The Tycho Slate: Complex Drawing and Editing in Tcl/Tk

H. John Reekie and Edward A. Lee
School of Electrical Engineering and Computer Sciences
University of California – Berkeley
Berkeley CA 94720
{johnr,eal}@eecs.berkeley.edu

Abstract

This paper introduces the Slate package, which has been developed as part of the Tycho project at UC Berkeley. The Slate is layered over the Tcl/Tk canvas, and contains features that we believe to be useful for implementing complex graphical editing and visualization widgets. The first key feature is the ability to define new item types in Tcl. The second is an implementation of the concept of interactor*, which abstracts low-level mouse events into self-contained objects. The third is access to and modification of items based on their shape, rather than raw coordinates. Combined with a straight-forward implementation of the model-view-controller architecture, the Slate is capable of implementing quite sophisticated graphical editors.*

1 Introduction

The Tk canvas provides a simple but powerful set of structured graphics primitives, and is perhaps the easiest toolkit available for drawing simple 2D graphics. When we started to use the Tk canvas with a view towards implementing various graphical diagram editors, however, we realized that we needed a more powerful layer of abstraction.

To illustrate, figure 1 shows a mock-up visual program of the kind that we were interested in implementing. This is a sample of a language developed in [10]. The rectangle marked "let" encloses an expression, and the value of the expression is indicated by the arrow connecting the two triangular "terminals." The let-expression itself is connected to another function box. Items within the let-expression can be moved only within that frame, while moving the whole frame will move everything contained within it. This combination of hierarchy and complex user interaction necessitated a higher-level framework than the raw canvas.

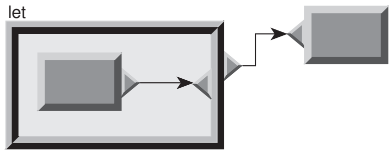

Figure 1: A fragment of a visual program

We decided very early on that our implementation would be only in [incr Tcl], in order to guarantee portability. More recently, we used the namespace facility of Tcl 8.0 to port the code to Tcl 8.0, and it now runs in either Tcl and [incr Tcl]. Although the Tcl-only decision presented its own set of implementation challenges, we were able to achieve acceptable performance and the portability we desired. The Slate should, in fact, work with *any* canvas extension, since it uses only the standard canvas interface provided by Tcl/Tk.

After some experimentation, we ended up with a package that implements the following:

User-defined items The key feature that makes the Slate useful is the ability to define new item types in Tcl. Items can be composed recursively, producing a straight-forward visual hierarchy, as is common in many graphics packages (see, for example, [2]). All of the canvas methods are rewritten to handle hierarchical items.

Item shapes Every item has a shape, such as *point*, *rectangle*, *polygon*, or a custom-designed shape. Items can be queried for the coordinates of a *feature*, such as the north-east corner or the second vertex. Items can be requested to move one or more features, reshaping the item.

Interactors Event-handlers can be bound to any level of the visual hierarchy. In addition, we implemented a more abstract and more powerful user interaction framework, in which par-

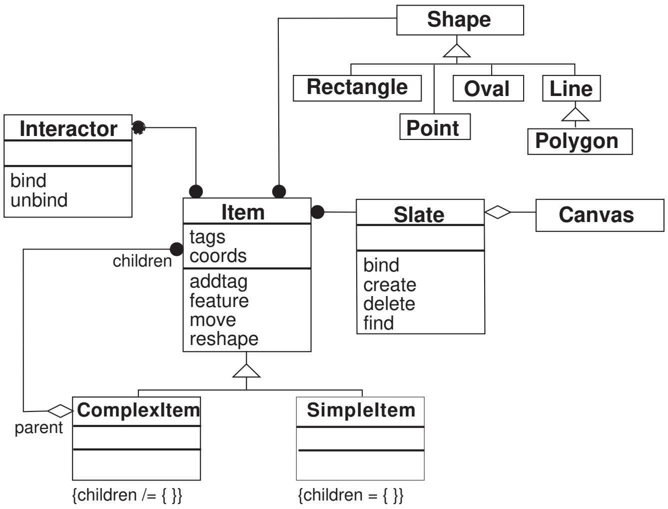

Figure 2: The key framework classes

ticular sequences of user interaction events are abstracted into objects called *interactors*.

