menu that is larger than the whole menu. We make this shape non visible but pickable and we tag it with the string "background". When the mouse cursor enters this shape (line 17), it means that the cursor has left the whole menu. When the user releases the mouse button, we make this shape non pickable so as not to interfere with other graphical shapes (line 18).

4.3. Animation

The *SwingStates* canvas and state machines support animation. Any shape, any tag and even the whole canvas can be animated using mechanisms similar to those found in other toolkits that support animation, such as ZVTM [21] or Piccolo [20]. An animation is an object that can be started, suspended, resumed and stopped. It has a duration T, a number of laps N and a delay d. While the animation is running, a parameter t goes from 0 to 1 over T milliseconds during odd laps, and from 1 down to 0 during even laps. The parameter t evolves according to

[¶]Note that one could also specialize the shape representing the item to hold the command, but this would be less generic. With the tag approach, any shape can be a menu item.

either a linear or a sigmoid function of absolute time (the latter implements the typical slow in-slow out of cartoon animations [29]). Every d milliseconds, the animation's `step` method is called to update the shapes it controls and trigger a redisplay of the canvas. *SwingStates* includes a set of predefined animations to change the style and geometrical attributes of shapes and tags, such as changing the transparency level to make an object fade in or out.

The main originality of *SwingStates* animations is their integration with state machines and with the tagging mechanism. The combination of these two features help manage the scheduling of animations. For example, in order to make an object shrink while another object changes its color, the developer can use two animations with the same duration, tag them with the same animation tag (a subclass of regular extensional tags) and call the tag's `start` method to start both animations simultaneously. Animations trigger events when started, suspended, resumed and stopped. These events can be handled in state machines using transitions such as `AnimationStarted` that take an animation or an animation tag as parameter.

For example, we may want to improve the graphical transitions in our control menu by fading the menu out when the user releases the mouse button rather than instantly hiding it. Also, because users are faster at visually searching for an item when they know the menu layout, we want to make item labels more remanent than the menu background which clutters the user's working area and does not contain useful information. We use two animations that continuously increase transparency (figure 9): `animBg` (line 1) for making the item backgrounds disappear, followed by `animLabels` (line 2) for making item labels disappear. `animBg` is started by the state machine that runs the control menu each time an item has been selected. In order to start `animLabels` only once `animBg` has stopped, we use a separate state machine (lines 3-24) that runs in parallel with the control menu state machine (section 7 will elaborate on the various ways to combine multiples state machines).

When `animBg` is started, this machine goes from state `idle` to state `bgDisappearing` (line 4-6). Then, when `animBg` ends (line 8), `animLabels` is started (line 11) and the current state becomes `labelsDisappearing` until `animLabels` ends (line 19). We also force the current animation to stop if the user invokes the menu again by pressing the mouse button while the menu is still fading out (lines 14-16 and 20-22).

This section described how to use *SwingStates*' canvas and state machines to implement a control menu, a widget that current user interface toolkits do not feature. The canvas' graphical model made it easy to describe the visual aspect of the menu while the use of tags and state machines resulted in a compact and easy to follow description of the interaction. By using a separate state machine for animation, we kept the descriptions simple and modular. The next section describes how *SwingStates* can also be used to redefine the interaction with existing Swing widgets, therefore providing developers with a complete tool for leveraging advanced interaction techniques in existing as well as new Java Swing applications.

5. ENHANCING INTERACTION WITH EXISTING WIDGETS

While some interaction techniques require novel widgets, as shown with the control menu in the previous section, many can be implemented on top of existing widgets. For example,

```
1  AnimationTransparency animBg = new AnimationTransparency(0);
2  AnimationTransparency animLabels = new AnimationTransparency(0);
...
3  new CStateMachine(canvas) {
4    public State idle = new State() {
5      Transition t = new AnimationStarted(animBg, ">> bgDisappearing") { };
6    };
7    public State bgDisappearing = new State() {
8      Transition t = new AnimationStopped(animBg, ">> labelsDisappearing") {
9        public void action() {
10         // all labels are tagged with tagLabels so they can be animated at once
11         tagLabels.animate(animLabels);
12       }
13     };
14     Transition invokeMenu = new Press(BUTTON1, ">> idle") {
15       public void action() { animBg.stop(); }
16     };
17   };
18   public State labelsDisappearing = new State() {
19     Transition t = new AnimationStopped(animLabels, ">> idle") { };
20     Transition invokeMenu = new Press(BUTTON1, ">> idle") {
21       public void action() { animLabels.stop(); }
22     };
23   };
24 };
```

Figure 9. State machine for fade-out animation of a menu

OrthoZoom [30] is a navigation technique that turns a traditional scrollbar into a powerful multi-scale navigation tool for very large documents: dragging the cursor along the colinear dimension of the scrollbar, i.e. vertically in a vertical scrollbar, pans the document as with a regular scrollbar while dragging the cursor along the orthogonal dimension, i.e. horizontally in a vertical scrollbar, scales the whole document up or down to move faster or slower (Figure 10). As another example, CrossY [31] augments traditional widgets so they can be manipulated using crossing interactions instead of clicking. In CrossY, crossing a checkbox selects or deselects it, crossing a scrollbar one time or several times in a row makes the document move by a single step or by entire pages, etc.

SwingStates allows developers to reuse existing Swing widgets and redefine their interaction with a state machine without having to reimplement the widgets' rendering. First, *SwingStates* features a class of state machine that replaces Swing's event handlers for processing the standard mouse, keyboard and timer events. Second, *SwingStates* features a special class of state machines and component tags to define interactions over a set of heterogeneous widgets.

`JStateMachine` is a class of state machines designed to specify the interaction within and across existing Swing widgets. It contains transitions suffixed by `*OnComponent` to handle events on a given Swing widget and transitions suffixed by `*OnTag` to handle events on a

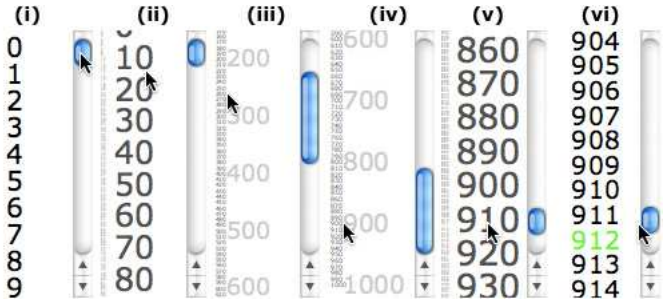

Figure 10. OrthoZoom Scroller: moving the cursor to the left (i)(ii)(iii) zooms out, as shown by the range of numbers and the size of the scrollbar thumb. The user may then move the cursor down (iv) to move quickly to the end of the range and then to the right (v) (vi) in order to zoom in and adjust the destination position

tagged widget. Similarly to the graphical shapes of a canvas, the widgets in a Swing container can be tagged. In particular, when a `JStateMachine` is attached to a widget, this widget is automatically tagged by the name of its class, e.g. a button has the tag "javax.swing.JButton" and a checkbox has the tag "javax.swing.JCheckBox".

The example in figure 11 turns a regular scrollbar into an OrthoZoom scroller that controls the content of a canvas. The state machine is attached to the panel that contains the scrollbar and canvas because, unlike a regular scrollbar, OrthoZoom uses the area of the controlled widget for the interaction. Pressing the mouse button on the scrollbar (line 6) starts a regular scrolling interaction while the cursor stays within the scrollbar (lines 14–18) and until the mouse button is released (line 19). If the cursor is dragged over to the canvas (lines 9–13), the document is scaled by a zoom factor that depends on the horizontal distance between the cursor and the right edge of the canvas and is then scrolled (see [30] for the mapping function). The state machine also implements rate-scrolling [30]: when the mouse cursor leaves the canvas during a scrolling interaction (line 20), we arm a repeating timer (line 22) to scroll continuously (lines 32-36) until the mouse cursor reenters the canvas (lines 27–31).

