Viewpoints Research Institute

Making Applications in KSWorld

Yoshiki Ohshima, Aran Lunzer, Bert Freudenberg
Ted Kaehler

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2013-003

Making Applications in KSWorld

Yoshiki Ohshima Aran Lunzer Bert Freudenberg Ted Kaehler
Viewpoints Research Institute
yoshiki@vpri.org, aran@acm.org, bert@freudenbergs.de, ted@vpri.org

Abstract

We report on our experiences in creating a GUI framework called KSWorld, which supports an interactive and declarative manner of application writing. The framework embodies direct manipulation, a high degree of loose coupling, and time-aware execution derived from Functional Reactive Programming (FRP). We also describe how a universal document editor was developed in this framework.

The fields, or slots, of graphical widgets in KSWorld are reactive variables. Definitions of such variables can be added or modified in a localized manner, allowing on-the-fly customization of the visual and behavioral aspects of widgets and entire applications. Thus the KSWorld environment supports highly exploratory application building: a user constructs the appearance interactively with direct manipulation, then attaches and refines reactive-variable definitions to achieve the desired overall behavior.

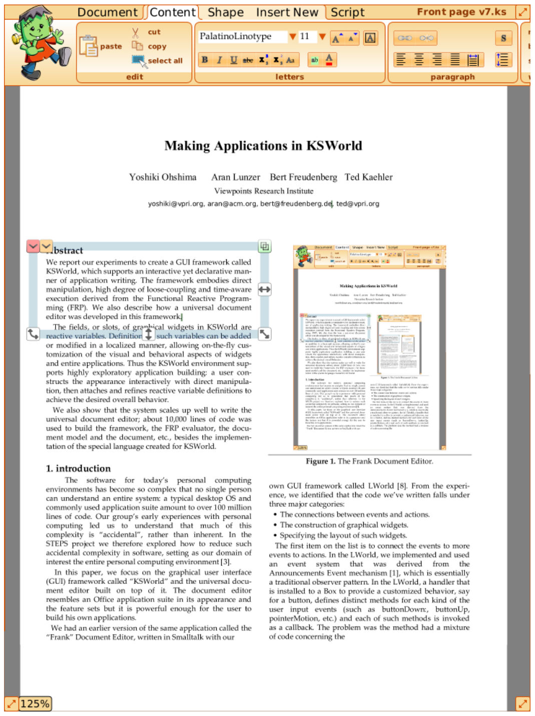
We also show that the system scales up sufficiently to support a universal document editor. About 10,000 lines of code were needed to build the framework, the FRP evaluator, the document model and the editor, including the implementation of the special language created for KSWorld.

1. Introduction

The software for today's personal computing environments has become so complex that no single person can understand an entire system: a typical desktop OS and commonly used application suite amount to over 100 million lines of code. Our group's early experiences with personal computing led us to understand that much of this complexity is "accidental", rather than inherent. In the STEPS project we therefore explored how to reduce such accidental complexity in software, setting as our domain of interest the entire personal computing environment [1].

In this paper, we focus on the graphical user interface (GUI) framework called KSWorld and the universal document editor built on top of it. The document editor resembles an Office application suite in its appearance and feature set, but is powerful enough for users to build their own applications.

The document editor is called Frank. We had written an earlier version of Frank in Smalltalk, using our own



Making Applications in KSWorld

Yoshiki Ohshima Aran Lunzer Bert Freudenberg Ted Kaehler
Viewpoints Research Institute
yoshiki@vpri.org, aran@acm.org, bert@freudenberg.de, ted@vpri.org

Abstract

We report our experiments to create a GUI framework called KSWorld, which supports an interactive yet declarative manner of application writing. The framework embodies direct manipulation, high degree of loose-coupling and time-aware execution derived from the Functional Reactive Programming (FRP). We also describe how a universal document editor was developed in this framework.

The fields, or slots, of graphical widgets in KSWorld are reactive variables. Definition such variables can be added or modified in a localized manner, allowing on-the-fly customization of the visual and behavioral aspects of widgets and entire applications. Thus the KSWorld environment supports highly exploratory application building; a user constructs the appearance interactively with direct manipulation, then attaches and refines reactive variable definitions to achieve the desired overall behavior.

We also show that the system scales up well to write the universal document editor; about 10,000 lines of code was used to build the framework, the FRP evaluator, the document model and the document, etc., besides the implementation of the special language created for KSWorld.

1. introduction

The software for today's personal computing environments has become so complex that no single person can understand an entire system: a typical desktop OS and commonly used application suite amount to over 100 million lines of code. Our group's early experiences with personal computing led us to understand that much of this complexity is "accidental", rather than inherent. In the STEPS project we therefore explored how to reduce such accidental complexity in software, setting as our domain of interest the entire personal computing environment [3].

In this paper, we focus on the graphical user interface (GUI) framework called "KSWorld" and the universal document editor built on top of it. The document editor resembles an Office application suite in its appearance and the feature sets but it is powerful enough for the user to build his own applications.

We had an earlier version of the same application called the "Frank" Document Editor, written in Smalltalk with our

Figure 1. The Frank Document Editor.

own GUI framework called LWorld [8]. From the experience, we identified that the code we've written falls under three major categories:

- The connections between events and actions.
- The construction of graphical widgets.
- Specifying the layout of such widgets.

The first item on the list is to connect the events to more events to actions. In the LWorld, we implemented and used an event system that was derived from the Announcements Event mechanism [1], which is essentially a traditional observer pattern. In the LWorld, a handler that is installed to a Box to provide a customized behavior, say for a button, defines distinct methods for each kind of the user input events (such as buttonDown:, buttonUp, pointerMotion, etc.) and each of such methods is invoked as a callback. The problem was the method had a mixture of code concerning the

Figure 1. The Frank Document Editor.

GUI framework called LWorld [2]. From that experience, we identified that the code falls into three major categories:

- The connections between events and actions.
- The construction of graphical widgets.
- Specifying the layout of such widgets.

The first category is code for connecting input events through to actions. In LWorld, we implemented and used an event system derived from the Announcements Event mechanism [3], which is essentially a traditional observer pattern. A handler in LWorld that is installed to a graphical element (`Box`) to provide customized behavior for, say, a button, defines distinct methods for each kind of user input event (`buttonDown`, `buttonUp`, `pointerMotion`, etc.), and each of these methods is invoked as a callback. The problem was that each method ended up as a mix of code relating to the button's behavior and to its visual appearance, while the code defining logical states of the button, such as `pressed` or `entered`, was scattered across the callback methods making it hard to discover how state transitions occur.

As described in an earlier report on KScript and KSWorld [4], we decided to employ Functional Reactive Programming (FRP) [5]. Among other issues, FRP provides a simple solution for the problem outlined above. In KSWorld the states of a button, such as `pressed`, are manifested as dataflow graph node specifications. A dataflow node can depend on multiple sources, and KScript allows the node to access its own previous value. With these features we can express concisely the definition of each state variable. We can also express, quite separately from the variable holding logical state, streams such as for changing the appearance of a button. The cleaner resulting code gave us roughly a 5 to 1 reduction in lines of code. Again, please refer to [4] for more details.

With the first item on our list under control, the next issue is how to construct the `Box`es. Traditionally this tends to involve writing code that instantiates `Box`es, then specifies their sizes, locations, colors and other properties. The following is an example (in Smalltalk, for the old LWorld) that creates a title bar for a window-like widget as shown in Figure 2:

```
newTitleRow: ext named: titleString for: owner
  | titleRow title nameWrap dismissButton |
  titleRow := LBox extent: ext color: LBox themeColor.

  title := LBox newLabel: titleString.
  nameWrap := LBox extent: (ext x //2 @ ext y)
                   color: Color transparent.
  nameWrap name: 'nameWrap'.
  nameWrap clipping: true.
  nameWrap add: title.
  title translation: 2@4.
  titleRow add: nameWrap.

  dismissButton := LBox withShape: LBox dismissIcon.
  dismissButton name: 'dismiss'.
  titleRow add: dismissButton.
  ^ titleRow.
```

Here, `#extent:color:` and `#withShape:` are primitive `Box` instantiation methods, while `newLabel:` is a convenience method to set up more `Box`es that represent the characters in a one-line text-field `Box`.