We also added other useful methods, such as highlighting and selection. Unlike many research toolkits, we did not implement a constraint system, for reasons detailed in a later section.

2 The core graphics framework

The Slate is implemented as a set of classes, organized around one key class called *Slate*. We have tried to make the Slate a plug-in replacement for the Tk canvas. A simple example:

```
set slate [::tycho::slate .s \
      -height 300 -width 400 \
      -background white]
pack $slate -fill both -expand 1
```

Figure 2 shows a conceptual representation, in the Object Modeling Notation [11], of the key classes. The diagram is conceptual only, because many of these classes do not in fact exist – in the implementation, they are faked using Tcl procedures, canvas item tags, and associative arrays.

The *Slate* class is wrapped around a Tk canvas. It contains an arbitrary number of *Item*s, which are graphical elements that appear on the screen. An Item is either a *ComplexItem*, which is in turn an aggregation of Items, or a *SimpleItem*, which represents a single Tk canvas item such as a line or rectangle. Each item has a *Shape* (see section 2.3), and can be operated on by an arbitrary number of *Interactor*s (see section 3.2).

2.1 Complex item classes

To add a new item type to the slate, the programmer subclasses the *ComplexItem* class. Once defined, items of the new type can be created and manipulated just like regular Tk canvas items. For example, one of the item types that we supply with the Slate is called *Frame*, because it mimics the appearance of the Tk *frame* widget. To create a new frame on a slate, we can execute code such as this:

```
set frame [$slate create Frame \
            50 50 100 100 \
            -color green -relief ridge]
```

which produces the item shown at the top left of figure 3. This item behaves like any other Tk canvas item – for example, we can change its coordinates:

```
$slate coords $frame 70 70 140 120
```

We can move it:

```
$slate move $frame 40 20
```

We can get a list of items overlapping a given region of the canvas, which will (in this case) include this item:

```
set found [$slate find \
      overlapping 100 100 200 200]
```

Figure 3 shows several other complex item types. At the top right is a *Solid*, which is a polygon with a pseudo-3D border like *Frame*; at the bottom left is a *LabeledRectangle*, which is a rectangle with a label and arbitrary graphics nested within it (in this

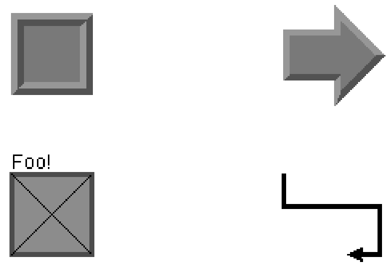

Figure 3: Some pre-defined complex items

case, two lines); at the bottom right is a *SmartLine*, which is a line that, given two end-points and the directions at those ends (n, s, e, or w), draws itself as one or more orthogonal segments. We emphasize that these items are a sample of those that we found useful for building graphical editors, and that it is quite simple to create new item types or to extend existing types by subclassing.

2.2 Constructing hierarchy

Any subclass of *ComplexItem* can add items to itself, thus creating a recursive hierarchy of items. For example, a *Frame* item consists of four simple items: a rectangle for the central surface, two polygons for the "lit" and "shaded" borders, and a transparent rectangle that is used as a place-holder for the coordinates of the whole *Frame*.

In addition to creating hierarchy by creating new item types, an instance of the *ComplexItem* class can have arbitrary sub-items added to it. To illustrate, we can create a blank complex item:

```
set citem [$slate create ComplexItem \
            50 50 100 100]
```

The coordinates give the region to be occupied by the item so that methods such as *coords* will operate correctly. Now we can add items to it. For the sake of example, let's add a pair of lines and an oval to it:

```
$slate createchild $citem line \
             50 50 100 100
$slate createchild $citem line \
             50 100 100 50
$slate createchild $citem oval \
             60 60 90 90 -fill green
```

The item that results is shown at the left of figure 4. Like any slate item, this item responds to methods that move and scale it. For example,

```
$slate coords $citem
```

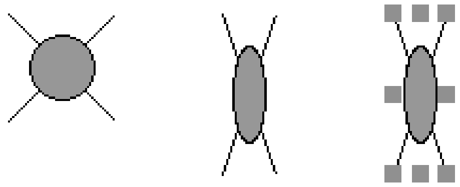

Figure 4: A dynamically-constructed complex item

will return {`50 50 100 100`}. The code

```
$slate scale $citem 50 50 0.5 1.5
```

will scale the item, producing the item at the center of figure 4.

