KScript and KSWorld: A Time-Aware and Mostly Declarative Language and Interactive GUI Framework

Yoshiki Ohshima, Aran Lunzer, Bert Freudenberg, Ted Kaehler

This paper will be presented at and will appear in the ACM Proceedings of the "Onward! 2013" Conference held in Indianapolis, IN, October, 2013

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 Technical Report TR-2013-002

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

KScript and KSWorld: A Time-Aware and Mostly Declarative Language and Interactive GUI Framework

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 a language called KScript and a GUI framework called KSWorld. The goal of KScript and KSWorld is to try to reduce the accidental complexity in GUI framework code and application building. We aim for an understandable, concise way to specify an application's behavior and appearance, minimizing extra details that arise only because of the medium being used.

KScript is a dynamic language based on the declarative and time-aware dataflow-style execution model of Functional Reactive Programming (FRP), extended with support for loose coupling among program elements and a high degree of program reconfigurability.

KSWorld is built using KScript. 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 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 illustrate our use of KSWorld to build an editor for general graphical documents, including dynamic documents that serve as active essays. The graphical building blocks for documents are the same as those used for building the editor itself, enabling a bootstrapping process in which the earliest working version of the editor can be used to create further components for its own interface.

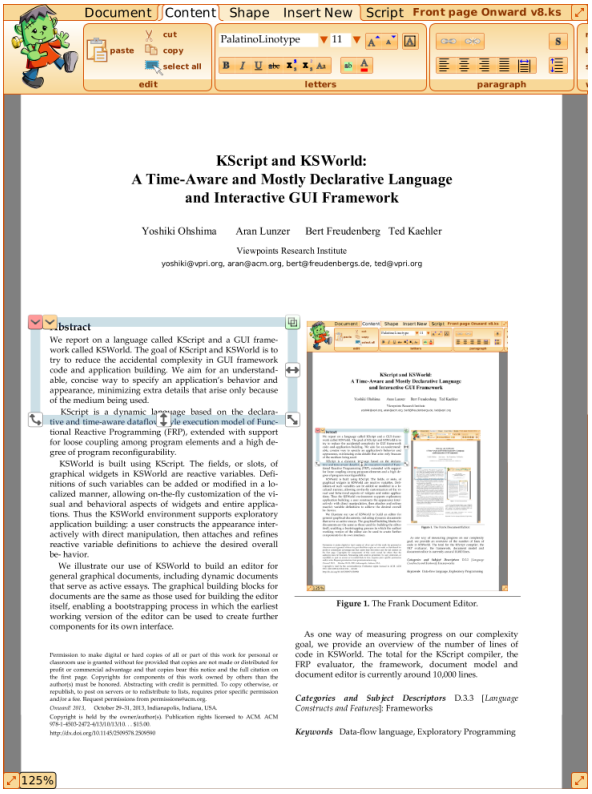


Figure 1. The Frank Document Editor.

As one way of measuring progress on our complexity goal, we provide an overview of the number of lines of code in KSWorld. The total for the KScript compiler, the FRP evaluator, the framework, document model and document editor is currently around 10,000 lines.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Frameworks

Keywords Data-flow language; Exploratory Programming

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 end-user authoring environment. We feel that end-users should be able to make applications of the same kind as those they are using. Toward this goal, the environment they use should not be just an application suite like Microsoft Office, but also an authoring environment like HyperCard or Etoys [2].

From HyperCard we can borrow the simple "stack of cards" document model. To move beyond HyperCard, however, we would like to dissolve the barrier between system-defined and user-defined widgets, making everything uniform. We would also like to be able to embed one object into another without limitation, to construct larger documents. Meanwhile from Etoys we can borrow direct-manipulation authoring, but wish to go beyond Etoys by having a better execution model, especially a better model of time, and making it easy to export and import parts of a project.