More complex interactions can be implemented by using a feature of Swing called the *glasspane*. The glasspane is an overlay plane that floats above the widgets in a window and that can contain arbitrary Swing components. *SwingStates* can create a canvas in the glasspane of a window; by attaching a state machine to that canvas, a wide range of novel interaction techniques can be implemented.

Figure 12 illustrates the crossing interaction described at the beginning of this section: checkboxes can be selected and buttons can be activated by crossing them instead of clicking

```
1  JStateMachine jsm = new JStateMachine() {

2    public State start = new State() {
3      public void enter() {
4        // scale view to 1
5      };
6      Transition startControl = new PressOnTag("javax.swing.JScrollBar", BUTTON1, ">> control") { };
7    };

8    public State control = new State() {
9      Transition orthoZoom = new DragOnTag("fr.lri.swingstates.canvas.Canvas", BUTTON1) {
10       public void action() {
11         // zoom and pan
12       }
13     };
14     Transition simplePan = new DragOnTag("javax.swing.JScrollBar", BUTTON1) {
15       public void action() {
16         // simple panning
17       }
18     };
19     Transition stopControl = new Release(BUTTON1, ">> start") { };
20     Transition startRateScrolling = new Leave(">> rateScroll") {
21       public void action() {
22         armTimer(40, true);
23       }
24     };
25   };

26   public State rateScroll = new State() {
27     Transition stopRateScrolling = new Enter(">> control") {
28       public void action() {
29         disarmTimer();
30       }
31     };
32     Transition timeOut = new TimeOut() {
33       public void action() {
34         // rate scrolling: one scroll step
35       }
36     };
37     Transition stopControl = new Release(BUTTON1, ">> start") { };
38   };

39 };
...
40 // frame is a window containing a Canvas and a vertical scrollbar
41 jsm.attachTo(frame);
```

Figure 11. State machine for an OrthoZoom Scroller

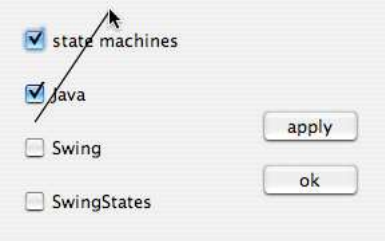

Figure 12. Crossing interaction with checkboxes and buttons

them[‖]. Note that an ink trail is left behind the cursor, which disappears when the mouse button (or pen) is released. This provides feedback to the user.

This interaction can easily be implemented with two state machines running in parallel. One state machine is attached to the glasspane. It displays the ink trail and erases it when the mouse button is released. The second state machine is attached to the frame covered by the glasspane. It detects enter/leave sequences of events on buttons and checkboxes and selects the checkbox or invokes the button as soon as the cursor finishes crossing the widget with the mouse button depressed.

In summary, *SwingStates* is designed to work with existing widgets when a complete redefinition is not necessary. Redefining the behavior of existing Swing widgets saves time when compared with creating new widgets from scratch. It also helps repurpose existing applications by keeping their visual appearance while changing the interaction, for example to improve their usability when used with different input devices such as a pen. The following section pushes this approach further by allowing *SwingStates* applications to use any input device, not just the predefined mouse (or pen) and keyboard.

6. SUPPORTING GENERALIZED INPUT

Figure 13 summarizes the set of transitions supported by *SwingStates*. It covers interaction with the keyboard and mouse, animations, and timers. While this represents a large vocabulary, this is still insufficient for some advanced interaction techniques that use non-standard devices. For example, Zliding [32] is a stylus-based multi-scale navigation technique that uses pressure to control zooming and movements to control panning. Swing is limited to standard mouse and keyboard events and therefore, even when using a pressure-sensitive pen, pressure information is not available.

[‖]This interaction technique is particularly well suited to pen-based devices such as PDAs and Tablet PCs, although it does work with a mouse as well.

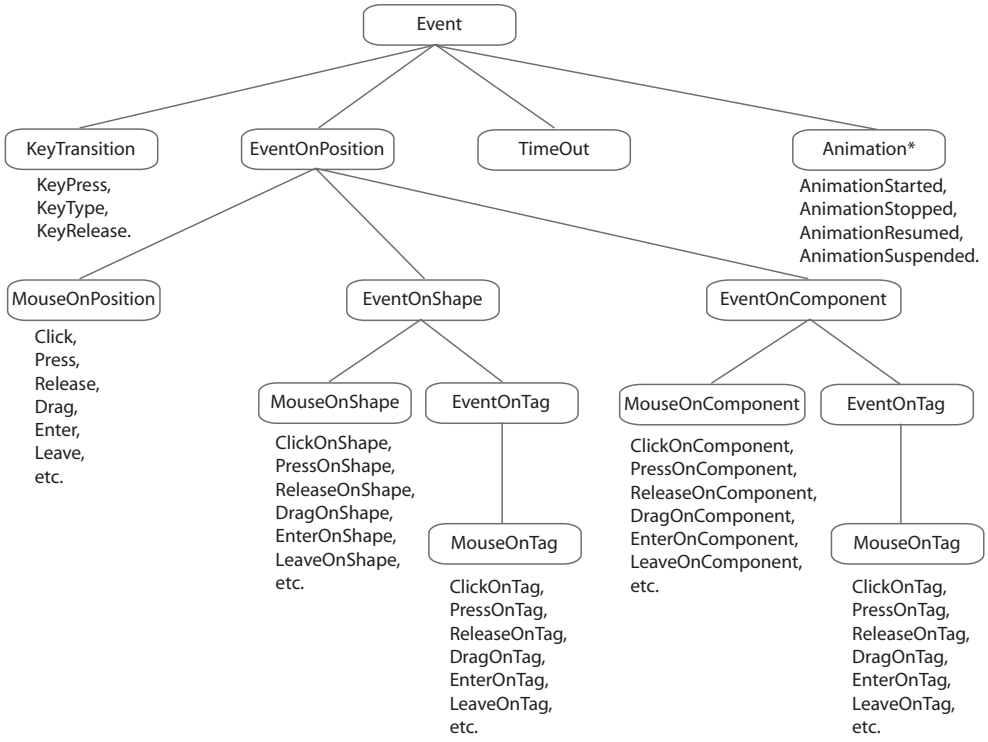

Figure 13. Classes of transitions available in *SwingStates*

Moreover, all devices that resemble a mouse are merged together, as are all devices that resemble a keyboard. So for example, on a laptop computer that has both a trackpad and a mouse, both devices control the single on-screen cursor and Swing reports the activity of this cursor rather than that of the actual input devices*. As a result, interface designers cannot implement bi-manual techniques such as toolglasses or bi-manual palettes (Figure 14). These techniques use one pointing device per hand in order to perform actions in parallel and have been shown to be more efficient than standard, unimanual palettes [33]. For example, a toolglass [34] is a semi-transparent palette that can be moved with a mouse or trackball held in the non-dominant hand while the cursor is still controlled by a mouse held in the dominant hand. To apply a tool at a given position, the user moves both the palette and cursor to this position and clicks with the cursor *through* the palette to apply the appropriate tool. A bi-

*This includes the fact that even though the mouse or trackball can be scrolled arbitrarily far away in any direction, the reported events have their x and y coordinates clamped to the screen physical dimensions.