Conceptually, all we want to do is to make a handful of `Box`es, setting up their properties and bringing them together. However, if this is done by writing textual code, it requires dozens of repetitive lines as shown above. As pointed out by Bret Victor, trying to construct graphical entities by indirectly manipulating symbols is in any case an unpleasant way of interacting with a computer. It also makes it hard to collaborate with designers who are good at using conventional graphics tools but to whom code of this kind is alien.

For this problem, we take an approach that draws upon our experiences from Etoys [6]: namely, we try to make an environment where the user can interactively construct the widgets and change their properties. Also, as an optional feature, we make available an importer for graphics data that has been created with external tools. This may appear to be similar to modern GUI interface builders such as XCode, but there is a major difference: in KSWorld, the application being built is always alive; there is no need to go through re-compilation, or to press any kind of "run" button.

Implementing such interactive authoring features adds to the complexity of the entire system. Remember, however, that we would like to have a system where the user can do exploratory construction of graphical widgets and scripts. We can utilize this characteristic to build the authoring tool itself, thus leading to a smaller and cleaner system. Also note that this concept of a highly dynamic environment may sound at odds with the FRP model, which is sometimes equated with hardware design, but—as the end-user spreadsheet application shows—allowing dynamic change in the reactive programming model can work well too.

The last item on our list is to describe the layout of multiple `Box`es. We provide a simple constraint-based layout mechanism (called `KSSimpleLayout`), along with vertical and horizontal layouts, to specify the locations and sizes of `Box`es. For example, the above-mentioned newTitleRow:named:for: code in LWorld was actually intertwined with the following code:

```
titleRow layout: (LSimpleLayout new
 keep: #topLeft of: 'nameWrap' to: #topLeft
     offset: 4@0;
 keep: #topRight of: 'dismiss' to: #topRight
     offset: -2@0; yourself).
```

The calls to `#keep:of:to:offset:` set up the constraints among the `Box`es mentioned by name, and the solver thereafter maintains their relative locations and sizes as specified.

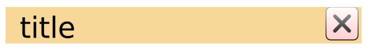


Figure 2. An example of a title bar.

Of course, one could imagine setting up a layout in a graphical manner, as one would do in Apple's XCode environment. However, taking a quick stock-check of lines of code used for constraints in LWorld we find just over 700. While this would be enough to make an interactive constraint editor in KSWorld, it is not clear that we would achieve any great saving in lines of code, or reduction in accidental complexity of the system. Therefore for now we regard the creation of such an interactive editor as a lower priority, and continue to write constraints using textual code.

The remainder of this document is organized as follows. In Section 2 we discuss the revised syntax of the KScript language. Then in Section 3 we walk through a simple example of building a widget in KSWorld, and in Section 4 illustrate the Shape Editor, which supports creation and editing of graphical shapes; this shows that suitably elaborate graphical elements can be made interactively with the tools available. In Section 5 we show that even the main command interface in the Document Editor is based on simple widgets that a user can build from scratch. Section 6 describes briefly how we made a generic text field based on the idea that laying out text is just a matter of writing a layout. All these elements are brought together in the Document Editor (as seen in Figure 1), described in Section 7. Finally, in Section 8 we switch point of view, examining how much code was written to implement each part of the system.

2. Revised KScript Language

Since our original presentation of the KScript language [4], we have made some purely syntactic changes to the language. Here we present KScript in its revised form.

2.1 Base language

The base object in KScript, called `KSObject`, is a simple dictionary that stores values under keys. A `KSObject` can understand a basic set of methods, and can inherit more methods from its parent.

The surface syntax resembles CoffeeScript [7]. In our search for a clean syntax we decided to try using the "off-side rule", in which level of indentation is used to define the nesting of code blocks.

CoffeeScript inherits some problems from its ancestor JavaScript, such as the fiddly distinction between arrow `->` and fat arrow `=>` in defining a function, indicating alternative ways to bind the `this` pseudo-variable. We simplified the language and eliminated such issues.

Unlike some languages that require a syntactic marker (such as `@` in Ruby) to distinguish temporary variables from objects' instance variables (fields), for KScript we wanted to favor a cleaner appearance. Both temporary variables and the fields of the receiver are referred to just by specifying a name. To distinguish the two, we require all temporary variables to be declared explicitly with `var`.

We use `:=` for the "common case" assignment operator (the special case is described below). Here is a simple piece of code:

```
aFunction := (a) ->
   var c := a + b
   return c
```

An anonymous function with a single argument called `a` is created by the `() ->` syntax and bound to the variable `aFunction`. In the body of the function, `b` is a reference to a field in `this` (the object in which this definition is executed), `c` is a temporary variable that is being assigned to, and the value of `c` is returned.

This syntax, where the temporary variables and fields are not distinguished syntactically, makes the compilation of code context dependent. That is, the meaning of a line of code can be different depending on the existence of local bindings. However, although it is possible to refer to a variable that is defined as an argument or temporary in a containing scope, in our experience the need for such "free" variables is rare: it is always possible to create a field in the receiver to hold the needed value. A further reason to avoid using free variables in definitions is that they cause problems in serialization and deserialization.

2.2 FRP-style dataflow extension

On top of the base language we added an FRP-style dataflow extension. As a dataflow definition is always stored into a field, it takes the following form:

```
fieldName <- expression
```

The left hand side of the special assignment operator `<-` is a field name, and the right hand side is a dataflow definition. When such a line is executed, the right hand side is evaluated to a kind of delayed object called a stream, and is assigned into the specified field of `this`. For example, the code snippet below uses a function called `timerE()` to create a stream that updates itself with a new value every 200 milliseconds, and this stream is assigned to a field called `myTimer`:

```
myTimer <- timerE(200)
```

Initially the stream created by timerE(200) has no value (strictly, it has `undefined` as its value), and each 200 "logical" milliseconds it acquires a new value corresponding to the logical time of the system.

The stream can be used by other streams:

```
fractionalPart <- myTimer % 1000
sound <- FMSound.pitch_dur_loudness(fractionalPart,
                                    0.2, 100)
player <- sound.play()
```

The operator `%` calculates the remainder, so the value in the `fractionalPart` stream is the milliseconds part of the `myTimer` stream (i.e., a sequence `[..., 0, 200, 400, 600, 800, 0, 200, ..., 800, 0, ...]`). This value is used by the `sound` stream to create an `FMSound` object with the specified pitch, 0.2 seconds duration, and 100 for loudness. The new value of the `sound` stream is in turn sent the `play()` message right away. The result is a stair-like tune.

The expression on the right of a `<-` assignment has a similar meaning to those quoted with `{!...!}` in Flapjax. When the compiler reads the expression, it treats the variable references as dependency sources (such as `myTimer` in `fractionalPart`, and `fractionalPart` in `sound`). This means that when a source changes, the expression will be evaluated to compute a new value for the stream.

An important point is that such variable references are loosely coupled. That is, the actual stream to be bound to the variable is looked up in the owning `KSObject` *each time* the referenced sources are checked for updates.

This scheme has some clear benefits. The order of the stream definitions in a chunk of code does not affect the program behavior (as in Compel, a single assignment language [8]); changing the dependency graph requires no extra bookkeeping effort; and the garbage collection works without needing to unregister dependents from their sources, or to detect finished streams.

There is a way to filter changes and stop them from propagating downstream, using the value `undefined`. In KScript's dataflow model, when the value computed for a stream is `undefined` the system treats it not as a new value for the stream but as a signal for not propagating changes further. For example, the stream `stopper` below does not update beyond 1,000, and the value in `timerViewer` stream does not exceed 10 (1,000 divided by 100):

```
stopper <- if myTimer > 1000 then undefined else myTimer
timerViewer <- stopper / 100
```

2.3 Behaviors and events

In FRP, there is a distinction between "behaviors", which represent continuous values over time, and "events", which represent sequences of discrete values.

Under the pull-based, or sampling-based evaluation scheme that KScript operates (explained in Section 2.5), a behavior can easily be converted to events and vice versa (a behavior is like a stream of events but the value of the last event is cached to be used as the current value; an event is like a behavior but each change in the current value is recorded as an event).