We decided to base our approach on interactive construction of applications, and reactive programming. Analogously, the way reactive programming works is similar to spreadsheets: a variable is defined using a formula that refers to other variables, and when any of the input variables (*sources*) changes, the variable that depends on them (the *dependent*) is updated. The dependency relationship is transitive, so updates cascade through the dependency network, which can be seen as a dataflow graph. This matches well the nature of a graphical user interface (GUI), where a large part of the code reacts to changes in objects and time-based events. We feel that the declarative nature of reactive programming makes such code cleaner.

In our current implementation we follow the formulation of Functional Reactive Programming (FRP) [3] for the distinction between continuous and discrete variables, and use combinator names derived from Flapjax [4].

Note that we cannot sacrifice the interactive and exploratory nature of systems like HyperCard and Etoys. Our approach came down to finding a good balance between declarative programming and having the environment be dynamic. We achieved this by incorporating the idea of *loose coupling* into the core of the system. The variables used in the definition of a dataflow node are not resolved at the time of definition. Rather, the references are late-bound and are re-resolved at every evaluation cycle to detect changes. Thus objects in the system are always loosely coupled. Removing or adding variables, making a forward reference to an object that has not yet been defined, and serializing node definitions all become straightforward.

In summary, KScript and KSWorld have the following characteristics:

Dictionary-like objects The base language of KScript resembles JavaScript, but with cleaner syntax and semantics. KScript provides simple object-oriented language features such as methods and fields in objects. An object in KScript is a variable-length dictionary, which supports adding new actions or fields on the fly to support exploratory programming.

FRP-style dataflow programming The base language is extended to support FRP-style dataflow programming. Each dependency description creates a reactive object called an event stream, or more simply a stream. The fields of an object are streams, and all participate in the dependency graph of the running system.

Reified Streams A stream is not only a node in the dependency graph, but also acts as a reified variable with useful capabilities. For example, there is a language construct to obtain the previous value of the stream (essentially this is the same as `pre` in Lucid [5], or `earlier` in Forms/3 [6]). Our stream variables also allow setting a new value for the stream from an interactive programming tool.

Late-bound variable resolution When a formula is defined for a stream, variable names that refer to dependency sources are recorded as keys for looking up the actual streams. Only when the stream is being checked for possible updates are the dependencies resolved, using the object that owns the stream as a namespace. This is the basis of the system's loose coupling between entities.

GUI framework We wrote a GUI framework called KSWorld that takes full advantage of the flexibility and dynamic nature of KScript. Graphical objects in KSWorld are KScript objects, and the user can modify the definitions of fields to construct applications.

Universal Document Editor On top of the GUI framework we built a universal document editor (called Frank) that eanbles the construction of documents and dynamic applications. Frank appears in Figure 1.

The rest of this paper is organized as follows. In Section 2 we explain the basic language features of KScript, then in Section 3 discuss how we have addressed some familiar limitations of the FRP model. Section 4 is an overview of the KSWorld framework, followed in Sections 5 and 6 by examples of building widgets and tool parts of increasing sophistication. In Section 7 we show how these small parts are put together to make an end-user authoring application, followed in Section 8 by examples of dynamic contents built using that application. Section 9 shows a breakdown of the

lines of code in the system as it stands, to give a rough sense of its complexity. Related work is discussed in Section 10.

2. KScript Language

This section describes the KScript language. KScript is a general-purpose language with features suitable for writing graphical applications. It can be considered a hybrid of a simple object-oriented language and reactive programming extensions.

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 encountered in executing a KScript program, or evaluated in the Inspector, 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 stream expression `sound.play()` defines a stream that depends on the `sound` stream, and the resulting stream is bound to the `player` field. When a new value arrives on the `sound` stream, the expression is evaluated with that new value. In this case, the `play()` method of the sound value is invoked so the result of running this program is a stair-like tune played by the speaker. Note that the method `play()` is "lifted" in the same way that the arithmetic operation `%` is lifted: The operations are applied to the values in the streams, rather than the streams themselves.

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 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, as 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. This is a similar observation to the "Reactive" values described in [9].

However, as the original FRP model formalized, mixing behaviors and events in an expression leads to semantic problems, especially when a program is being loaded or initialized. For example, 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 [10]).