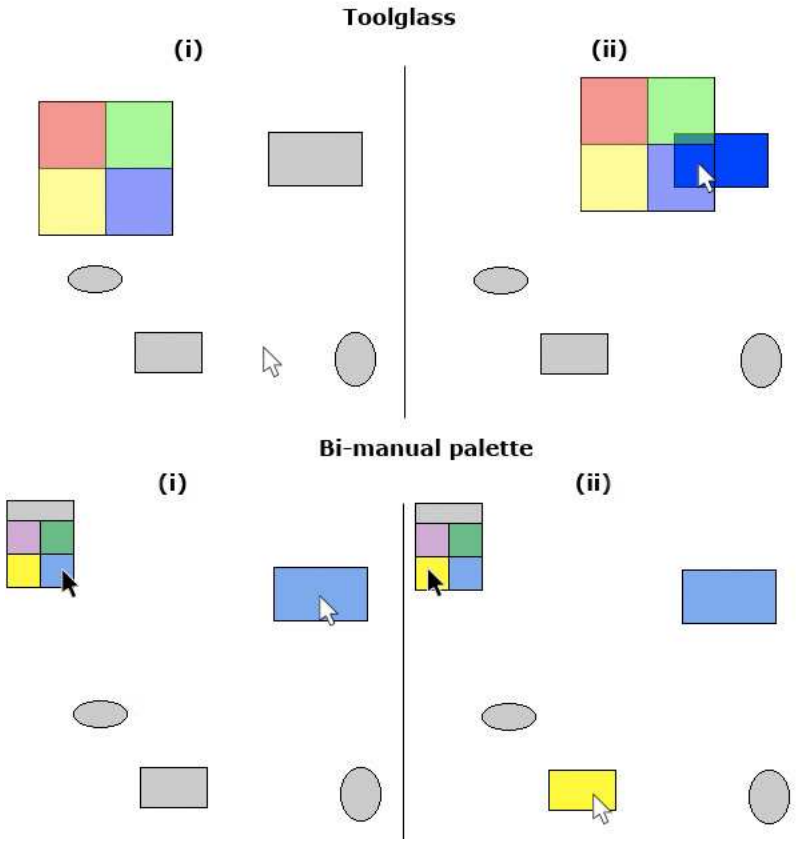

Figure 14. Toolglass and bi-manual palette

manual palette is another variant of a standard palette that uses two cursors, one for selecting tools with the non-dominant hand and one for applying the selected tool in the working area with the dominant hand. This technique minimizes mouse movements by avoiding round trips from the palette to the working area (the non-dominant cursor stays in the palette area while the dominant cursor stays in the working area).

The rest of this section describes how *SwingStates* supports such advanced interactions. It first describes the general input device model and its low-level events, and then the higher-level event model and its use to program a bi-manual palette.

6.1. Device model and low-level events

Handling non-standard input devices in Java requires access to low-level system information. In order to keep *SwingStates* as portable as possible, we use the JInput library [35], which is implemented on many hardware platforms. JInput describes input devices in terms of physical devices called *controllers*, each composed of a set of *axes* that describe its state. For example, a regular mouse is a controller, e.g., "USB Mouse", with two axes for its two buttons, e.g., "USB Mouse.left" and "USB Mouse.right", and two axes for the mouse displacements, e.g., "USB Mouse.x" and "USB Mouse.y". JInput can query which controllers are available and the current value of each axis.

We use an input manager that we first developed in the context of the TouchStone platform [36]. This component queries the values of the axes of all known devices every *delay* milliseconds and generates an event each time an axis has changed**. These events can be handled in state machines with `AxesEvent` transitions.

Handling such low-level events however is cumbersome. For example, tracking a mouse requires listening to three axes ("USB Mouse.x", "USB Mouse.y" and "USB Mouse.left"), testing the value of "USB Mouse.left" to determine whether the mouse button was pressed or released, and duplicating transitions for "USB Mouse.x" and "USB Mouse.y" to track mouse motion. For this reason, we provide a higher-level model for handling events.

6.2. Handling high-level events

In order to support input events at a similar level of abstraction as Swing, e.g., a click at a given 2D location, while supporting multiple devices and distinguishing them, *SwingStates* features the `GMouse` class. `GMouse` is in fact a state machine that consumes low-level events generated by the axes of a given JInput mouse controller and fires higher-level `PickerEvent` events to the Swing component that is attached to it. A `PickerEvent` is a specialization of the standard `java.awt.MouseEvent` class used by Swing that has an extra `GMouse` field.

The `GMouse` state machine implements the semantics of mouse events. For example, it detects a click as a button press quickly followed by a button release and sends the appropriate Press, Release and Click events; it integrates the relative displacements provided by the mouse into an absolute position within the component and clamps it to the component size; etc. Since the events fired by `GMouse` are subclasses of standard Swing input events, they can be handled by regular Swing components and *SwingStates* state machines as if they were regular input events, therefore providing upward compatibility.

The real value of `GMouse` however is that it can be used to implement bi-manual interaction techniques. For example, the machine in figure 15 uses a *SwingStates* canvas to implement a bi-manual palette for a laptop equipped with a trackpad and a USB mouse. Each tool in the palette has a tag named "tool" while the shapes that can be edited have a tag called "shape". The canvas also contains a black cursor associated with the trackpad (lines 1 and 3), used to

**The fact that JInput requires polling is a potential performance problem, although it has not been an issue for us until now.

```
1  CImage blackCursor = canvas.newImage("blackCursor.gif");
2  CImage whiteCursor = canvas.newImage("whiteCursor.gif");
// CanvasMouse is a subclass of GMouse that controls a canvas shape, here a cursor
3  CanvasMouse shapePicker = new CanvasMouse("USB Mouse", canvas, blackCursor);
4  CanvasMouse colorPicker = new CanvasMouse("Trackpad", canvas, whiteCursor);

5  new CStateMachine(canvas) {
6    Color colorSelected = null;

7    public State s = new SelectionState() {
8      Transition changeColor = new PressOnTag(colorPicker, "tool", BUTTON1, ">> on") {
9        public void action() {
10         colorSelected = (Color)getShape().getFillPaint();
11       }
12     };

13     Transition applyColor = new PressOnTag(shapePicker, "shape", BUTTON1) {
14       public boolean guard() { return colorSelected!= null }
15       public void action() {
16         getShape().setFillPaint(colorSelected);
17       }
18     };
19   };

20 };
```

Figure 15. State machine for a bi-manual palette

select tools in the palette, and a white cursor associated with the USB mouse to apply the current tool (lines 2 and 4). The `PressOnTag` transitions take an optional first argument that specifies the input device that sent the event. Pressing the trackpad button when the black cursor is over a color tool in the palette changes the current color (lines 8–12) while pressing the mouse button when the white cursor is over a shape fills this shape with the current color (lines 13–18).

6.3. Integration into standard interfaces

Since `PickerEvent` events are standard Swing events, this generalized input scheme works with any standard mouse listener as well as with the state machines presented in this article. There are still a few problems however with the seamless integration of generalized input into *SwingStates*. This part of the toolkit is therefore still experimental.

The main problem is the proper management of the cursor(s). Although we integrate the relative x and y displacements provided by the mouse controller into absolute values, these absolute values do not match the system cursor location because (i) the system cursor is controlled by *all* pointing devices and (ii) its position is modified by system properties such

as cursor acceleration. An alternative would be for *SwingStates* to hide the system cursor and manage its own cursors, e.g. in a canvas associated to the glasspane. However the user must also be able to use the system cursor outside *SwingStates*-controlled widgets. Using Swing's notion of focus, we can track when the system cursor enters a widget that uses generalized input, hide the system cursor, constrain it to stay inside the widget (we use `java.awt.Robot` for that purpose), and display our cursor. But this is unduly complicated and does not work reliably on all platforms[††].

The work presented in this section is a first step to integrate generalized input into a full-fledge Java user interface toolkit. To the best of our knowledge, this has not been done before. However, we are still investigating how to make such an integration completely seamless, robust and independent of the runtime platform.