However, they still need to be treated differently, and mixing them in computation can cause semantic problems. Also, whether to reinstate the value of a stream upon deserializing is dictated by whether the stream is a behavior or not (we discuss this in more detail in [9]).

In KScript, a behavior is defined with an initial value and an expression that produces the values that follow. The initial value is given either with the keyword `fby` (meaning "followed by", and borrowed from Lucid), or the function `startsWith()` (borrowed from Flapjax). For example, a behavior that represents a Point starting from (0, 0) and moving to the right over time can be written as:

```
aPoint <- P(0, 0) fby P(timerE(100) / 10, 0)
```

A stream that has no stream references in its definition is called a value stream. To create a value stream that acts as a behavior, the function `streamOf()` is used. It takes one argument and creates a constant stream with that argument as the value. To create a value stream that acts as an event, the function `eventStream()` is used.

2.4 Combinators

In addition to the basic expressions used in the examples above, KScript offers several *combinators* that combine other streams to make a sub-graph in a dependency network. The combinators' names and functionality are drawn from FRP implementations, especially Flapjax.

2.4.1 Expressions and "when" constructs

As described above, when a stream reference appears in the definition of another stream, the compiler marks it as a source. Below, `color` is a source for the stream bound to `fillUpdater`:

```
fillUpdater <- this.fill(color)
```

When the dependency specification is more complex, or it would be convenient to bind the value of the trigger to a temporary variable, one can use the `when-then` form to specify the trigger and response:

```
fillUpdater <- when
                 Color.gray(timerE(100) % 100 / 100) :c
               then
                 this.fill(c)
```

The `timerE` triggers the `gray` method of `Color`. The resulting color value is bound to a temporary variable `c` and used in the `then` clause, which will be evaluated and becomes the new value of the stream. As is the case here, it is sometimes true that the side effects caused by the `then` clause are more interesting than the actual value.

Internally, the `when` form is syntactic sugar for the more traditional combinator `mapE`, and an argument-less variation of it called `doE`. The following two lines are equivalent:

```
beeper <- when mouseDown then this.beep()
beeper <- mouseDown.doE(() -> this.beep())
```

2.4.2 `mergeE`

The `mergeE` combinator takes multiple stream expressions as its arguments, and updates itself whenever the value of any of those expressions changes.

The value of the `mergeE` is the value of the expression that most recently changed. However, again it is sometimes the case that the actual value of `mergeE` is not used in the triggered computation; what is important is just the fact that something is to be updated. For example, imagine you have a line segment object (called a Connector) in an interactive sketch application, and it has to update its graphical appearance in response to movement of either of its end points (bound to `start` and `end`), or to a change in the width or fill of its line style. We watch for any of these changes with a single `mergeE`, then invoke a method with side-effects (`updateConnector()`) to recompute the graphical appearance:

```
updateLine <-
    when
      mergeE(
        start.transformation,
        end.transformation,
        fill,
        width)
    then
     this.updateConnector()
```

2.4.3 `anyE`

In GUI programming, there is often a need to watch a collection of homogeneous objects and detect when any of those objects changes. For example, a menu can be defined as a collection of buttons, that reacts when the `fire` stream of any of the buttons is updated due to a click from the user. The `anyE` combinator takes as arguments a collection of objects and the name of the stream to watch. For example:

```
Evaluator.addStreamsFrom = (anObject) ->
  for stream in anObject
    // add stream to the list of streams

Evaluator.sortAndEvaluateAt = (logicalTime) ->
  var sorted = this.topologicallySortedStreams()
  for stream in sorted
      stream.updateIfNecessary(logicalTime)
```

Figure 3. The evaluation method of KSObject in pseudo-code

```
items  := col      // a collection of buttons
fire   <- anyE(items, "fire")
```

The `items` field is holding the button collection. The `anyE` stream looks for a new value in the `fire` stream of any item, and updates itself with that value.

2.4.4 `timerE`

This was already used in Section 2.2. It takes a numeric argument (in fact it could be a stream expression, but we have not yet found a use case for this) and creates a stream that updates itself after each passing of the specified number of milliseconds.

2.4.5 `delayE`

`delayE` delays the propagation of events for a specified length of time. The syntax of `delayE` looks like a message send. It takes a numeric argument, and delays upstream events by the specified number of milliseconds before propagating them. For example, compare these two stream definitions:

```
beeper <- buttonDown.doE(() -> this.beep())
beeper <- buttonDown.delayE(500).doE(() -> this.beep())
```

In effect, the first definition of `beeper` creates a pipeline that has two nodes (`buttonDown` and `doE`), and that makes a beep noise when the mouse button is pressed. The second definition has `delayE(500)` inserted into the pipeline; this causes each event from `buttonDown` to be delayed for 500 milliseconds before triggering the `doE`.

2.5 Evaluation scheme

The basic strategy of the evaluation scheme in KScript can be considered a pull-based implementation of FRP with all streams being looked at. The evaluation cycle is tied to the display update cycle; at each cycle, the streams involved in the system are sorted into their dependency order and evaluated if necessary.

As described in Section 2.2, a stream holds the names of its sources. These symbolic references are resolved at the beginning of each evaluation cycle, and the dependency information is used to topologically sort the stream into a list. Each stream in the list is then checked to see if any of its sources has been updated since the last cycle. If so, the expression for the stream is evaluated immediately and the value updated, possibly affecting streams later in the list.

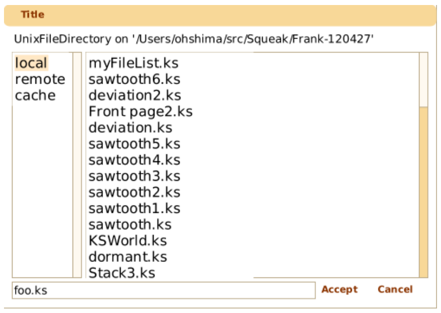


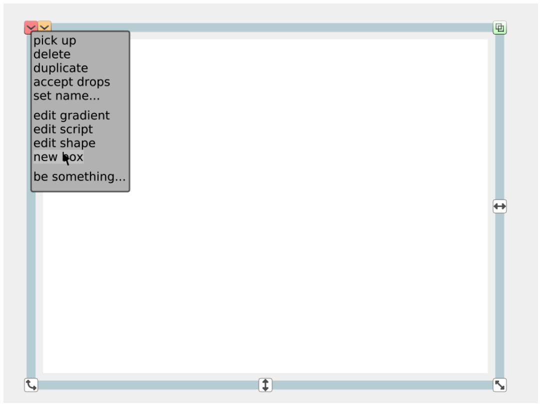
Figure 4. The File List.

3. Example: The File List

For interacting with files, we would like to have a standard dialog to show the files that are available and allow the user to choose one. In this section we illustrate how we can make a File List, shown in Figure 4, from a set of rudimentary tools. It needs to show a list of directories, a list of files in the selected directory, and the full path and name of the selected file. Pressing the Accept button will make the selected file's details available to client code, while pressing the Cancel button will just close the File List.

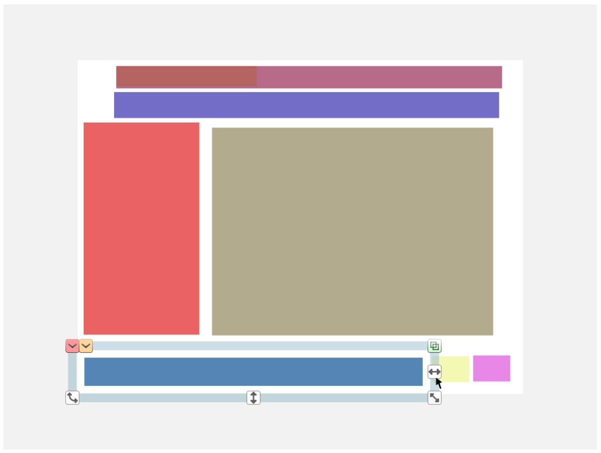
The steps in making a tool in KSWorld are as follows: 1) Make a compound widget. 2) Edit the properties and styles with the Inspector and the Shape Editor, if necessary. 3) Write code to specify the layout, if necessary. 4) Write code to connect the events and actions. This can be done either in the Inspector or in the code editor of the hosting environment. And, 5) Write code to set up the widget with its layout and behavior.