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 stream of Point values starting from `(0, 0)` and moving to the positive x-direction over time can be expressed as:

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

where the constructor `P` creates a Point object.

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 (thus not having an initial value), the 0-ary 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()
```

One might think that the `updateConnector()` method could be registered as some kind of callback for these four fields. However there is an advantage to using `mergeE`; the merged stream represents the clear "policy" that specifies what constitutes the condition to trigger `updateConnector()`, and `updateConnector()` has to be mentioned only once in the program.

2.4.3 Field accesses

Fields can contain other objects that in turn contain streams, and it is common that a program needs to combine the streams from different objects. The `mergeE` example above includes such accesses, to the transformation streams within the `start` and `end` objects. A dot notation for a accessing field, such as:

```
start.transformation
```

can be used in the stream definition of another stream as a source stream.

In Sections 4.7 and 7.2, two special "virtual fields" to refer to the top level window object and the document editor handler object are introduced. It is often useful to write a stream definition that involves streams that are fields of the top level window. Having a virtual field allows easy access to them.

2.4.4 `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:

```
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.

2.4.5 `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.

2.4.6 `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`.

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

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

Figure 2. The evaluation method of KSObject in pseudo-code

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. The reason for using a pull-based implementation is that the platform that KScript employs is a synchronized display cycle mechanism. It is not inherent for the semantics of KScript, but it serves well for the design of the platform.

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 sort the streams topologically into a linear 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 2 shows the evaluation scheme in pseudo-code. See Section 4.5 for more information.

3. Dealing with Issues in the FRP Model

The original FRP provides very clean semantics and helps programmers to reason about the program statically. However, there are two problems we needed to deal with to achieve our goal of making an interactive environment.

One of the major problems with FRP is that you cannot have a circular dependency among streams. Unfortunately, circular dependencies do tend to arise in GUI programming. For example, imagine that we are creating a text field with

a scroll bar. When the user types new text into the field, the visible area of the text may change so the location of the knob in the scroll bar may have to be changed. At the same time, however, any change in the knob position (such as when dragged by the user) should change the visible area of the text. This is a circular dependency.

Also, imagine if there is support for turtle geometry. The key concept in turtle geometry is specifying the turtle's movement in differential form: for example, the command `rt 2` computes a new value of the `heading` variable from its old value. If the program is naively written as

```
heading <- heading + 2
```

it would mean that the `heading` variable depends on itself, becoming an even more direct form of circular dependency. For this to mean anything sensible, there needs to be a way to distinguish the old and new values of a variable.

Another problem is the static nature of FRP. To support a more exploratory style of programming, we need a more dynamic language.

3.1 Setting values into streams

We want an inspector on an object to allow the user to change the object's values and stream definitions on the fly. Similarly, it should be possible to change a graphical object's position and geometry interactively via the halo mechanism (see Section 4.8).

To support such actions, a stream supports an operation called `set`, which sets a new current value. It is typically used on value streams (i.e., streams defined without dependency sources). For example, there is a stream that represents the geometry transformation matrix of a `Box` (a basic graphics widget). In a pure form of FRP a value stream would truly stay constant, but by use of `set` we can allow the value to be updated in response to direct manipulation and exploratory actions by a user. This is analogous to the `receiverE` and `sendEvent` mechanism in Flapjax.

When a stream is updated via `set`, the dependents of the stream will be updated with the new value. However, the issue that arises is that the topological sorter for streams cannot detect the dependency when `set` is used from imperative code. As a consequence, the order for the dependents to be updated can vary based on the sorting result. This approach does give rise to temporarily inconsistent values ("glitches", in FRP terminology), but we opted to deal with these in the cases where they arise.

For the case of a text field with scroll bar, instead of specifying the positions of the text area and the scroll knob using mutually dependent streams, we use side-effecting methods that request value changes using `set` when necessitated by a change in the other stream.