7. OVERCOMING THE LIMITATIONS OF STATE MACHINES

State machines are a simple formalism that is easy to understand and well adapted to describing interaction [9, 37]. Their main limitation however is the lack of scalability: as the system becomes larger and more complex, the number of states and the number of transitions grows, sometimes exponentially, and they become hard to maintain. This problem is well-known and a number of solutions have been presented to solve it, such as the early Recursive Transition Networks (RTN) [38]. Most solutions consist of adding modularity and a notion of hierarchy to state machines. The most well-known and succesful such extension is Harel's StateCharts [39], which introduce a hierarchy of states (a state can contain a state machine) and precisely defines the semantics of this construct. StateCharts were used in the StateMaster User Interface Management System [40], and a variant of them specifically tailored to designing user interfaces was used in the more recent HsmTk toolkit [18]. StateCharts however are significantly more complicated and hard to learn than plain state machines, and our experience is that user interface designers and developers have difficulties exploiting their power. Other approaches include Petri Nets [41], which have also been used to specify user interfaces, for example in the PetShop system [42]. Here too, the learning curve is steep, making the adoption of such a model by developers difficult.

Our experience in using state machines to describe interaction techniques is that the state explosion is not an issue when the state machine describes a single interaction, but it becomes a problem when combining the state machines for several techniques into a single one or when adding visual feedback, such as animations, across different techniques. Our solution is to support a form of modularity where multiple state machines can be active simultaneously and can run independently while communicating with each other.

[††]Some platforms require special privileges or extensions to control the position of the system cursor, which is understandable as it can be used for phishing attacks.

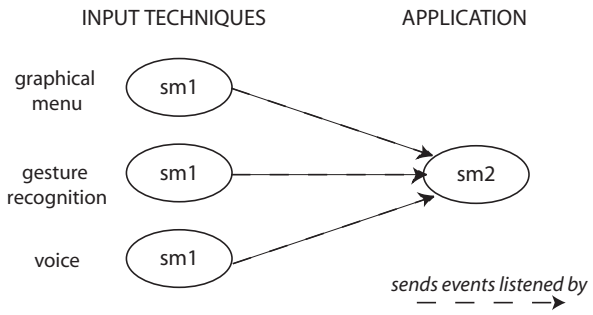

Figure 16. Communication between state machines

7.1. *SwingStates*' model of communicating state machines

With *SwingStates*, any number of state machines can run simulateneously. A state machine can be active, i.e., handling the events it receives, or inactive, i.e., ignoring events. It can send events to other state machines (including itself) and, of course, it can handle events received from other state machines. The execution model is strictly sequential, i.e., we do not rely on each state machine running in a different thread for example.

This modularity offers a high power of expression. In the rest of this section, we illustrate three typical constructs or *patterns* that we have identified and used successfully to address situations typically found in user interfaces: Stacked state machines, shared transitions, and parallel state machines.

Note that *SwingStates* can be used to nest a state machine inside the state of another state machine, therefore reproducing the basic hierarchical construct of StateCharts. We do not describe this pattern here because we have not found compelling examples yet of its use in user interfaces.

7.2. Stacking state machines

A user interface can often be seen as a system that transforms low-level input events such as clicks and drags into high-level commands such as "cut", "copy", or "paste". As we have seen in section 6, low-level input events can actually be generated by an even lower-level state machine processing device events. At the command level, commands such as "cut", "copy", and "paste" may be processed by a high-level state machine that ignores a paste command if the selection is empty, i.e. unless a cut or copy has been performed before.

We call this arrangement *stacking* because each state machine works at a different level of abstraction: device events, input events, command events. This separation of concern has the added advantage that a state machine can easily be replaced by another one or work in parallel with other ones as described later in this section.

```
CStateMachine sm1 = new CStateMachine() {
    Gesture g = new Gesture();
    CPolyLine ink = canvas.newPolyLine().setDrawable(false);

    public State default = new State() {
       Transition beginGesture = new Press(BUTTON1, "--> gesturing") {
          public void action() {
             g.reset(getPoint());
             ink.reset(getPoint()).setDrawable(true);
          }
       };
    };

    public State gesturing = new State() {
       Transition draw = new Drag(BUTTON1) {
          public void action() {
             g.addPoint(getPoint());
             ink. addPoint(getPoint()).
          }
       };
       Transition endGesture = new Release(BUTTON1) {
          public void action() {
             String commandName =  classifier.classify(g);
             fireEvent(new VirtualEvent(commandName));
             ink.setDrawable(false);
          }
       };
    };

 };
```

Figure 17. A machine transforming mouse events into command names. The machine records a gesture, draws gesture ink and fires events with the commands recognized by the gesture classifier.

```
StateMachine sm2 = new CStateMachine() {
    public State clipboardEmpty = new State() {
       Transition copy = new Event("copy", "--> clipboardFull") { };
    };
    public State clipboardFull = new State() {
       [...]
    };
 };
 sm1.addStateMachineListener(sm2);
```

Figure 18. A state machine that listens to high-level command events fired by state machine `sm1` of Figure 17

For example, it is often good practice for a user interface to offer different modalities for entering the same commands: one may use a graphical menu with the list of commands, a gesture recognition system where each command is specified by the shape of a mouse stroke, or a voice recognition system with one word per command. As shown in figure 16, a different state machine can be used to implement each of the modalities, without the higher-level of the application even knowing how the command was entered. We illustrate here the stacking of a gesture recognition machine and a command handling machine.

SwingStates supports gesture recognition to facilitate the development of pen-based interfaces. It implements Rubine's algorithm for classifying a gesture among a finite vocabulary of gesture classes [43]. To build the gesture vocabulary, the developer uses *SwingStates*' training application to enter the name of each gesture classes and to draw examples for each class. The classifier learns the vocabulary from these examples and stores its data in a file that is loaded by the application at run-time. Adding gesture recognition to an application only requires a few line of code that are typically added to a state machine that draws the ink, as shown in Figure 17.

State machines can communicate with their environments (and with other state machines) using a simple notification mechanism: each state machine holds a list of `StateMachineListener` objects that are notified each time this machine fires an event. `StateMachineListener` is a simple interface that is implemented, in particular, by state machines: when a state machine receives a notification, it simply processes the corresponding event. Figure 18 illustrates stacking: The state machine `sm2` processes the events, such as "copy", fired by `fireEvent` in `sm1`. As illustrated by Figure 16, machine `sm1` for gesture recognition could be replaced or completed by a machine for voice recognition and/or a machine implementing a traditional menu interaction as soon as they fire events understood by `sm2`.

```
class ControlState extends State {
  Transition stopControl = new Release(BUTTON1, ">> start") { };
}

State control = new ControlState() {
  Transition orthoZoom =
            new DragOnTag("fr.lri.swingstates.canvas.Canvas", BUTTON1) {...};
  Transition simplePan = new DragOnTag("javax.swing.JScrollBar", BUTTON1) {...};
  Transition startRateScrolling = new Leave(">> rateScroll") {...};
};

State rateScroll = new ControlState() {
  Transition stopRateScrolling = new Enter(">> control") {...};
  Transition timeOut = new TimeOut() {...};
};
```

Figure 19. Two states share a common transition to implement a control menu

7.3. Sharing transitions among states

When a transition is active in several states, this transition must be duplicated in each of the states in a conventional state machine. For example, in figure 11, the `stopControl` transition is duplicated in states `control` and `rateScroll` (lines 19 and 37). A hierarchical model such as StateCharts can solve this problem by grouping these two states inside a super-state and adding the `stopControl` transition to that super-state, but we find this solution cumbersome.