We start from an empty `Box`, then use its halo menu to add child `Box`es within it:



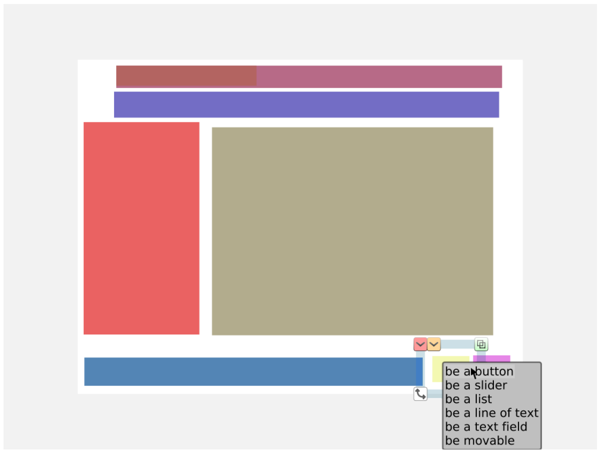
Making a new child `Box` from the halo menu.

In all, we need eight such `Box`es. We use the halo to resize and roughly place each one to get a feel for the eventual layout. Note that KSWorld initializes each new `Box` with a random color, which helps ensure that they are visually distinct at this stage:



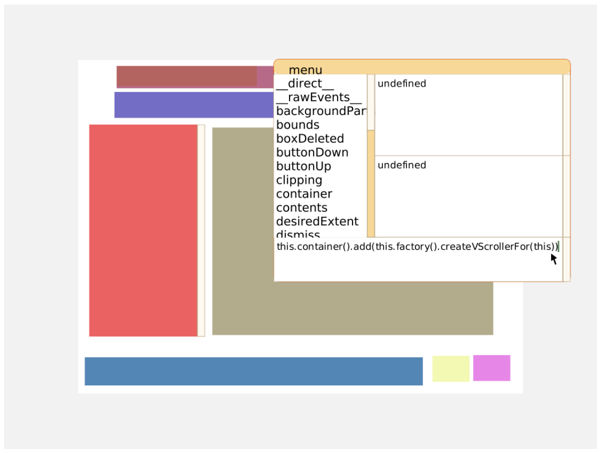
A rough sketch of `Box`es for the File List.

All these `Boxes` have only the most basic behaviors, so the next step is to assign appropriate additional standard behaviors as needed. For example, using the "be something" halo sub-menu we can give a `Box` the behavior of a button.



Attach behaviors to some `Box`es.

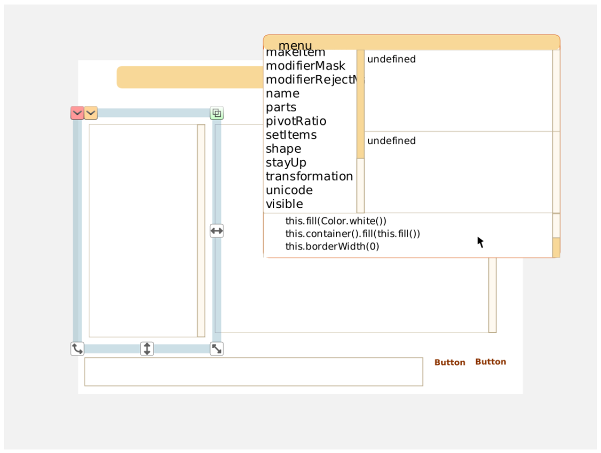
At this point we realize that the directory list and file list are likely to be taller than the tool, so their respective fields need to be scrollable. For edits and actions beyond those available through the halo menu we can use the rudimentary Inspector tool, that lets us inspect the slot values of an object and execute KScript expressions in the object's context. Here we evaluate a line of code to turn a list into a vertical scroller:



```
this.container().add(this.factory().createVScrollerFor(this))
```

Make each list `Box` scrollable.

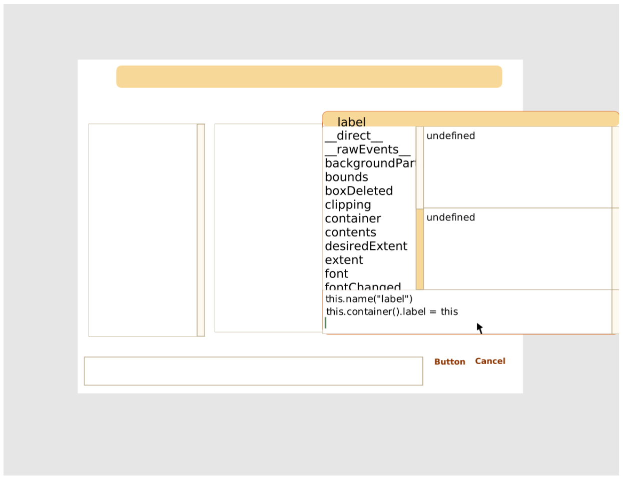
The Inspector can also be used to set all the `Boxes`' fills and border styles as desired:



```
this.fill(Color.white())
this.container().fill(this.fill())
this.borderWidth(0)
```

Set fills and border widths.

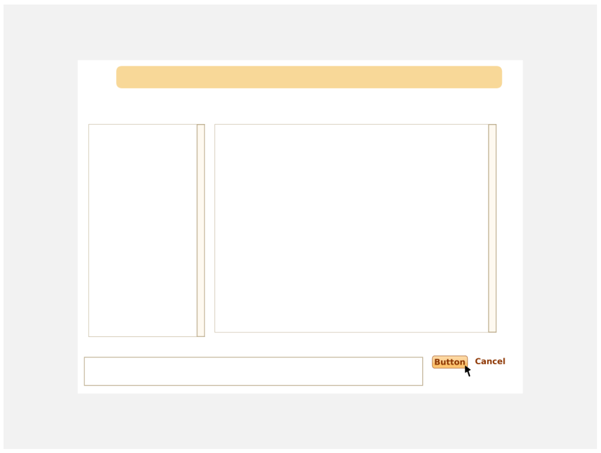
In the same manner we assign each `Box` into a slot in the overall widget so it can be referred to in the KScript dataflow defining the File List's behavior (shown in Appendix B), and give it a name by which it will be referenced for the layout:



```
this.name("label")
this.container().label = this
```

Set names for components.

Note that these Boxes are live and already reacting to user input. Buttons highlight when the pointer enters them.



A new button, reacting to the pointer.

The code for the layout of the File List is written in an external text editor. It is about 25 lines of constraints specifying the relationships among 8 widgets; it appears in its entirety in Appendix A.

There is a small piece of code to set up the File List. It will install the layout, modify the label of the Accept button as supplied by the client, and set up client-supplied defaults for the file-name wildcard patterns and the browsing start-points referred to as shortcuts:

```
setup := (title, acceptLabel, fileName, patterns,
         extent, theShortcuts) ->
  acceptButton.textContents(acceptLabel)

  this.layout(this.fileListLayout())

  patterns <- streamOf(patterns.findTokens(","))
  shortcuts <- streamOf(theShortcuts)
  this.behavior(fileName)

  return this
```

The third line installs the layout into the Box. As we write and adjust the code for the layout, we could execute this line on its own to check the overall appearance of the composite.

The File List also needs a definition of the behavior method that is called from setup, specifying the actions that should be performed in response to relevant events such as choosing (clicking) in the lists. The full listing of the behavior method is given in Appendix B. One highlight is this stream definition:

```
fire <- when
          acceptButton.fire
        then
          { dir: selectedShortcut,
           file: nameField.textContents()}
```

where selectedShortcut is the currently selected shortcut and nameField is a Box that is showing the currently selected file name. This definition specifies that when the acceptButton's fire stream is updated, the fire stream of the File List itself will acquire a new value that is an object with two fields. The client of the File List sets up its own stream that watches this fire stream to trigger a response to the chosen file.