In general, arbitrarily complex items can be built up this way. Figure 5 shows one such hierarchy, similar to those we use in one of our graphical editors. As a general rule, a programmer should define a new subclass of *ComplexItem* when a particular graphical representation is used again and again, and use dynamic composition of items, such as just given, when items are combined as part of the editing operations in a graphical editor.

2.3 Shape

Each item on a slate has a *shape*. Shapes provide more sophisticated control over the coordinates of an item than just its raw coordinates. Simple items have a shape that cannot be changed; complex items have a shape that is determined by the class defining that item. Predefined shapes mimic the primitive canvas item types: *point*, *rectangle*, *oval*, *line*, and *polygon*.

An item with a given shape has a set of attributes called *features*. Features are inspired by Gleicher's work on constraint-based graphics [3]. A feature is typically a point location on the item. An item can be queried to find the value of a feature, and the feature can be moved to change the shape of the item. For rectangular items, the default features are its center, the four corners, and the four edges; for lines, the features are the vertices of the line.

For example, to find the coordinates of the north-west corner of a rectangular item, we could execute:

```
set northwest [$slate feature $frame nw]
```

To reshape the item by moving the north-west corner left and down ten pixels, we could execute:

```
$slate reshape $frame -10 10 nw
```

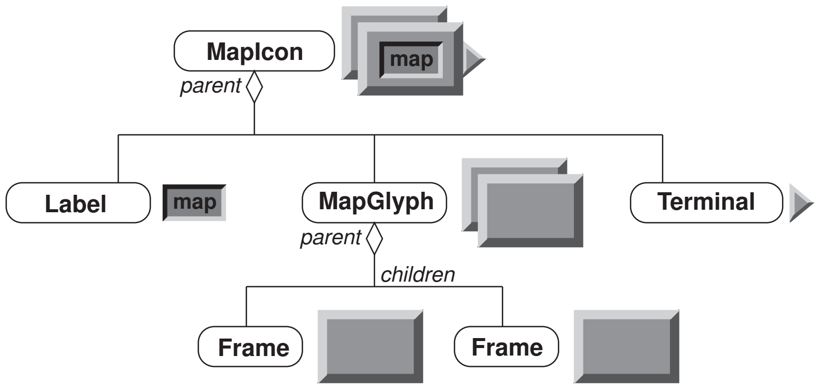

Figure 5: A sample visual hierarchy

All features can be read, but not all can be set; *center*, for example, can be read but not set. Any feature that can be set can also be *grappled* – that is, have a grab handle attached to it. For example, executing the code:

```
$slate grapple $citem
```

will add grab handles to the four corners and edges of the given item. The effect of executing this code is shown on the right of figure 4 – this item can be resized by dragging any of the grab handles.

The *Shape* classes are implemented as collections of class procedures – each class handles feature queries and reshaping of all items with that shape. Complex items can choose their own set of features by overriding their shape-related methods. For example, the *Terminal* item in figure 5 has features called "origin" and "terminal," which are its base and connection point respectively.

2.4 Tags

Tagging in the Slate is a fairly straight-forward extension of the way that tags are implemented on the Tk canvas. Any item can be tagged. For example, given some item *citem*, we can write

```
$slate addtag "fred" withtag $citem
```

Some time later, we could write

```
$slate move "fred" -10 0
```

which would move *citem*, including all of its components, ten pixels to the left. (Any other item tagged with "fred" will also be moved.) Any node in the hierarchy can be tagged in this way, and performing an operation on the tag will operate on the corresponding subtree of the hierarchy.