3.2 Accessing the previous value

It is convenient to be able to access the previous value of a stream. In KScript, when the prime mark (`'`) is attached to a variable name referencing a stream, it evaluates to the previous value of the stream. This can be used in computing a new value. Consider this example:

```
nat <- 0 fby when timer then nat' + 1
```

The stream `nat` starts with 0, and recomputes its value whenever the stream called `timer` is updated. The new value is the previous value incremented by 1.

Note the use of `when-then`. Imagine if a user forgot to specify the trigger (`timer`), and wrote:

```
nat <- 0 fby nat' + 1
```

Because a variable with the prime mark is not registered as a dependency source, this stream would never update. The `when` clause is thus a way to specify additional dependencies.

Accessing the previous value also allows mutually dependent streams to be computed simultaneously. For example, one can define a pair of values that each follows the other:

```
a <- true fby when timer then b'
b <- false fby when timer then a'
```

4. KSWorld: the GUI framework

KSWorld is a GUI framework that supports exploratory reactive programming. The ideas on how to structure graphical objects are drawn from Morphic [11], Lessphic [12], and Tweak [13]. The framework maintains a 2.5-dimensional display scene as a tree whose nodes are graphical objects called `Boxes`, and where a parent/child relationship between nodes signifies containment.

4.1 Boxes

A `Box` inherits from `KSObject` and serves as the entity that a user sees and interacts with. It manages the streams needed to make it behave as a graphical object. The `container` stream represents the container (parent) in the display tree; `contents` holds the contained (child) `Boxes` as a collection; `shape` represents the visual appearance; and `transformation` is a 2×3 transformation matrix relative to its container. There are also derived streams such as `extent`, which is the inherent extent of the `Box`, and `bounds`, which is computed by transforming `extent` into the container's coordinate system. Since these streams need to have a value all the time (i.e., from the moment the `Box` is instantiated), they are defined as behaviors in the FRP sense, with meaningful initial values.

Taking the idea of a uniform object model seriously, we made even the individual characters in a text field be separate `Boxes`.

4.2 Graphics model

From the STEPS project we use a full-featured vector graphics engine called Gezira [14]. The shape property of a `Box` holds an object that packages quadratic Bézier contour definitions along with fill and stroke data. We also provide a

canvas abstraction on top of the core Gezira engine and in normal operation render all Boxes with Gezira[1].

Each of the character Boxes mentioned above holds vector data for its shape, created from a TrueType glyph.

4.3 User event routing

When the framework receives a user event (such as "buttonDown", "keyUp", etc.) from an external device, it must decide which Box should handle the event. If there is a Box holding the "focus", the event goes there; otherwise the framework traverses the display scene depth-first to find the deepest Box that contains the position of the event and has a value stream whose name matches the event type. For instance, if a Box is interested in buttonDown, it declares this using:

```
buttonDown <- eventStream()
```

When the framework finds the appropriate recipient, the event will be set into the recipient's stream, and any dependent streams will be triggered during the subsequent evaluation phase.

The most of the framework, such as the routing of events, and the management of the display tree is mostly written in a procedural rather than reactive style. The reason is that routing events requires ordering, which is easier to express in a procedural manner. For example, imagine that there are several Boxes in a display scene that have a way of responding to mouse-over events. A purely reactive description of the response for each box would be: "when the mouse pointer is within my bounds, react to it like this". However, the 2.5-dimensional structure of the display dictates that when Boxes overlap only the front-most Box containing the event location should react. To orchestrate this choice in purely reactive code would be awkward.

4.4 Event triggering

Once events are delivered to the dataflow model, specifying the reaction is simple. For example, the following stream definition makes the Box owning the stream jump to the right when it receives a mouse-down event:

```
myMover <- when buttonDown then this.translateBy(P(10, 0))
```

Actions that involve multiple objects besides the owner of a stream typically refer to those objects through fields in the owner. For example, the code below defines a stream that keeps the top left of the stream-owning Box coincident with the top left of its container's first child Box.

```
otherBox <- streamOf(container.first())
myAligner <- this.topLeft(otherBox.bounds.topLeft())
```

The stream otherBox is initialized with the Box whose bounds are to be watched (the container's first child). The myAligner stream refers to otherBox and responds whenever there is a change in that Box's bounds, or if otherBox is set to a different Box.