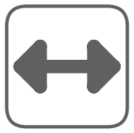
Because of the loose coupling of stream names, the File List does not need to contain any knowledge of the client (or potentially multiple clients); its sole job is to set a new value into the fire stream. Thus developing the File List and its clients can be done independently.

In total, about 25 lines of layout specification, 40 lines of stream definitions and 10 lines of setup code was enough to implement a usable File List. This compares to roughly 250 lines of Smalltalk code used to implement a comparable File List for LWorld.

4. Example: Making an Icon

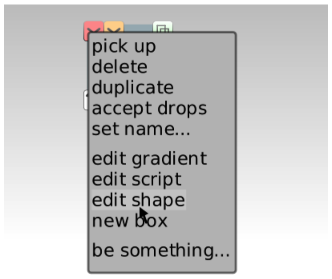
To build up from a bare-bones system to an end-user oriented application, we need a way to create more visually pleasing graphics. Again we would like to do this directly, rather than by manipulating symbols in textual code. We have therefore built a vector-graphics Shape Editor sufficient for simple graphics (for more elaborate compositions, such as the Frank cartoon character, we provide an importer for reading SVG files built outside the system).

Let's see if we can build the icon used in the halo to resize the target Box horizontally.



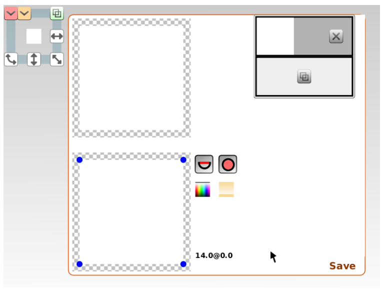
The Resize icon.

We start with a small white square Box, and invoke the Shape Editor from the halo menu.



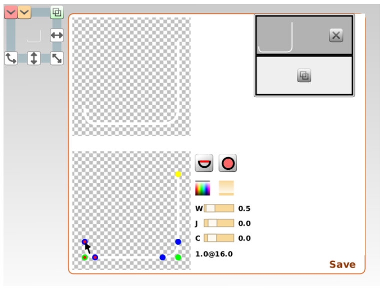
Invoking the Shape Editor.

This opens up an editor for the Shape of this Box. A Shape comprises multiple paths, that can be considered as layers. In the editor, the layers appear in a scrollable list on the right, the currently selected layer in an edit view at bottom left, and the composite view of all layers at top left. In the middle are controls for toggling whether the path is open or closed, and whether filled or unfilled (stroked). There are also buttons for launching pickers for a solid fill color or for a gradient fill. When editing a stroked path, sliders appear for setting its width, and its join and cap shapes. The individual segments within a path (each a quadratic Bézier curve) are shown using colored manipulation handles: blue for segment end points, green for control points. The user edits segments by dragging these handles, with movements that are usually constrained to a grid of integer coordinates. When editing an open path, segments can be added and deleted.



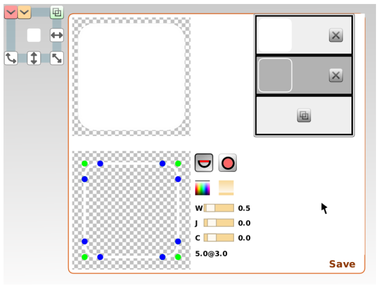
Initial shape, with a single filled path.

Our first task is to round the Box's corners. We change the path from filled to stroked, then break it open and move the end points to add corner segments.



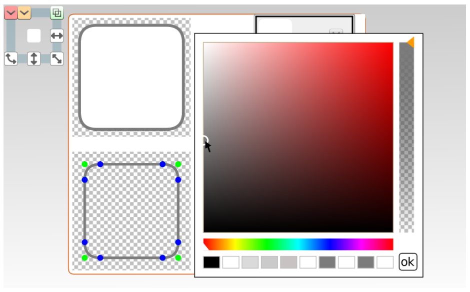
Rebuild the path with rounded corners.

Once the path is complete we close it again, then duplicate the current layer and fill the path in one of the two resulting layers.



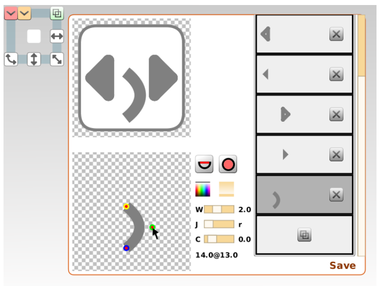
Duplicate the layer, and fill one copy.

The stroked layer is to become the border, so we bring up a color picker to give it a solid fill.



Set a gray fill for the border.

We repeat the addition and editing of layers to construct the parts of the icon, as seen in the growing list on the right of the editor. While manipulating the final segment to form the line through the middle, we might find that it looks like an elephant moving its trunk:

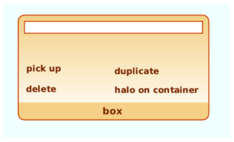


Move the final path into place.

Recovering from this distraction, we move the path segment into its proper position and press the save button to store the completed composition as the new Shape for the original white Box. We can then save the Box to a file for later use.

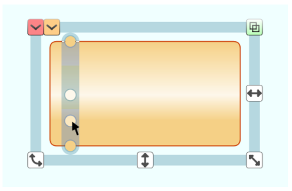
5. Example: A Panel for the Tool Bar

We now demonstrate how we make a panel, also known as a bubble, containing commands for the Document Editor.



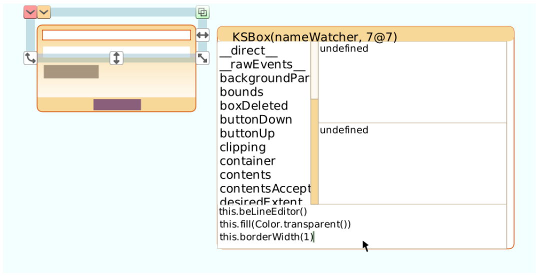
A box-editing bubble, as it appears when no box is selected.

The first step is to create a Box to be the bubble, give it rounded corners and a border just as for the icon in the previous section, and an appropriate gradient fill. In general the Shape Editor can be used to create a fill for each layer of a shape, but in this case the entire shape only needs a single linear gradient, so it can be added using the gradient tool invoked directly from the halo:



Using the halo's gradient tool.

Then, as seen before, we can add a number of Boxes to become the bubble's buttons, labels and so forth. In this example we are building a bubble that supports manipulation of whichever Box within the document the user has highlighted with the halo. This bubble needs an editable text field to hold the name of the selected Box. We first customize a Box to turn it into a one-line text editor:



```
KSBox(nameWatcher, 7@7)
__direct__          undefined
__rawEvents__
backgroundPar
bounds
boxDeleted
buttonDown          undefined
buttonUp
clipping
container
contents
contentsAccept
desiredExtent
this.beLineEditor()
this.fill(Color.transparent())
this.borderWidth(1)
```

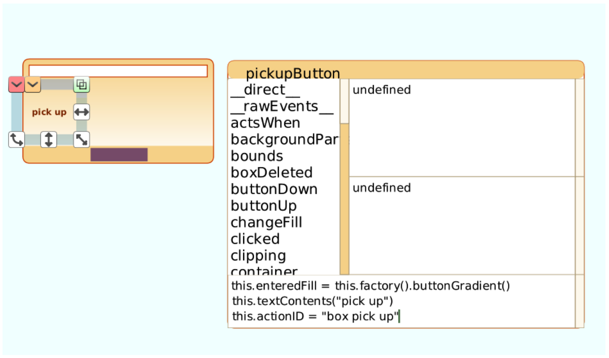
Customizing a part within the bubble.

Then we add the following stream to make the text field update according to the selected Box's name:

```
selectionWatcher <-
  when
    DocEditor.selectedDocBox :b
  then
    this.textContents(if b then b.printString() else "")
```

where the virtual field DocEditor always refers to the Document Editor handler, so DocEditor.selectedDocBox refers to the selected Box, and the result is converted to a string and shown in this Box (see Section 7.2 for more explanation of selectedDocBox).