SwingStates provides an alternative solution, based on inheritance. Since `State` is a Java class, it can be specialized like any other class. If the specialization contains transitions, these transitions will be shared by all instances of the specialized classes. In our example, the developer defines a new class, called `ControlState`, that inherits from `State` and contains the `stopControl` transition. Then, `control` and `rateScroll` are declared as `ControlState` instead of `State` and implement their specific transitions, as illustrated in Figure 19.

7.4. Running state machines in parallel

The exponential growth in the number of states of a state machine typically occurs when combining two state machines, i.e. when each state of one machine must be combined with each state of the other, resulting in a total number of states that is the product of the number of states of each machine. We have found that running the two state machines in parallel, together with some simple communication among them, solves the explosion problem in all practical cases.

Consider for example an interface that uses both a linear menu and a marking menu [44] (a variant of the circular menu described earlier), and that highlights the background of the menu items when the mouse cursor is over it. Rather than trying to combine all three

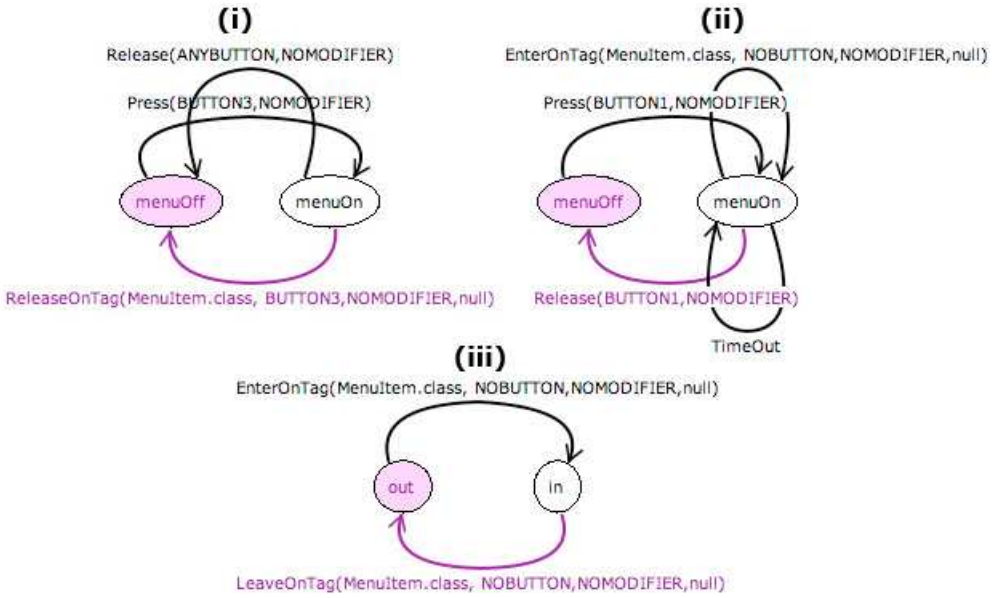

Figure 20. State machines in parallel: (i) machine for linear menu, (ii) machine for marking menu and (iii) machine for highlighting any menu item

interactions into a single state machine, it is much easier, both conceptually and from a programming perspective, to use three state machines (figure 20). The two state machines at the top implement menu selection (linear menu on the left, marking menu on the right), while the state machine at the bottom implements highlighting of any menu item. The three state machines work as follows:

- A linear menu is invoked by pressing the left mouse button. Once invoked, releasing the mouse button on an item selects this item; releasing it out of the menu simply hides it.
- A marking menu is a circular menu which is invoked by pressing the right mouse button. The menu is shown only after a given delay. Releasing the mouse button selects the last visited item, even if the menu was not visible (the user does not need to release the mouse inside the item itself).
- Highlighting simply tracks whether the mouse is inside a visible item or not, and changes the item background color accordingly.

When running the system, one of the menus state machine will run in parallel with the highlighting state machine: in the same way as multiple AWT/Swing listeners of the same type can be attached to the same graphical component, multiple state machines can be attached to the same component, e.g., a *SwingStates* canvas, so that events occuring on this canvas are dispatched to all the attached machines by the standard AWT dispatch thread. While

the state explosion in this example would be manageable if we were to combine the state machines together because there are so few states, it does illustrate the advantage of handling different interaction features in separate state machines. The modular design makes it easy, in particular, to add a new type of menu, such as the control menu seen before, to change the kind of highlighting, e.g., to use animation, or to add a new feature, e.g., tooltips that provide additional information on the item under the cursor.

Parallelism is powerful but can of course lead to concurrent actions on the same object, even though the serialization of event handling eliminates "true" concurrency. Instead of constraining the possible combinations, *SwingStates* provides a tool that visualizes state machines as they run to help developers debug their programs (see Figure 20). This tool uses the `StateMachineEventListener` interface to listen to the internal activity of a given machine. Any object that implements this interface can be attached to a machine and be notified each time the machine is suspended, resumed, reset or triggers a transition, providing yet another way for state machine to control each other.

8. EVALUATING SWINGSTATES

Evaluating a user interface toolkit is a well-known problem with no single satisfying answer [45]. The ultimate measure of success is to be widely adopted by developers, but of course this takes years before a fair assessment can be made. In the meantime, one could conduct usability studies with developers to gather their feedback about the toolkit, but there is no agreed upon set of tasks for evaluating such development tools. One could also report only on purely technical evaluation criteria such as the size and memory footprint of the toolkit or its performance, but these require standardized benchmark to be meaningful.

On a technical level, *SwingStates* contains about 10.000 lines of Java source lines (not counting comments and documentation), and the object library (`jar` file) is 272 Kb. The library together with the examples from the on-line tutorial (`http://swingstates.sf.net/tutorial/`) is 616 Kb. We have not experienced so far any perceivable performance degradation due to the use of state machines when compared with pure Swing applications. The canvas heavily relies on Java2D which unfortunately does not take advantage of the advanced capabilities of today's graphics hardware and therefore can have performance problems when managing a large number of shapes. The canvas was not designed for such situations however and there is plenty of room for optimization using well-known techniques from 2D and 3D interactive graphics. Overall, *SwingStates* has proved to be both efficient and lightweight at the technical level.

The rest of this section presents a more thorough evaluation of *SwingStates* from two perspectives: a software engineering perspective based on design principles, and a user perspective based on our experience in using *SwingStates* for teaching.

8.1. Applying design principles

During the design of *SwingStates*, we used a small number of design principles from software engineering and human-computer interaction to guide our choices. Our goal was to follow Alan Kay's famous quote: "*Simple things should be simple, complex things should be possible*".

First, we wanted to keep the existing simplicity and power of Java and Swing by building on them rather than fighting against them. This provided the main incentive for describing state machines directly in Java as opposed to graphically or with a dedicated language. Similar to the use of operator overloading in the Ubit toolkit [46] to create an expressive syntax for building widgets, *SwingStates* uses Java anonymous inner classes to provide a natural syntax for state machines. The only drawback is the fact that the output state of a transition must be specified by a text string rather than the name of the state. The advantages however vastly outweighs this drawback: state machines take advantage of Java scoping rules and inheritance. Integration with Java and Swing also led us to rely on Java2D for the canvas graphics, to implement the canvas as a Swing widget, and to allow state machines to work with unmodified Swing widgets. Overall, these features greatly facilitate the adoption of *SwingStates* by Java developers, as they leverages their knowledge of the language and the toolkit.