[1] We also support an OpenGL back-end that uses the same canvas abstraction and displays Gezira-generated textures.

```
while true
    var currentTime := getMilliseconds()
    registerEvents()
    var evaluator := Evaluator.new()
    window.withAllBoxesDo((box) ->
        evaluator.addStreamsFrom(box))
    evaluator.sortAndEvaluateAt(
                window.mapTime(currentTime))
    // layout phase
    window.withAllBoxesDo((box) ->
        box.layOut())
    window.draw()
    sleepUntilNextFrame()
```

Figure 3. The top-level loop of KSWorld in pseudo-code

4.5 The top-level loop

Figure 3 illustrates the top-level loop of KSWorld. The getMilliseconds() function retrieves the physical wall-clock time. In registerEvents(), each raw event delivered to the system since the last cycle is routed to an appropriate Box. After this, the display tree is traversed to build the dependency graph that will be triggered by the raw events and timers (withAllBoxesDo() applies the given function to all contained boxes). As described below, the graph itself can change from one display cycle to the next, so on each cycle we sort the streams (with a simple caching scheme, described in Section 9.1) and then evaluate them. The argument to sortAndEvaluateAt() is a number representing the logical time to be used in evaluating the streams; the mapTime() method derives this logical time from the physical time. After the evaluation, the layout for all Boxes is performed as a separate phase, and the loop then sleeps until the next cycle. In KSWorld a typical cycle rate is 50 frames per second.

The KSWorld is hosted in the Morphic framework in the Squeak Smalltalk environment. A Squeak graphical object (Morph) holds a bitmap (Form) for allowing Gezira engine to render onto it. Morphic events delivered to the Morph are converted to KSObjects that represent events. In other words, Squeak provides only a minimum of support for the graphics and events.

4.6 Box layout

One of the important features of a GUI framework is a layout mechanism. It might seem appealing to write a layout as a set of dependency relationships between Boxes' bounds streams, but since a typical layout specification involves relationships that are multi-directional, the dependencies would tend to become circular.

Therefore we use procedural code for this part of the framework too. A Box can be given a layout object that is responsible for setting the locations and sizes of each of its children. The layout phase of the top-level loop traverses the

display tree and triggers these layout objects. The resulting calculations cause changes in the Boxes' `transformation` and `bounds` streams; any other streams that depend on these will get their chance to respond in the next cycle.

Our standard layout objects include a (multi-directional) constraint-based layout, and layouts for vertical and horizontal lists.

4.7 Special objects in the display scene

There are two Boxes that receive special support from the framework. One of them is the Window, which represents the top-level Box in the display scene. Besides being the top node in the tree, it maintains global system information and provides the interface to the external environment, such as receiving user events.

The second special Box is the Hand, which is the abstract representation of the user's pointing device. It too behaves as a normal Box in the display scene, except that it usually has its position correlated with the location of the pointing-device cursor.

It is common to access the Window from a Box. To integrate this lookup in the late-binding resolution mechanism, we add a virtual field called `__topContainer__` to each Box. Each time this field is accessed it looks up the Box's current `container` chain and returns the top-level Box, which is the Window.

4.8 Halo

For interacting with graphical objects in KSWorld we provide a halo mechanism [15]. The halo is a highlight around the selected target Box (seen as the blue frame in Figure 1), and provides direct-manipulation means for moving, rotating, resizing and scaling the target without triggering the target's own actions. For example, the halo allows a user to move or resize a button without triggering the button action. As the halo itself exists within the uniform object model, it is made up of Boxes and uses dependencies to track changes in the target, such as for repositioning and resizing itself when the target's `transformation` or `bounds` values are changed by running code.

One problem is that this tracking would introduce a circular dependency if naively implemented, given that when the user moves the halo the target should follow it, but conversely when the target Box is moved by code the halo should follow the target.