3 Interaction mechanisms

The visual hierarchy provides an elegant means of constructing drawings, but does not by itself provide user interaction. To effectively support construction of visual language editors, we need to provide ways of adding user interaction.

3.1 Bindings

The first interaction mechanism supported by the slate directly mimics the binding mechanism of the Tk canvas. With this mechanism, a command can be bound to an event and an item. For example, executing

```
$slate bind $citem <Button-1> {puts !!}
```

will add a binding to *citem*. Whenever the mouse is clicked on item *citem*, the string "!!" is printed to the console.

In the presence of a visual hierarchy, bindings take on some subtle complications. In figure 5, for example, I want a click on either frame to be handled by the top-level item. Dragging these items should move the whole tree of items. The terminal item, however, should respond differently – in this case, I want clicking and dragging on the terminal item to create a new arrowed line and extend the end of the line to follow the cursor. Also, dragging on the text label will sometimes need to select a region of the text.

Our solution is to *mark* nodes of the tree: only marked nodes are able to respond to user input. Figure 6 illustrates a marked tree, in which nodes a and e are marked. With respect to any node, its *root node* is the root of the lowest marked sub-tree containing it. The top-level node is the root of the whole tree, and is implicitly marked. Conceptually,

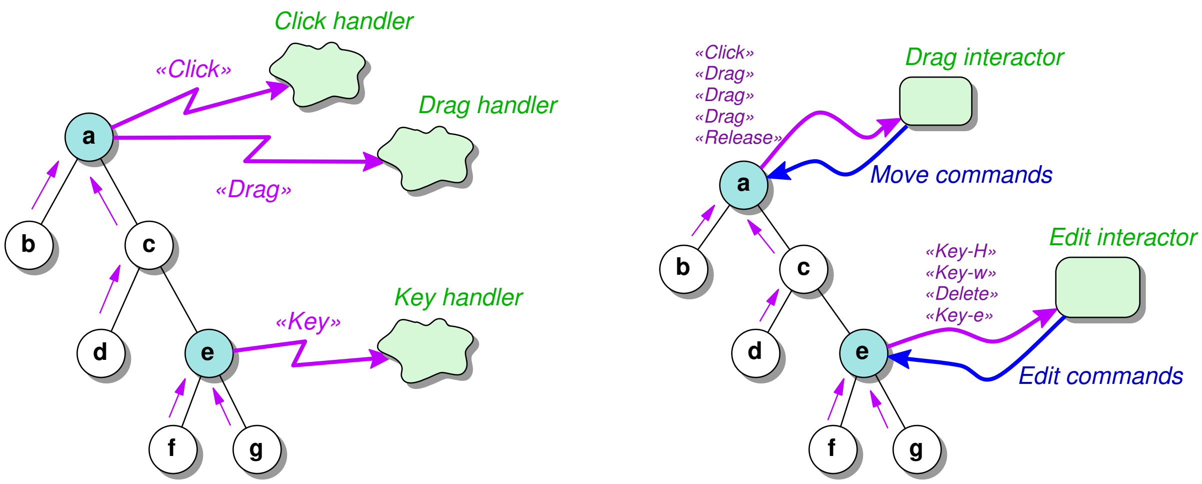

Figure 6: Events and interactors in the visual hierarchy

when an event occurs on an item, the event is propagated up the tree until it reaches a marked node, at which point the event is handled. This is illustrated at the left of figure 6: node a will handle events from itself, b, c, and d; node e will handle events from itself, f, and g.

In the current implementation of the Slate, nodes can only be marked when they are created. Returning to our earlier example, we can create a child item of a *ComplexItem* that responds to user input with code such as this:

```
set flag [$slate createrootchild \
      $citem rectangle \
      40 50 50 60 -fill red]
$slate bind $flag <Button-1> \
      {puts Foo}
```

Events can also be used with tags. For example, we can write

```
$slate bind "fred" <Button-1> \
         {puts "Clicked fred!"}
```

and any top-level or marked child item tagged with *fred* – that is, it and the nodes for which it handles events – will respond to the event.