The bubble contains buttons to trigger editing commands on the selected Box. Each button is to change its fill when the pointer rolls over it, to provide feedback. In this case we have pre-built a gradient fill and made it accessible through a convenience method, so we can just run a short script to set up this enteredFill along with the button's label and action identifier:



```
pickupButton
__direct__          undefined
__rawEvents__
actsWhen
backgroundPar
bounds
boxDeleted
buttonDown          undefined
buttonUp
changeFill
clicked
clipping
container
this.enteredFill = this.factory().buttonGradient()
this.textContents("pick up")
this.actionID = "box pick up"
```

Setting up one of the bubble's command buttons.

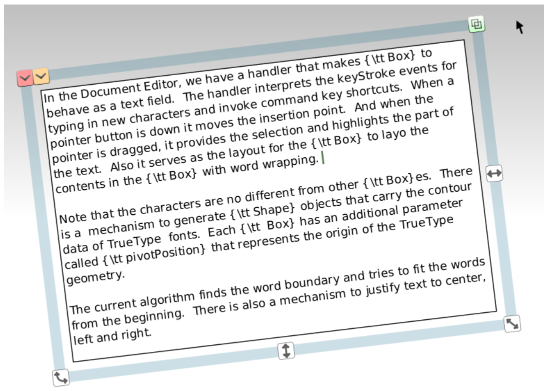
Each of the buttons has a stream called fire, which acquires a new value when the button is clicked. The bubble consolidates the fire streams of its buttons into a single fire stream of its own, using the following stream definition:

```
fire <- anyE(contents, "fire")
```

The entire tool bar of the Document Editor, in turn, consolidates the fire streams of its constituent bubbles. This form of implementation allows largely independent development of the bubbles' clients, the bubbles themselves, and even of the tool bar. The developer of the client can make progress without the tool bar being available yet, knowing that the client code will just need to watch the fire stream of an object that will be looked up through a named field. The internal structure of the tool bar is also hidden from the client, so the developer of the bubbles is free to explore alternative organizations of commands.

To finish this bubble we create the other command buttons as duplicates of the first, giving them appropriate locations, labels and action identifiers. The finished bubble is stored in a file directory for later use.

6. Text Fields



In the Document Editor, we have a handler that makes {\tt Box} to behave as a text field. The handler interprets the keyStroke events for typing in new characters and invoke command key shortcuts. When a pointer button is down it moves the insertion point. And when the pointer is dragged, it provides the selection and highlights the part of the text. Also it serves as the layout for the {\tt Box} to layo the contents in the {\tt Box} with word wrapping.

Note that the characters are no different from other {\tt Box}es. There is a mechanism to generate {\tt Shape} objects that carry the contour data of TrueType fonts. Each {\tt Box} has an additional parameter called {\tt pivotPosition} that represents the origin of the TrueType geometry.

The current algorithm finds the word boundary and tries to fit the words from the beginning. There is also a mechanism to justify text to center, left and right.

Figure 5. A Text Field. Each letter is a Box with vector graphics data representing the contour of a glyph.

In KSWorld, we have a handler that makes a Box behave as a text field. The handler interprets keyStroke events for typing in new characters and invoking command key shortcuts. A pointer click is interpreted as setting the position of the insertion point, and dragging the pointer defines a selection, highlighting part of the text. The handler also serves as the layout object for the Box, arranging its contents with suitable wrapping at word boundaries.

Note that each character in text is represented by a Box, like any other in a KSWorld composition. The Shape for each such Box is generated from the contour data for the relevant character in the chosen TrueType font. Each Box has an additional parameter called pivotPosition that represents the origin for the TrueType geometry.

Through iterative design we have arrived at a set of concise declarative rules that together specify left-to-right word layout with wrapping. The current implementation of a text field's layout object is a transcription of these rules.

There is also a mechanism to support more elaborate text layout, such as justifying text at the left and/or right, and centering it. The heights of the Boxes making up a line in a text field can vary; the layout finds the tallest Box in each line in order to set the positions of all Boxes in the line.

7. Putting It All Together: the Document Editor

In this section we show how a Document Editor resembling a productivity-suite application can be created out of the KSWorld Boxes presented up to now. One important observation is that the editor itself does not have to have much functionality, because in our design each Box that would become part of a document already embodies features for being customized not only in terms of appearance but also with actions and behaviors. A large part of the Document Editor's job is simply to provide a convenient user interface to these features.

The overall design of the Document Editor borrows from Microsoft Office's ribbon interface [10]. Each command bubble, as described above, contains a set of commands that are closely related. When a Box in the editing area is highlighted with the halo, the tool bar will show only bubbles that are relevant to that Box (or to the document as a whole). There are too many bubbles for all of them to be seen at once, so we group them into tabs such that the most frequently used bubbles appear by default, and we let the user access the rest by selecting other tabs. Managing this tool bar structure is one of the Document Editor's responsibilities.

The Document Editor also provides the UI for navigating to a document and to a page within it, starting from a set of directory shortcuts and ending with a list of thumbnails for the document's pages. We call this interface the directory browser (see Section 7.3).

Buttons within the Document Editor allow the user to hide the tool bar and directory browser selectively, for example to give priority to the document itself when giving a presentation. The document can also be zoomed to a range of viewing scales.

Finally, the Tile Scripting area (see Section 7.4) supports "presentation builds" for each page of a document, in which the visibility of individual Boxes on the page can be controlled through a tile-based scripting language.

7.1 The Document Model

While the basic model of a document is simply homogeneous Boxes embedded into each other, we wanted to have a higher-level structure allowing end-users to organize document contents.

From our past experiments, we adopted a HyperCard-like model of multiple cards (or pages) gathered into a stack. Conceptually, a KSWorld stack is an ordered collection of Boxes that each represent one page. Additional properties control which child Boxes are specific to a single page, and which are shared among many (e.g., to act as a background or template). When the user turns or jumps to a different page, any changes made to the current page are stored into the data structure before the new page's data are brought in and displayed.

The model's combination of uniform object embedding and pages in a stack covers a variety of document types. A slide in a presentation maps naturally to a page, while a lengthy body of text can either appear in a scrolling field on one page or be split automatically across many.

7.2 Bubble selection

The current target of the halo is held in a stream called haloTarget belonging to the top-level Box in a KSWorld application. To customize the editor interface depending on the highlighted Box, the Document Editor needs a stream that depends on haloTarget of the top-level Window Box, which is accessible via the TopContainer virtual field. One could start to define the reaction logic as follows:

```
bubbleWatcher <-
  when
    mergeE(TopContainer.haloTarget,
           textSelection, whole.extent)
  then
    this.checkBubbleVisibility()
```

where checkBubbleVisibility() decides the set of bubbles to be shown, based not only on the halo highlight but also the existence of a text selection, and the size of the Document Editor as a whole (which determines how many bubbles will fit on the tool bar).

However, remember that the Document Editor interface itself is made up of Boxes, that a user might want to examine or customize. It would be bad if attempting to put the halo on a Box within a bubble, for example, caused that bubble itself to be categorized as irrelevant and removed from the display. This is a case for filtering the haloTarget stream by inserting the value undefined to suppress unwanted responses. We define a stream that checks whether the halo target is within the document or not:

```
selectedDocBox <-
  when
    TopContainer.haloTarget :box
  then
    if box && this.boxBelongsToDoc(box) || box == nil
      box
    else
      undefined
```

This stream updates itself to undefined when the highlighted Box is not part of the document (note that nil is also a valid value for haloTarget, meaning that no Box is high-

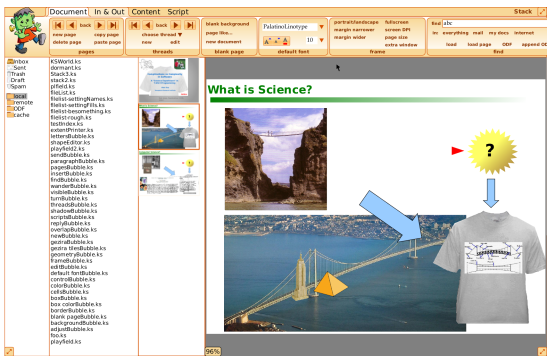


Figure 6. The Directory Browser on the left.