We again get around this circular dependency with the help of `set` (see Section 3.1). When the user drags the halo, the target's `transformation` is updated via the `set` mechanism. In the opposite direction, when the target Box is moved by code, the halo moves to its appropriate position during the layout phase.

5. Building Basic Widgets

5.1 Buttons

The goal of KSWorld is to support not only applications with pre-made widgets, but also to be able to write such widgets and customize them. In other words, we would like to write everything in the same framework. We begin with a button widget as an example, to see how compactly it can be written. The button should have a small but useful set of features, such as being able to configure whether it fires on press or on release, and providing various forms of graphical feedback based on its state.

For bootstrapping, the code shown below may be written in a text editor, but once the system is working the stream definitions can be given interactively in the Inspector tool (as shown later).

A button needs to handle and interpret pointer events. As described in Section 4.3, value streams are created and bound to the event type names in the Box that is to receive them[2]:

```
buttonDown   <- eventStream()
buttonUp     <- eventStream()
pointerLeave <- eventStream()
pointerEnter <- eventStream()
motionQuery  <- eventStream()
```

Further streams are defined to represent the button's state:

```
pressed <- false fby
             mergeE(buttonDown.asBoolean(),
                    not buttonUp.asBoolean(),
                    not pointerLeave.asBoolean())
entered <- false fby
             mergeE(pointerEnter.asBoolean(),
                    not pointerLeave.asBoolean())
actsWhen <- streamOf("buttonUp")
selected <- streamof(false)
```

The function `asBoolean()` treats `null`, `undefined`, and `false` as false and everything else as true. Thus an expression `buttonUp.asBoolean()` generates a true value each time the `buttonUp` stream is updated with a new event. The `mergeE` for `pressed` updates itself whenever a `buttonDown`, `buttonUp` or `pointerLeave` event is received, and returns a boolean value that reflects which of those events happened most recently: true if it was a `buttonDown`, false for either of the other two. So `pressed` is true when a `buttonDown` has been received and has not yet been followed by a `buttonUp` or `pointerLeave`. The `entered` state similarly looks at `pointerEnter` and `pointerLeave`.

We can now write the definition of the `clicked` stream, which is to become true when the user has released the mouse over a button Box that was previously in its `pressed` state (i.e., filtering out cases where the pointer drifts off the button before the mouse is released):

```
clicked <- buttonUp.asBoolean() && pressed'
```

[2] For convenience, a method `listen()` can be used to create a set of event streams from a list of the event names.

Note the use of the prime mark on `pressed` to indicate that it is the previous value that is of interest. Note too that the prime mark means that `pressed` is not registered as a dependency source of `clicked`, which is as we want for this stream: `clicked` should only be updated when a new `buttonUp` arrives, not whenever `pressed` changes.

Based on these states and events, we can write a stream that truly makes a button be a button; namely, the `fire` stream, which updates when the button triggers. When does a button fire? In usual cases, clicking (mouse down and then up) is the right gesture, but for some interactions we may want the button to fire as soon as the mouse is pressed. To support this, a variable called `actsWhen` is added, and the `fire` stream definition looks like this:

```
fire <- when (if (actsWhen == "buttonUp" && clicked ||
                  actsWhen == "buttonDown" &&
                  buttonDown.asBoolean()) == true
                true
              else
                undefined)
        then
          var ev := if actsWhen == "buttonUp"
                      buttonUp
                    else
                      buttonDown
          {item: this, event: ev}
```

The `when` part of this definition confirms a valid confluence of settings and events for the button to trigger, and the `then` part makes a new object with `item` and `event` fields to denote which object fired in response to what event.

The code from the definition of `pressed` through to that of `fire` is enough to turn a plain `Box` into a functioning button, yet amounts to only about 12 lines (though we have folded them here to suit the article format).

One of the benefits of this style of description is a clear separation of concerns. The button's responsibility is just to set a new value on its `fire` stream, so there is no need for clients to register callbacks. And the specification of transitions among logical states is separate from that of the graphical appearance.