If used without any hierarchy, the slate thus provides the same mechanism as the canvas for event-handling. The visual hierarchy aids more complex item construction without discarding this powerful mechanism. However, the binding mechanism is low-level, and complex user interaction built on this mechanism very quickly mushrooms into spaghetti-like code.

3.2 Interactors

The second interaction mechanism is based on *interactors*, proposed by Myers in 1990 [7] and implemented in the Garnet toolkit and its successor, Amulet [8, 9]. Interactors abstract user interaction from the lower-level events upon which they are built, and in the process modularize the code and make it more re-usable.

An interactor is an object that intercepts events and translates them into operations on a *target* item. For example, a *Follower* interactor – so called because it "follows" the mouse – translates mouse events into calls to the *moveclick*, *movedrag*, and *moverelease* methods of the slate. To create a Follower interactor, we can execute, say:

```
set follower \
    [$slate interactor Follower]
```

To make an interactor operate on an item, we *bind* the interactor to that item. For example,

```
$follower bind $frame -button 1
```

Now, dragging the frame item with the mouse makes it move – simple! To stop the item from responding to the mouse, unbind the interactor:

```
$follower unbind $frame -button 1
```

The right side of figure 6 illustrates two interactors bound to nodes of a hierarchy. Interactors can be cascaded to create more complex interaction. For example, an interactor that moves an object only within a certain region of the slate can be cascaded with an interactor that quantizes movement to ten-pixel steps.

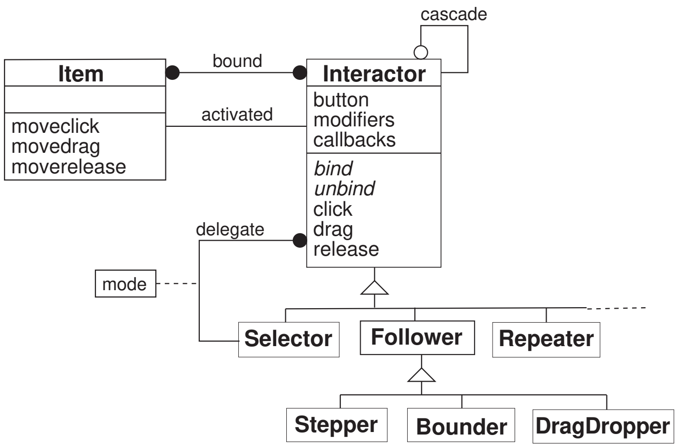

Figure 7: The interactor classes

A more complex combination is achieved with the *Selector* interactor, which manages a graphical selection in the same manner as typical drawing programs. When items become selected, the Selector *delegates* interaction events to another interactor according to various conditions that can be set up by the programmer. For example, it might forward mouse and keyboard events to a *LineEditor* interactor if there is only a single text item selected, and forward mouse events to a *Follower* interactor if more than one item is selected.

Interactors allow user interaction code to be modularized, and, just as importantly, reused. In addition, they allow highly dynamic user interaction, such as changing the effect of the mouse depending on the context (such as the number of items selected). This kind of dynamic modification of user interaction would be very difficult if coded directly using event bindings. Some additional examples of interactors are given in section 5.

4 Implementation notes

As mentioned previously, the ComplexItem "class" and its subclasses, which define new item types, are not really implemented using classes. Because the natural implementation of complex visual elements in Tk utilizes item tags to avoid recursive tree-walks, we flattened the object structure as well. (In a language that supports fine-grained objects, such as Java, we would use separate objects and recursive tree-walk algorithms.)

Each "class" is thus a collection of procedures that accept a slate, the contained canvas, and an item ID as the first three arguments. Methods of the slate call procedures in the appropriate namespace when necessary – for example, a command such as

```
$slate create Frame 40 40 80 80
```

will call the *construct* procedure in the Frame namespace. To simulate inheritance, each namespace contains an associative array mapping method names to the appropriate procedure, which is overwritten in each "sub-class."