Second, we used two design principles called reification and polymorphism [47]: *reification* consists in turning abstract concepts into concrete objects while *polymorphism* consists in designing functions, objects and methods so that they can be used in different contexts. These concepts are deeply rooted in object-oriented design and languages: reification corresponds to representing concepts as classes and objects, while polymorphism includes ad hoc polymorphism, where two classes can define methods with the same name so that the same method can be applied to objects of different classes, and inclusion polymorphism, where a subclass can redefine a method from its superclass. Reification and polymorphism are also powerful tools for designing user interfaces. For example, tools in tool palettes reify abstract commands such as "change color" or "cut-copy-paste'; a scrollbar reifies the notion of navigating a large document, etc. Many commands and objects in user interfaces are polymorphic, such as "open", which can open a folder or launch an application, or the clipboard, which can hold data of any type (text, image, sound, etc.).

State machines are a perfect example of reification: they reify the notion of interaction, which is otherwise represented by a disconnected set of event handlers. Other user interface toolkits, such as Interactors [14] and SubArctic's dispatch agents [24], also externalize the management of interaction into separate objects. *SwingStates* however turns interactions into first-class objects that can be freely manipulated: multiple state machines can run simultaneously, a state machine can transparently replace another one, etc.

The canvas also illustrates the use of reification since the content of the canvas is a set of objects (the shapes), each with their own geometrical and graphical attributes. By contrast, other canvases such as the original AWT canvas or Swing's JPanel simply provide a drawing area and a set of painting methods. The notion of 2D structured graphics as implemented in *SwingStates* is not new, of course: the Tk canvas, for example, takes the same approach. *SwingStates* however pushes the use of reification and polymorphism further than Tk with its implementation of tags.

SwingStates tags reify the concept of a group and generalize it: groups may overlap, and they can be defined by intension rather than by extension. Tags can also have their own behavior and properties. Finally, by applying the principle of polymorphism to shapes, tags and the whole canvas, most of the methods applicable to a shape (or a widget) are also applicable to a shape tag (or a widget tag) and to the canvas itself, which corresponds to the group of all its shapes. In addition, a state machine can be attached to a single shape or a single tag rather than the whole canvas so that developers can program a specific interaction fo a shape or group of shapes. Altogether, these features drastically simplify the programming of advanced interfaces by providing developers with a high power of expression.

As a final touch, all the modifier methods in *SwingStates* return the object that they have just modified so that their result can be directly used for invoking another method of that object. This makes programs more concise. For example, the five lines below (one assignment and four method calls):

```
CShape e = new CEllipse(10,10,20,30);
e.rotateTo(Math.PI/4);
e.setFillPaint(Color.red);
e.setOutlined(false);
e.addTo(canvas);
```

are replaced by a single instruction (note that the *Canvas* methods of the form *new** act as constructors and return the newly created object instead of the canvas itself):

```
CShape e = canvas.newEllipse(10,10,20,30)
             .rotateTo(Math.PI/4).setFillPaint(Color.red).setOutlined(false);
```

Note that Swing unfortunately does not use this idiom, which makes the initialization code quite verbose when instantiating widgets.

8.2. Benchmarking using advanced interaction techniques

Using benchmarks is a well-known evaluation process in computer science in general and in some areas of Human-Computer Interaction (HCI) such as information visualization [48], but it has not been used to evaluate user interface toolkits. In this section, we report on a benchmark that we have created for that purpose. The benchmark consists of a set of advanced graphical interaction techniques published in the HCI literature over the past ten years, which represent the state-of-the-art in the field. We have used this benchmark with Master's level students as part of an HCI course for the past few years, asking them to implement these techniques with several toolkits. We believe that this benchmark helps evaluate two aspects of a user interface toolkit: the technical capabilities of the toolkit for implementing advanced user interfaces, and its ease-of-use by non-expert developers.

For the past five years, we have taught a Master's level HCI course at Université Paris-Sud that includes a project where students, working in pairs, implement one of a set of interaction techniques based on its description in a research article. Over the years, students have used a wide range of toolkits, including Tcl/Tk, GTk, Swing, even Visual Basic. The results have been disappointing for us and frustrating for the students: they were spending a lot of time on the project but in many cases they were not able to implement the technique properly, sometimes not at all.

Table I. Interaction techniques of our experiment with second year master students

Interaction technique	Number of groups
Marking Menu [44]	2 groups
Flow Menu [49]	1 group
Gesture-based interface [50]	2 groups
Side Views [51]	2 groups
Local tools [52]	2 groups
Alignment stick [53]	2 groups
Pushpins, rulers and pens [54]	2 groups
Magnetic guidelines [55]	1 group

For the past two years (2005-2006 and 2006-2007), we have used *SwingStates* for this project, with roughly the same set of interaction techniques as in previous years. Table I lists the eight techniques that we offered in 2005 together with the research article that the students had to read to understand and implement the technique and the number of groups that chose that technique. Most (but not all) students knew Java and had some experience with Swing. We presented *SwingStates* in a one-hour lecture class and gave the students access to the on-line documentation and a wiki site containing a tutorial and some sample code. We did not have hands-on training with the students. This was a deliberate choice on our part to make the conditions as similar to previous years as possible.

The results were well beyond our expectations. Unlike in previous years when students were using other toolkits, all groups completed their projects in time with very little or no help at all from us, demonstrating the power and simplicity of the canvas and state machines to implement advanced interactions. Figure 21 shows three examples from the 2005 class: alignment stick, local tools and magnetic guidelines. By examining the source code, we observed that students understood the concepts and possibilities of *SwingStates* and put them to good use. For example, even though we had not mentioned the possibility of using several machines during the presentation, the group for the project "Pushpins, rulers and pens" used two machines running in parallel, one for the rulers and one for the pens. The mean length of the programs was 750 lines of source code, including the state machines (250 lines on average per machine). The state machines had between 2 and 9 states and 8 to 32 transitions. Only one group (SideViews) extended the `CShape` class to manage an history of transformations. All groups used tags to group objects, e.g., the tools in a tool palette, and for interaction, e.g., to distinguish tools from other application objects. Some groups however did not understand how the `*OnTag` transitions worked and used `*OnShape` transitions instead with a guard that checks the shape's tag. The version of *SwingStates* that we used that year [56] did not feature intentional tags. Some groups reported difficulties, such as manipulating an object and its descendants, that can now be solved with intentional tags.