So now let's add the appearance. The stream that represents the current graphics appearance is called `looks`, and each value it takes is a dictionary of `fill` and `borderFill` for the button. The value is computed when the state of the button changes. There is another stream named `changeFill` that calls side-effecting methods `fill()` and `borderFill()` to cause the actual change of appearance in the button `Box`:

```
highlightEnabled <- streamOf(true)
looks <- this.defaultLooks() fby
           when mergeE(entered, pressed, selected)
           then ...

changeFill <- when looks :f then
    if f.fill
        this.fill(f.fill)
    if f.borderFill
        this.borderFill(f.borderFill)
```

5.2 Menus

In KSWorld, a menu is simply a coordinated collection of buttons. The first part of the method that sets up a `Box` to act as a menu is written procedurally, and creates the right number of buttons based on an argument that lists the menu items. These buttons are stored in the menu `Box`'s `items` field. Then the second part of the method sets up the streams to bring the menu to life:

```
items <- ...        // the ordered collection of buttons
fire <- anyE(items, "fire")
```

Effectively the menu itself behaves like a big button, with its own `fire` stream. This stream uses `anyE` to detect when any button in the `items` collection fires, and stores the button's `fire` event (as described earlier) as its own new value. The `item` field in that event holds the button itself, which is how a client of the menu can see which item fired.

6. Building Tools

In this section we show how larger widgets can be made interactively in KSWorld. Having written the code for buttons and text layout with the help of tools in the hosting environment, we are now starting to bootstrap the system in itself.

6.1 File list

The first tool we are going to make is the File List, shown in Figure 4. In a File List there is a list of directories, a list of files in the selected directory, and a field to show the selected directory and file. Pressing the Accept button will trigger some action, while pressing the Cancel button will close the File List without triggering.

The steps in making a tool in KSWorld are as follows:

- Make a compound widget.
- Edit the properties and styles with the Inspector and the ShapeEditor, if necessary.
- Write code to specify the layout, if necessary.
- 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.
- Write code to set up the widget with the layout and actions.

We start from an empty `Box`. By using the default halo menu we add new `Box`es into it, then use the halo to resize and roughly place each of them to get a feel for the eventual layout.

A rudimentary Inspector tool allows us to inspect the values of an object and execute KScript expressions in the object's context. Using the Inspector we give each `Box` a name, and set its fill and border style:

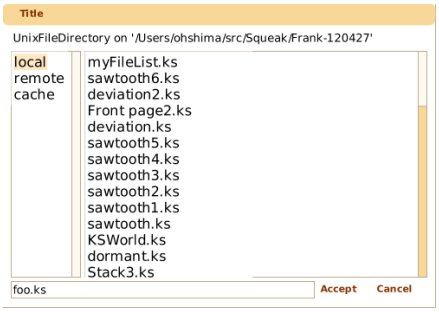
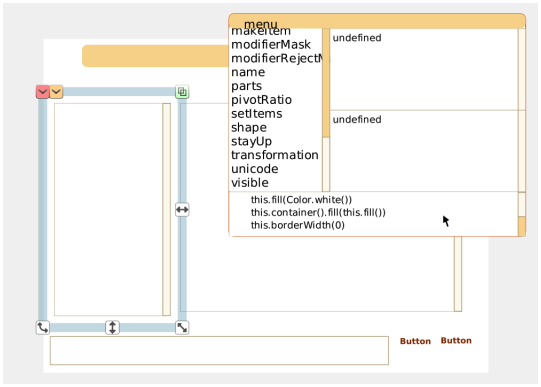


Figure 4. The File List.



Set fills.

At this point we can also use the Inspector to attach certain behaviors to Boxes to customize them: some are turned into buttons, some into lists.