We used some other tricks as well. For example, to access option variables and instance variables, we index a shared array by a combination of a name and the ID of the item. This allows the Slate uniform access to the internals of complex items, regardless of class. For example, a procedure in Frame that accesses its *color* option would use:

```
set foo $option(color$id)
```

We use tags to simulate hierarchical complex items. Although this seems conceptually simple, actually doing it in the presence of event bindings is a little tricky. Each complex item is assigned a unique ID when it is created – we use an integer preceded by an underscore (the Tk canvas uses plain integers). Every simple item within a complex item is tagged with that ID:

> *Tagging rule 1*: Every simple item within a complex item is tagged with that item's ID.

The corollary is that every simple item is tagged with the IDs of all items above it in the hierarchy – this makes moving a subtree easy. Now, user interaction often requires finding the root item containing a simple item. To make this efficient, each simple item within a hierarchy is given a special tag:

Tagging rule 2: Every simple item contained in a complex item is tagged with the ID of its root prefixed by "!".

The corollary of this rule is that any simple item that does not have a tag starting with "!" is not in a hierarchy. With these rules, it is easy (conceptually – the implementation is a little tricky) to find, move, and manipulate complex and simple items.

Now, some additional rules are needed to effectively deal with event bindings (and, by extension, interactors). Since a simple item can have only one root, it can have *at most* one tag beginning with "!". Therefore, when binding an event to a complex item, we follow this rule:

Binding rule 1: To bind an event to a complex item, bind to the tag constructed by prefixing its ID with "!".

The net effect is exactly as it would be if events were propagated up the hierarchy until a marked node were found. Finally, events can also be bound to tags, which is the same as binding to a tag on the canvas:

Binding rule 2: To bind an event to a tag, bind the event to that tag on the canvas.

The implementation of most slate methods based on these rules is reasonably straight-forward. Each method tests for three types of argument – a simple item ID, a complex item ID, or a tag – and acts accordingly. For example, a typical method has the pattern:

```
if { [string match {[0-9]*} $tag] } {
    # Process a canvas item
    ...
} elseif { [string match {_*} $tag] } {
    # Process a complex item
    ...
} else {
    # It's a tag: find matching items
    set items [find withtag $tag]
    ...
}
```

5 Building useful tools

The Slate itself is only part of the infrastructure needed to build useful tools. In this section we briefly describe our implementation of some tools that use the Slate.

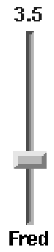

Figure 8: A custom slider widget

5.1 Custom widgets

One of our first uses of the Slate was to build a custom slider widget, shown in figure 8. (It looks much better in real life when you have several lined up together.) This slider widget mimics the sliders used in audio control equipment in appearance, but mimics the Tk *scale* widget in behavior. Four interactors are used to produce the desired user interaction.

Figure 9 shows code for a simplified version of this widget. The four sections of this code:

1. Create the four items show in the diagram: two text labels and two pseudo-3D rectangles.

2. Create and bind a *Bounder* interactor, which moves the slider bar up and down and keeps it within the desired limits.

3. Create and cascade a *Stepper* interactor, which quantizes movement to multiples of 0.5 (in this example).

4. Define a procedure that is called whenever the bar is moved. The procedure calculates the value represented by the bar, and updates the numeric text item.

In the real Slider widget, there are two other interactors that i) step the slider towards the mouse if the left button is clicked on the background, and ii) cause the bar to jump to the position of a button-2 click and then follow the cursor.

All told, constructing this widget was relatively easy, and we didn't have to write a single event binding.

5.2 Graphical editors

Figure 10 shows a snapshot of one of the graphical editors constructed using the Slate. This editor is the front-end for Ptolemy II, a new version

```
# Create the display elements
set value [$slate create text 50 20 -text 0 -anchor s -fill blue]
set trough [$slate create Frame 48 23 52 143 -color darkgrey \
        -borderwidth 2 -relief sunken]
set bar [$slate create Frame 40 132 60 142 \
        -color darkseagreen -borderwidth 3]
set label [$slate create text 50 150 -text "Fred" \
                -anchor n -justify center]

# The bounder moves the bar along the trough
set bounder [$slate interactor Bounder \
        -dragcommand "updateWidget $slate $bar $value" \
        -constrain y -bounds {0 24 0 142}]
$bounder bind $bar -button 1

# The stepper quantizes movement to increments of 0.5
set stepper [$slate interactor Stepper -stepsize [expr 108.0/22]]
$bounder cascade $stepper

proc updateWidget {slate bar value args} {
    set position [expr [lindex [$slate coords $bar] 1] + 5]
    set x [expr (137.0-$position)/108.0 * 10.0]
    $slate itemconfigure $value -text [format %.1f $x]
}
```