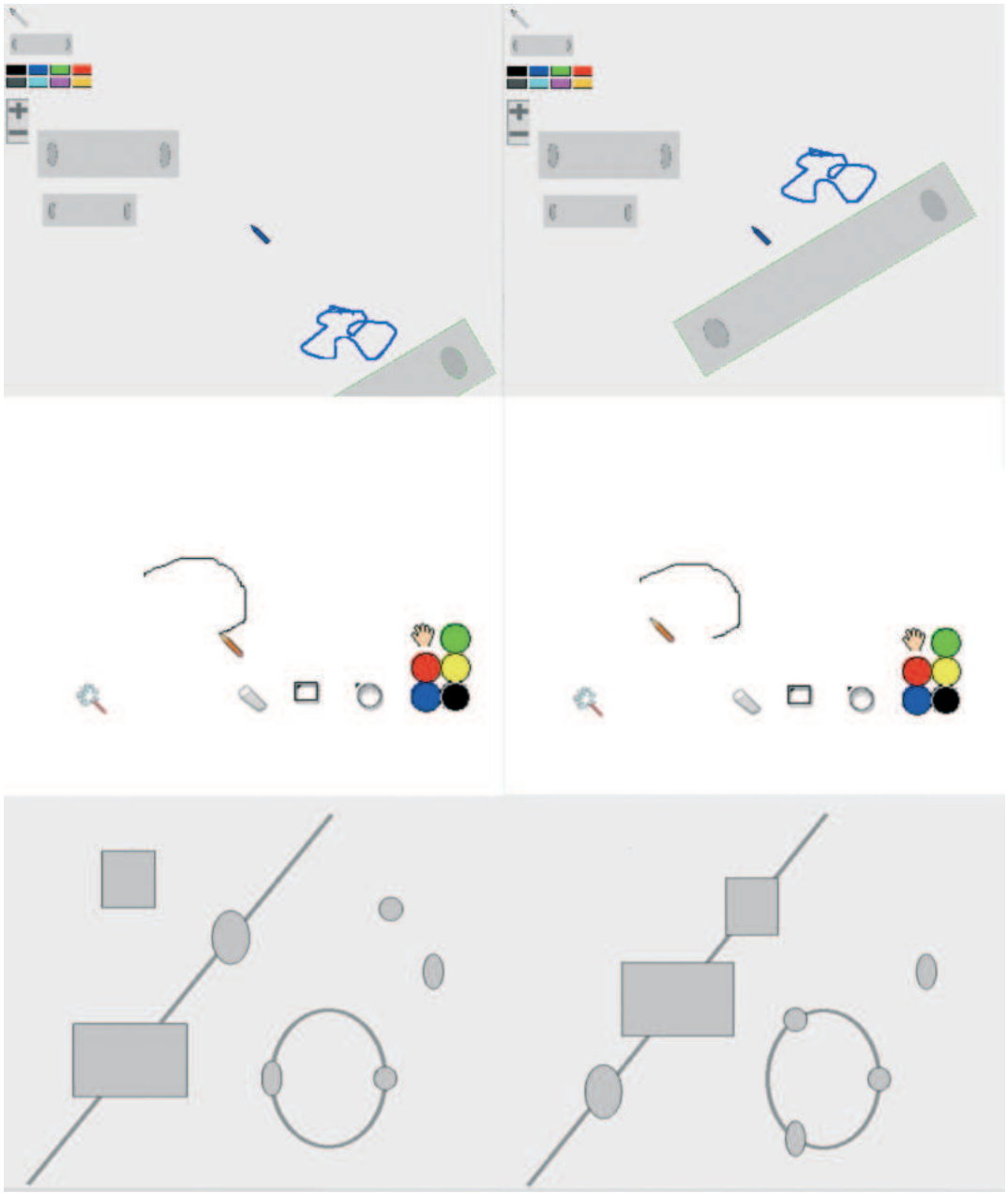

Figure 21. Examples of projects implemented by second year master students

Since this 2005 experiment, we now use *SwingStates* as well in a first year Master's level class "Introduction to HCI". This class is more applied, as students have many TA sessions with hands-on programming. We teach Java2D and Swing in the first part of the class and the students' program a graphical editor for creating and manipulating geometrical shapes. We introduce *SwingStates* in the second part of the class, and the students program the same graphical editor with *SwingStates*. While this was not a fully controlled experiment (we should have had some students start with *SwingStates* and then use Java2D and Swing, but this was not practical in the context of a class), the results are instructive: the *SwingStates* programs were significantly shorter, and even though it was not a graded exercise, a number of students implemented additional tools in the *SwingStates* version of their editor. Two of the three teaching assistants did not have prior experience with *SwingStates*, but we did not detect significant differences in the quality of the student projects from their respective groups (the graded project consisted in developing an iconic file explorer similar to the Mac OS X Finder).

In summary, these evaluations showed that developers can implement state-of-the-art interaction techniques using *SwingStates*, i.e. that at least some "complex" things are possible. It also showed that students (and TAs) with some basic Java experience can easily adopt *SwingStates*, i.e. that the learning curve is gentle.

9. CONCLUSION

SwingStates (`http://swingstates.sf.net/`) is a novel way to use state machines to program advanced interaction techniques with the popular Java Swing toolkit. The tight integration between the state machines, the Java programming language and the Swing toolkit combines power and simplicity, eliciting a programming style where multiple state machines can work independently or in combination to create rich interactions. *SwingStates* can be used both to modify existing Swing widgets and to create new widgets by extending its `Canvas`. *SwingStates* also supports arbitrary input devices to implement state-of-the-art techniques such as those based on bi-manual and pressure-sensitive input. The design of *SwingStates* was guided by a few design principles that provide both simplicity ("simple things are simple") and power ("complex things are possible"). We have used a benchmark approach to assess these properties on *SwingStates*, and we used the toolkit successfully with Master's level students for teaching as well as within our group to explore novel interaction techniques.

Our plans for future work include supporting the dissemination of *SwingStates* and encouraging its use for teaching, research and development. We also want to improve the support for generalized input and consider an OpenGL-based canvas that would provide the level of performance necessary to data-intensive applications such as information visualization. We are interested in exploring higher-level constructs to combine state machines, as well as supporting distributed interfaces where multiple computers cooperate to deliver an interactive experience, such as Rekimoto's pick-and-drop [57], where data is transfered from a PDA to a wall display simply by picking and dropping it with a pen. Finally, while *SwingStates* is tightly linked to Java and Swing, we are interested in applying a similar approach based on state machines to other contexts such as DHTML/Javascript/AJAX for the development of Web applications.

REFERENCES

1. Rumbaugh, J. and Jacobson, I. *The unified modeling language*. 1996 ; University Video Communications
2. Logan S. *Gtk+ Programming in C* Prentice Hall, 2001.
3. Myers B. Separating application code from toolkits: eliminating the spaghetti of call-backs. *UIST '91: Symposium on User Interface Software and Technology* 1991; ACM Press, New York, NY, USA; 211–220.
4. Callahan J, Hopkins D, Weiser M. and Shneiderman B. An empirical comparison of pie vs. linear menus. *CHI '88: Proceedings of the SIGCHI conference on Human Factors in computing systems* 1988 ; ACM Press, New York, NY, USA; 95–100.
5. StateWORKS.
   http://www.stateworks.com [28 May 2007].
6. State Machine Compiler page.
   http://smc.sourceforge.net [28 May 2007].
7. State Chart XML (SCXML): State Machine Notation for Control Abstraction.
   http://jakarta.apache.org/commons/scxml/index.html [28 May 2007].
8. Newman W. A system for interactive graphical programming. *Seminal Graphics: PionShaped the Field* 1968 ; ACM Press, New York, NY, USA; 409–416.
9. Jacob R. Using Formal Specifications in the Design of a Human-Computer Interface. *Communications ACM* 1983 ; **26**(4):259–264.
10. Wasserman A . Extending State Transition Diagrams for the Specification of Human-Computer Interaction. *IEEE Transactions on Software Engineering* 1985 ; **11**(8):699–713.
11. Beaudouin-Lafon M. User Interface Management Systems : Present and Future. *Eurographics'91, Invited State of the Art Report, Focus on Computer Graphics Series* 1991 ; Springer-Verlag, London, UK; 197–223.
12. Buxton W. A three-state model of graphical input. *INTERACT '90: Proceedings of the IFIP TC13 Third Interational Conference on Human-Computer Interaction* 1990 ; North-Holland; 449–456.
13. Hinckley K. and Czerwinski M. and Sinclair M. Interaction and modeling techniques for desktop two-handed input. *UIST '98: Proceedings of the 11th annual ACM symposium on User interface software and technology*, 1998 ; ACM Press, New York, NY, USA; 49–58.
14. Myers B. A New Model for Handling Input. *ACM Transactions on Information Systems* 1990 ; ACM Press, New York, NY, USA; **8**(3):289–320.
15. Myers B, Giuse D, Dannenberg R, Vander Zanden B, Kosbie D, Pervin E, Mickish A and Marchal P. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer* 1990 ; **23**(1):71–85.
16. Myers B, McDaniel R, Miller R, Ferency A, Faulring A, Kyle B, Mickish A, Klimovitski A and Doan P. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering* 1997 ; **23**(6):347–365.
17. Jacob R, Deligiannidis L, and Morrison S. A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Graphics* 1999 ; **6**(1):1–46.
18. Blanch R and Beaudouin-Lafon M. Programming Rich Interactions using the Hierarchical State Machine Toolkit. *AVI '06: Conference on Advanced Visual Interfaces* 2006 ; ACM Press, New York, NY, USA; 51–58.
19. Eng E. Qt GUI Toolkit: Porting graphics to multiple platforms using a GUI toolkit. *Linux J.* 1996; (31):article 2.
20. Bederson B, Grosjean J and Meyer J. Toolkit Design for Interactive Structured Graphics. *IEEE Transactions on Software Engineering* 2004 ; **30**(8):535–546.
21. Pietriga E. A Toolkit for Addressing HCI Issues in Visual Language Environments. *VL/HCC'05: Symposium on Visual Languages and Human-Centric Computing* 2006 ; IEEE Computer Society, Washington, DC, USA; 145–152.
22. Fekete JD. InfoVis. *InfoVis* 2004; **00**:167–174.
23. Heer J, Card S and Landay J. Prefuse: a toolkit for interactive information visualization. *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems* 2005; ACM Press, New York, NY, USA; 421–430.
24. Hudson S, Mankoff J and Smith I. Extensible input handling in the subArctic toolkit. *CHI '05: Conference on Human Factors in Computing Systems* 2005; ACM Press, New York, NY, USA; 381–390.
25. Pook S, Lecolinet E, Vaysseix G and Barillot E. Control menus: excecution and control in a single interactor. *CHI '00: Extended Abstracts on Human Factors in Computing Systems* 2000 ; ACM Press, New York, NY, USA; 263–264.