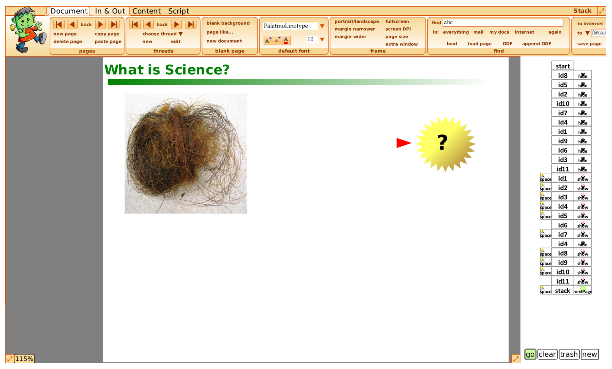


Figure 7. The Tile Scripting area on the right.

lighted). If the `bubbleWatcher` uses this filtered stream in place of `haloTarget`, it will only respond to halo placement within the document:

```
bubbleWatcher <-
  when
    mergeE(selectedDocBox,
           textSelection, whole.extent)
  then
    this.checkBubbleVisibility()
```

7.3 Directory Browser

On the left side of the Document Editor are three lists supporting navigation among documents and the pages within a document. From left to right, the lists hold a pre-defined set of “short cuts” to local or remote directories, a list of documents in the currently selected directory, and a list of thumbnails for the pages in the selected document.

These lists can be hidden selectively to open up more screen space for the document. Taking advantage of the highly dynamic nature of `Box` compositions, of which the Document Editor as a whole is one instance, this hiding and showing is achieved simply by replacing the layout object that arranges the sub-components of the interface.

7.4 Tile Scripting

In the retractable pane on the right side of the Document Editor is a simple tile-based scripting system that is designed to control the “presentation build” of a document page, for example in which some of the page’s `Boxes` are hidden to start with then progressively revealed as the keyboard space bar is pressed.

Figure 7 shows a page with document `Boxes` named `id1`, `id2`, etc. When the page is loaded the sequence of tiles will be executed from the top, so the objects with a `hide` tile attached will initially be hidden. The script then waits at the first line that has a `space` trigger attached. When the user hits the space bar, this trigger is satisfied and the tiles down to the next trigger will be executed.

The scripting area has its own interpreter, which simply visits the `Box` structure of the script and installs a keystroke or button-down event stream on each trigger `Box` it finds.

As well as allowing such scripts to be edited manually, we support building them programatically. For example, Frank’s ODF importer converts the visual effects specifications in an ODP file into KSWorld scripting tiles.

8. Line Counts

One of the goals of the STEPS project is to reduce the accidental complexity of software systems. The number of lines of code needed to write a system is one way to get a feel for such complexity.

As demonstrated above, KSWorld is already more than a single-purpose, minimal GUI framework: it supports direct-manipulation construction and authoring of new user documents and applications, and saving and loading documents.

Table 1 shows a breakdown of the lines of code in the system at the time of writing this report. The parts of the system summarized in the first subtotal (10,055) are considered to be those essential for implementing the Document Editor. The next entry, “Gezira Bindings”, is semi-essential. The remaining parts are not essential, but help generally with application development and optimization.

Below we briefly discuss each of the table entries. Before doing so, we should point out that KSWorld is currently hosted in the Squeak Smalltalk development environment. While most of KSWorld’s features are written in KScript, some optional or lower-level features are for the time being written in Smalltalk.

Also note that KScript itself can be considered a hybrid of two languages: a JavaScript-like basic object-oriented language, and a dataflow extension. From our experience, the number of lines of code required to implement in KScript a feature that does not make use of dataflow is comparable to implementing it in Smalltalk. Dataflow-based features are considerably more compact.

LOC	Total	description
753		KScript Compiler
291		Basic Object Model
654		FRP implementation
2,133		Basic Model of Box
548		Box Support
962		Text Support for Box
760		Common Handlers for Box
716		Layout
209		Stack
1,769		Document Editor
1,260		Serialization
	10,055	Sub Total
2,330		Gezira Bindings
	2,330	Subtotal of above.
288		OpenGL Rendering
95		Spreadsheet Table
492		SVG Importing
1,140		ODF Importing
1,110		Development Support
1,848		Tests
	4,973	Subtotal of above.
	17,358	Total

Table 1. The lines of code in the KSWorld and the Document Editor.

8.1 KScript compiler (753 lines)

The compiler reads KScript code and translates it to Squeak Smalltalk. The basic parts are a parser that generates a parse tree and a backend that generates Smalltalk from this tree. It is similar to our past experiments on a one-pass JavaScript compiler [11], but the separation of parser and back end resulted in a somewhat larger OMeta2/Squeak description, of around 450 lines.

In addition to the basic translation scheme, we need a translator to expand dataflow expressions into stream definitions. This expander is about 170 lines.

8.2 The basic object model (291 lines)

The basic object model is implemented on top of Squeak's dictionary class. The lines in this category are for additional behaviors such as equality tests and customized field access.

Note that this entry does not include the implementation of the execution engine, primitive objects or primitive data types. KScript uses numbers, strings, sequenceable collections and keyed collections from the hosting language, and relies on Squeak's execution engine. However, we made efforts to keep such dependencies to a minimum; it should be possible to port KScript to any common object-oriented language without needing large amounts of extra code.

8.3 The FRP implementation (654 lines)

The FRP implementation consists of a dependency sorter that does a topological sort of dependencies, and the (non-optimized, somewhat repetitive) implementation of around a dozen combinators.

8.4 The Box model (2,133 lines)

In addition to the very basic code needed to manage and draw the `Box` display tree, this count includes the code for direct manipulation features such as resizing a `Box`'s `Shape`, and the specialized behaviors of the top-level `Box` and of a "hand" `Box` that implements pointer-related facilities.

8.5 Box support (548 lines)

For the `Box` model to be functional and usable, it has to have access to the frame buffer for displaying graphics, and to the incoming raw user events. Currently these are provided by the Squeak environment in which KSWorld is hosted, through a custom Morph called `KSMorph`. We also include at this level the implementation of a `WorldState` object that manages the evaluation of the KSWorld.

8.6 Common handlers for boxes (760 lines)

A typical application requires common widgets, realized as `Boxes` with particular behaviors. This includes buttons, menus (which we implement as lists of buttons), scroll bars, connectors, and more elaborate widgets such as the color picker and gradient editor. We believe the dataflow approach has helped to keep this code compact: recall that a button is about 50 lines of code, as described in [4].

8.7 Layout (716 lines)

There are currently three kinds of layout, the simplest being `VerticalLayout` and `HorizontalLayout`, which arrange `Boxes` in a container vertically or horizontally. These layouts support flags to configure the spacing between `Boxes`, and how to behave when the `Boxes` overflow the available size.

The third kind of layout object is `KSSimpleLayout`, which was briefly mentioned in the Introduction. The programmer can specify `Boxes`' relative locations and extents in terms of constraints between them, and a solver ensures that these constraints are satisfied.

8.8 Text support (962 lines)

As discussed in Section 6, KSWorld text fields are implemented as flows of `Boxes` holding one letter each (this is a design decision inherited from LWorld and its predecessors such as Lessphic). The layout object for a text field therefore performs a job similar to that of `HorizontalLayout`, except that it must take into account word boundaries as opportunities to wrap the layout onto successive lines. The line count for this item includes the text handler's support for keyboard input and various text-editing command shortcuts.

Each character `Box` has a `Shape` that is generated from the specified TrueType font's contour data for the character, taking into account settings for font size, emphasis and fill color.

8.9 Stack (209 lines)

As discussed in Section 7.1, the Frank Document Editor supports a "stack" multi-page document model inspired by HyperCard, though unlike HyperCard we allow unlimited embedding of `Boxes`.

8.10 Document Editor and associated controls (1,769 lines)

The biggest application in the current system is the Document Editor itself. It has various UI elements such as the selection-sensitive tool bar and its component bubbles, and controls for controlling the document display area and zoom ratio.

This category also includes the code for specialized controls such as the Shape Editor, and the Tile Scripting support for presentation builds (see Section 7.4).

8.11 Gezira bindings (2,330 lines)

The rendering of graphics to a virtual frame buffer is done by the Gezira graphics engine. To use this engine (which amounts to about 450 lines of Nile code), we have a set of classes that represent available fill styles and stroke types, and one class to represent a path: computing its bounds, supporting hit detection, etc. These bindings are written in Smalltalk.