Figure 9: A simple example of a custom widget

of Ptolemy [6] being written in Java. The particular system shown is a second-order continuous-time simulation, written by Jie Liu of UC Berkeley. In this editor, icons can be selected from a library stored as ASCII text, which describes each icon in terms of the ComplexItem type used to draw it, the number and location of terminals, the label, and the graphics to draw upon the surface of the icon.

Once placed, icons can be selected and moved around. (This is done using the Selector and Follower interactors.) When the mouse moves over an unconnected terminal of an icon, a *DragDropper* interactor is activated, which highlights the terminal to indicate that it is "ready." If the mouse is clicked on the terminal, the DragDropper creates a new SmartLine item, and reshapes the line so that the end follows the cursor. When the end of the line moves over a terminal, the DragDropper activates a call-back to test if the terminal is a suitable "drop target." If it is, it snaps [4] the end of the line to the connection point of the terminal, altering the shape of the line to make it join at the expected angle.

Internally, this graphical editor uses a variant of the model-view-controller architecture (see, for example, [1]). As the user places icons and connects terminals, the interactors forward events to either an edge controller or a vertex controller. The architecture is shown in figure 11. The controller first decides what the user interaction means in terms of the underlying semantics of the visual program, and modifies the *semantic model* accordingly (in this example, the semantic model is a directed graph). The controller also decides what visual aspects of the program have changed, such as moving the end of a line, or adding a new icon. It modifies the *layout model* accordingly, which in turn notifies the *view* containing the slate, which in turn updates to reflect the new appearance.

This seems, at first, somewhat complicated, but our experience indicates that it does leads to highly modular and customizable editors. We are, however, still gaining experience with this architecture. Interactive response is very good, partly because the interactor model is able to optimize incremental mouse movements while items are being dragged about on the screen.

Finally, we note here our observations on the use of *constraints*, as included in many experimental toolkits ([8, 9], for example). In constraint systems, the programmer sets up constraints between graph-

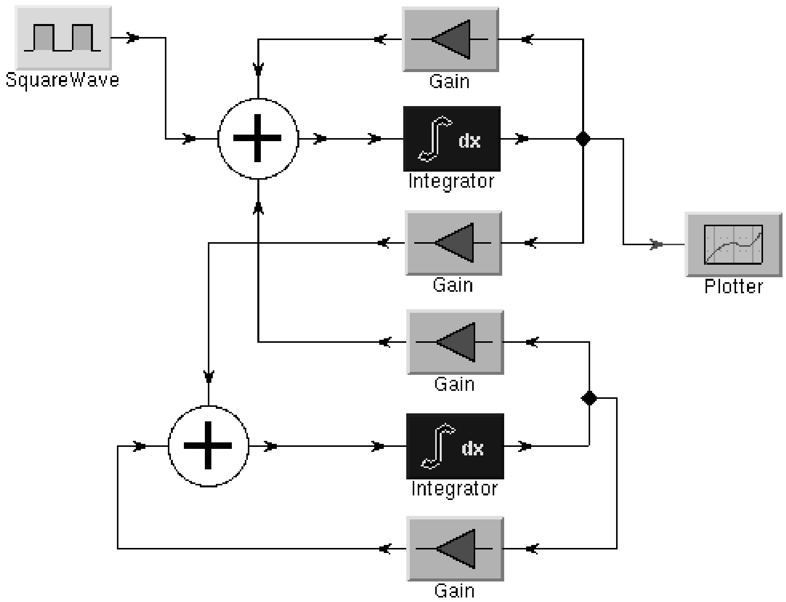

Figure 10: A snapshot of a graphical editor

ical items, which the constraint system attempts to maintain as conditions change. For example, our editor could have constraints set up such that the ends of attached lines move when an icon is moved.