26. Strauss, P.S. IRIS Inventor, a 3D graphics toolkit. *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications* 1993 ; ACM Press, New York, NY, USA; 192–200.
27. Bérard F. The GML canvas: Aiming at Ease of Use, Compactness and Flexibility in a Graphical Toolkit. *Technical Report TR-IMAG-CLIPS-IIHM-200601* 2006 ; CLIPS-IMAG.
28. Ousterhout J. *Tcl and the Tk Toolkit* Addison-Wesley, 1994.
29. Chang B-W and Ungar D. Animation: from cartoons to the user interface. *UIST '93: Proceedings of the 6th annual ACM symposium on User interface software and technology* 1993 ; ACM Press, New York, NY, USA; 45–55.
30. Appert C and Fekete J.D. OrthoZoom scroller: 1D multi-scale navigation. *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems* 2006 ; ACM Press, New York, NY, USA; 21–30.
31. Apitz G and Guimbretière F. CrossY: a crossing-based drawing application. *UIST '04: Symposium on User Interface Software and Technology* 2004 ; ACM Press, New York, NY, USA; 3–12.
32. Ramos G and Balakrishnan R. Zliding: fluid zooming and sliding for high precision parameter manipulation. *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology* 2005; ACM Press, New York, NY, USA; 143–152.
33. Appert C, Beaudouin-Lafon M and Mackay W. Context matters: Evaluating interaction techniques with the CIS model. *HCI '04: People and Computers XVIII - Design for Life* 2004 ; Springer-Verlag, London, UK; 279–295.
34. Kabbash P, Buxton W and Sellen A. Two-handed input in a compound task. *CHI '94: Conference on Human factors in computing systems* 1994; ACM Press, New York, NY, USA; 417–423.
35. JInput, The Java Input API Project. http://jinput.dev.java.net/ [8 July 2007]
36. Mackay W, Appert C, Beaudouin-Lafon M, Chapuis O, Du Y, Fekete JD and Guiard Y. Touchstone: exploratory design of experiments. *CHI '07: Conference companion on Human factors in computing systems* 2007; ACM Press, New York, NY, USA; 1425–1434.
37. Green M. A survey of three dialogue models. *ACM Transactions on Graphics* 1986 ; **5**(3):244–275.
38. Green M. The University of Alberta user interface management system. *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques* 1985; ACM Press, New York, NY, USA; 205–213.
39. Harel D. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 1987 ; **8**(3):237–274.
40. Wellner P. Statemaster: A UIMS based on statechart for prototyping and target implementation. *CHI'89: Conference on Human Factors in Computing Systems* 1989 ; ACM Press, New York, NY, USA; 177–182.
41. Palanque P, Bastide R, Dourte L and Sibertin-Blanc C. Design of User-Driven Interfaces Using Petri Nets and Objects. *CAiSE '93: Proceedings of Advanced Information Systems Engineering* 1993 ; Springer-Verlag, London, UK; 569–585.
42. Bastide R, Navarre D and Palanque P. A tool-supported design framework for safety critical interactive systems. *Interacting with Computers* 2003; **15**(3):309–328.
43. Rubine D. Specifying gestures by example. *SIGGRAPH '91: Conference on Computer Graphics and Interactive Techniques* 1991 ; ACM Press, New York, NY, USA; 329–337.
44. Kurtenbach G and Buxton W. User learning and performance with marking menus. *CHI '94: Conference on Human factors in computing systems* 1994; ACM Press, New York, NY, USA; 258–264.
45. Olsen, D.R. Evaluating User Interface Systems Research. *UIST '07: Symposium on User Interface Software and Technology* 2007 ; ACM Press, New York, NY, USA; 251–258.
46. Lecolinet E. A molecular architecture for creating advanced GUIs. *UIST '03: Proceedings of the 16th annual ACM symposium on User interface software and technology* 2003; ACM Press, New York, NY, USA; 135–144.
47. Beaudouin-Lafon M and Mackay W. Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. *AVI '00: Conference on Advanced Visual Interfaces* 2000 ; ACM Press, New York, NY, USA; 102–109.
48. Plaisant C. The challenge of information visualization evaluation. *AVI'04: Conference on Advanced Visual interfaces* 2004 ; ACM Press, New York, NY, USA; 109–116.
49. Guimbretière F and Winograd T. FlowMenu: combining command, text, and data entry. *UIST '00: Symposium on User Interface Software and Technology* 2000; ACM Press, New York, NY, USA; 213–216.

50. Landay J. SILK: Sketching Interfaces Like Krazy. *CHI '96: Conference companion on Human factors in computing systems* 1996; ACM Press, New York, NY, USA; 398–399.
51. Terry M and Mynatt E. Side views: persistent, on-demand previews for open-ended tasks. *UIST '02: Symposium on User interface Software and Technology* 2002 ; ACM Press, New York, NY, USA; 71–80.
52. Bederson B, Hollan J, Druin A, Stewart J, Rogers D and Proft D. Local tools: an alternative to tool palettes. *UIST '96: Symposium on User interface Software and Technology* 1996 ; ACM Press, New York, NY, USA; 169–170.
53. Raisamo R and Räihä K. A new direct manipulation technique for aligning objects in drawing programs. *UIST '96: Symposium on User interface Software and Technology* 1996 ; ACM Press, New York, NY, USA; 157–164.
54. St. Amant R. and Horton T. Characterizing tool use in an interactive drawing environment. *SMARTGRAPH '02: Symposium on Smart Graphics* 2002 ; ACM Press, New York, NY, USA; 86–93.
55. Beaudouin-Lafon M. Designing interaction, not interfaces. *AVI '04: Conference on Advanced Visual Interfaces* 2004 ; ACM Press, New York, NY, USA; 15–22.
56. Appert C and Beaudouin-Lafon M. SMCanvas: augmenter la boîte à outils Java Swing pour prototyper des techniques d'interaction avancées. *Conférence Francophone sur l'Interaction Homme-Machine* 2006 ; ACM Press, New York, NY, USA; 99–106.
57. Rekimoto J. Pick-and-drop: a direct manipulation technique for multiple computer environments. *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology* 1997 ; ACM Press, New York, NY, USA; 31–39.