8.12 Texture composition by OpenGL (288 lines)

Optionally, we can utilize a substantial speedup of graphics rendering by using OpenGL to compose the textures that Gezira generates. The line count for this category does not include the OpenGL binding code that Squeak provides, nor of course the OpenGL code itself.

8.13 Spreadsheet table (95 lines)

As the basic dataflow formalism supported by FRP is very close to the calculation model for a simple spreadsheet, making a spreadsheet widget was straightforward. Each cell is logically represented as a stream slot within a KSObject for the whole table, and for each stream we create a `Box` to display the appropriate value string.

8.14 ODF and SVG readers (1,632 lines)

We have fairly complete ODF and SVG readers, which read in data in XML format and generate a corresponding `Box` structure. Both can be considered non-essential convenience features.

8.15 Serialization and deserialization (1,260 lines)

This category includes code for generating and parsing S-expressions, and a simple compressor/decompressor for text data.

8.16 Development tools in Morphic (1,110 lines)

This category represents an intermediate stage in the bootstrapping process towards a KSWorld independent of its Squeak hosting environment. In Squeak Smalltalk we wrote various tools for inspecting, editing and adding code to KScript objects, specialized to the stream-based behavior of these objects.

8.17 Tests (1,848 lines)

We have generated many test cases to exercise the compiler, the behavior of KScript FRP streams, and the facilities of KSWorld. While there is no honor in stinting on test cases, the line count here could certainly be made lower by paying more attention to redundancy and repetition.

Acknowledgments

We would like to thank Arjun Guha for fruitful discussion on the design; Alan Borning for giving us insights from Kaleidoscope and other work; Dan Amelang for design discussions and for creating the Gezira graphics engine; Ian Piumarta for some key designs in the Lessphic and Quiche work; Hesam Samimi and Alan Borning for testing the KScript environment; Takashi Yamamiya for the implementation of the `KSObject` inspector and early design discussions; Alex Warth for the language-building ideas and creating OMeta; and Alan Kay for the ideas of loose coupling and time-based execution. Also, we would like to thank our late friend Andreas Raab for long-lasting ideas on building frameworks.

References

[1] Alan Kay, Dan Ingalls, Yoshiki Ohshima, Ian Piumarta, and Andreas Raab. Steps Toward the Reinvention of Programming. Technical report, Viewpoints Research Institute, 2006. Proposal to NSF; Granted on August 31st 2006.

[2] Viewpoints Research Insitute. STEPS Toward Expressive Programming Systems, 2010 Progress Report. Technical report, Viewpoints Research Institute, 2010. Submitted to the National Science Foundation (NSF) October 2010.

[3] Vassili Bykov and Cincom Smalltalk. Announcements Framework. A series of blog entries at: `http://www.cincomsmalltalk.com/userblogs/vbykov`.

[4] Yoshiki Ohshima, Bert Freudenberg, Aran Lunzer, and Ted Kaehler. A Report on KScript and KSWorld. Technical report, Viewpoints Research Institute, 2012. VPRI Research Note RN-2012-001.

[5] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[6] The Squeakland Foundation. Etoys. `http://squeakland.org`.

[7] Jeremy Ashkenas. Coffeescript. `coffeescript.org`.

[8] Larry G. Tesler and Horace J. Enea. A language design for concurrent processes. In *Proceedings of the April 30–May 2,*

1968, spring joint computer conference, AFIPS '68 (Spring), pages 403–408, New York, NY, USA, 1968. ACM.

[9] Yoshiki Ohshima. On Serializing and Deserializing FRP-style Interactive Programs. Technical report, Viewpoints Research Institute, 2013. VPRI Memo M-2013-001.

[10] Jensen Harris. The Story of the Ribbon. Movies and blogs at: `http://blogs.msdn.com/b/jensenh/archive/2008/03/12/the-story-of-the-ribbon.aspx`.

[11] Alessandro Warth. OMeta/JS 2.0. `http://tinlizzie.org/ometa-js`, also reported in [12].

[12] Alan Kay, Ian Piumarta, Kim Rose, Dan Ingalls, Dan Amelang, Ted Kaehler, Yoshiki Ohshima, Hesam Samimi, Chuck Thacker, Scott Wallace, Alessandro Warth, and Takashi Yamamiya. STEPS Toward Expressive Programming Systems, 2008 Progress Report. Technical report, Viewpoints Research Institute, 2008. Submitted to the National Science Foundation (NSF) October 2008.

A. The layout of the FileList

This is the layout of the File List.

```
layout
  ^ KSSimpleLayout new
    keep: #topLeft  of: 'titleBar' to: 0@0;
    keep: #right    of: 'titleBar' to: #right offset: 0;
    keep: #height   of: 'titleBar' to: 25;

    keep: #topLeft  of: 'directoryField' to: #bottomLeft of: 'titleBar' offset: 10@5;
    keep: #right    of: 'directoryField' to: #right offset: -10;
    keep: #height   of: 'directoryField' to: 20;

    keep: #topLeft  of: 'shortcutListScroller' to: #bottomLeft of: 'directoryField' offset: 0@5;
    keep: #width    of: 'shortcutListScroller' to: 80;
    keep: #bottom   of: 'shortcutListScroller' to: #bottom offset: -35;

    keep: #topLeft  of: 'fileListScroller' to: #topRight of: 'shortcutListScroller' offset: 5@0;
    keep: #right    of: 'fileListScroller' to: #right offset: -10;
    keep: #bottom   of: 'fileListScroller' to: #bottom offset: -35;

    keep: #bottomLeft  of: 'nameField' to: #bottomLeft offset: 10@ -10;
    keep: #height      of: 'nameField' to: 20;
    keep: #right       of: 'nameField' to: #left of: 'accept' offset: -5;

    keep: #bottomRight of: 'cancel' to: #bottomRight offset: -10@ -10;
    keep: #extent      of: 'cancel' to: 60@20;

    keep: #bottomRight of: 'accept' to: #bottomLeft of: 'cancel' offset: -5@0;
    keep: #extent      of: 'accept' to: 60@20;
    yourself
```

B. File List Actions

The code to attach expected behavior to the File List.

```
behavior := (initialFileName) ->
  // shortcuts holds the list of default directories.
  // We don't have a way to add or remove them right now.
  // So it is computed at the start up time.
  shortcutList.setItems(([x.value(), x.key()] for x in shortcuts))
  // When an item in shortcutList is selected, selectedShortcut will be updated.
  selectedShortcut <- shortcuts.first() fby
                      when
                        shortcutList.itemSelected :ev
                      then
                        (e in shortcuts when ev.handler.textContents() == e.key())
  // The following programatically triggers the list
  // selection action for the first item in shortcutList.
  shortcutList.first().fireRequest.set(true)

  // fileName is a field that contains the selected file name.  It uses "startsWith" construct
  // so it is a stream with an initial value.  When itemSelected happens, the string representation
  // of the box (in handler) will become the new value for fileName.
  fileName <- when fileList.itemSelected :ev
              then ev.handler.textContents()
```

```
            startsWith initialFileName

// When the current selection in shortcutList is updated,
// the fileList gets the new items based on the entries in the directory.
fileUpdater <- when selectedShortcut :s
               then
  var dir := s.value()
  var entries := ([{directory: dir, entry: entry}, entry.name()]
    for entry in dir.entries() when patterns.findFirst((p) ->
      p.match(entry.name())) > 0)
    entries := entries.sort((a,b) ->
      a.first().entry.modificationTime() > b.first().entry.modificationTime())
  // update the list in fileList
  fileList.setItems(entries)

// nameField gets a new string when fileName is changed.
updateNameField <- when fileName :name
                   then nameField.textContents(name)

// The contents of the directoryField is connected to shortcut
updateDirectoryField <- directoryField.textContents(selectedShortcut.value().asString())

// fire on this handler and the Box are bound to the fire of the accept button.
fire <- when acceptButton.fire then {dir: selectedShortcut, file: nameField.textContents()}

// Allows the File List to be dragged by the title bar.
label.beDraggerFor(this)
```