We wrote a simple constraint system early on, and found that writing constraints to catch the right conditions and to avoid cycles was tricky. We ended up trashing the constraint system – simple though it was – when we realized that the model-view-controller architecture offers a better and simpler alternative. If you look at figure 11, you can see that, as an icon is moved, the layout model must be modified for attached edges as well as the icon. How do we know what edges to move? Why, by looking at the semantic model!

Thus, we concluded that, in interfaces that have an underlying semantic model, using that model with an appropriate controller object is a more straight-forward solution than general-purpose constraint solvers. In addition, since we have a comprehensive set of interactors, it is also straight-forward to implement purely-syntactic constraints by appropriately configuring interactors. For example, keeping the slider in figure 8 on the right track does not require constraints, just the right interactor.

6 Concluding remarks

The Slate is, we feel, a powerful and useful tool that provides a significant increase in abstraction over the Tk canvas. Because it is written entirely in Tcl/[incr Tcl], it is highly portable and should work with any other canvas extension.

The Slate is part of the Tycho user interface system [5], which can be obtained from the Tycho home page:

```
http://ptolemy.eecs.berkeley.edu/tycho/
```

Current development versions of the Slate can be obtained from:

```
http://ptolemy.eecs.berkeley.edu/
              ~johnr/code/slate
```

Acknowledgments

Key infrastructure without which this project would not have been possible has been developed by: Michael McLennan of Bell Labs ([incr Tcl]/[incr Tk]), John Ousterhout of Scriptics Corporation (Tcl/Tk), and Mark L. Ulferts of DSC Communications Corp ([incr Widgets]). We also thank the reviewers for their careful reading and comments on this paper.

Tycho is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, NEC, Philips, and Rockwell.

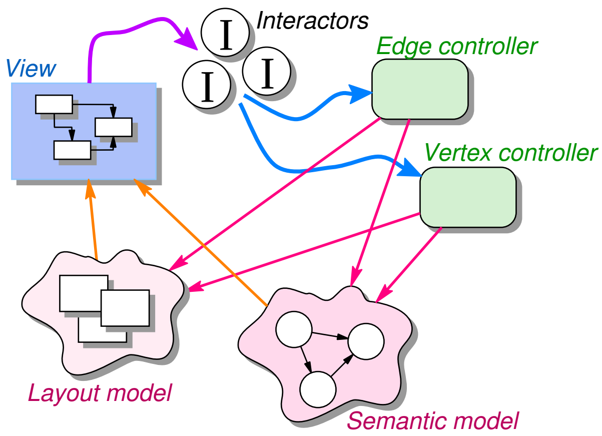

Figure 11: The architecture of a graphical editor

References

[1] Dave Collins. *Designing Object-Oriented User Interfaces*. Benjamin/Cummings, 1995.

[2] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wedley, 1996. Second Edition.

[3] Michael Gleicher. *A Differential Approach to Graphical Manipulation*. PhD thesis, Carnegie Mellon University, 1994. Also appears as CMU School of Computer Science Technical Report CMU-CS-94-217.

[4] Scott E. Hudson. Semantic snapping: A technique for semantic feedback at the lexical level. In *Proc 1990 SIGCHI Conference*, pages 65–70, April 1990.

[5] Christopher Hylands, Edward A. Lee, and H. John Reekie. The Tycho user interface system. In *The 5th Annual Tcl/Tk Workshop '97*, pages 149–157, 1997.

[6] Edward A. Lee and David G. Messerschmitt et al. An overview of the Ptolemy project. `http://ptolemy.eecs.berkeley.edu/papers/overview/`, March 1994.

[7] Brad. A Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990.

[8] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(1), November 1990.

[9] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doan. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

[10] H. John Reekie. *Realtime Signal Processing: Dataflow, Visual, and Functional Programming*. PhD thesis, School of Electrical Engineering, University of Technology, Sydney, Australia, September 1995.

[11] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.