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()}
```

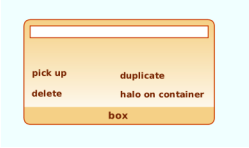
where selectedShortcut is the currently selected shortcut and nameField is a Box that is showing the currently selected file name. This definition specifies that when the acceptButton's fire stream is updated, the fire stream of the File List itself will acquire a new value that is an object with two 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; its sole job is to set a new value into the fire stream. Thus developing the File List and its various 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.

6.2 A panel for the tool bar

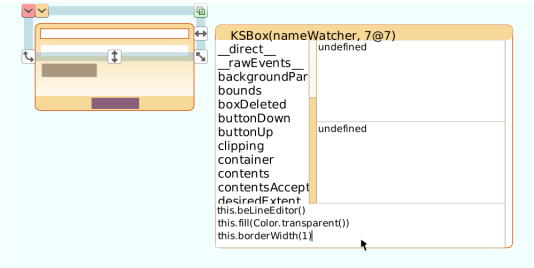
We now demonstrate how we make a panel, also known as a bubble, containing commands for the Document Editor.



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

The first step is to create a Box to be the bubble, and add an appropriate gradient fill and corners. 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:



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 (see Sections 2.4.3 and 7), so __docEditor__.selectedDocBox refers to the selected Box, and the result is converted to a string and shown in this Box.

The panel contains a number of buttons, making it conceptually similar to the way we defined a menu. As in the menu, the panel consolidates the fire streams of its children into its own fire stream:

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

Again, this form of implementation allows largely independent development of the panels' clients, the panels 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 panels is free to explore alternative organizations of commands.

7. Putting 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 [16]. 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 7.4).

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 7.5) 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).

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 The Document Editor handler

The core of the Document Editor is implemented as a "handler" object that is attached to a Box that represents the Document Editor. The handler can be referenced from other Boxes via the __docEditor__ virtual field. It is presented as a field so that the streams for tool bars and other widgets in the handler can pick up the changes in the streams of the Document Editor.

In a handler, the associated Box is stored in a field named whole.

7.3 Bubble selection

The current target of the halo is held in a stream called haloTarget of the Window Box (described in Section 4.7). To customize the editor interface depending on the highlighted Box, the Document Editor needs a stream that depends on haloTarget. 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 Doc-

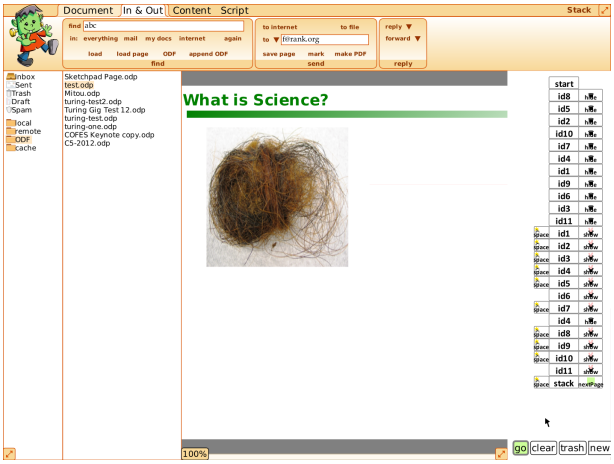


Figure 5. The Directory Browser on the left, and the Scripting Pane on the right.

ument 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 highlighted). 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()
```

7.4 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.

7.5 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 5 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. Example Documents

We now show examples of dynamic documents that were made in the Document Editor (notice also that the first screenshot in this paper shows a recreation of the paper's own title page).

8.1 An active essay on standard deviation

We are especially interested in interactive documents that capitalize on the computer's ability to demonstrate abstract ideas concretely and visually. The first example here is to explain the concepts of average and standard deviation. Imagine that we are creating an online encyclopedia article: rather than just having a static page, or some non-interactive animated GIFs, the article should provide interactive features that let the reader explore the topic.

Figure 6 shows the essay. The text is a simple explanation of the two concepts, but what is notable is the interactive aspect. There are seven sliders representing numbers, that the user can adjust by moving the slider knobs up and down. A moving horizontal line represents the current average of the numbers.

In addition, the bottom half of the text contains numeric readouts. These are in fact live spreadsheet cells, though lib-

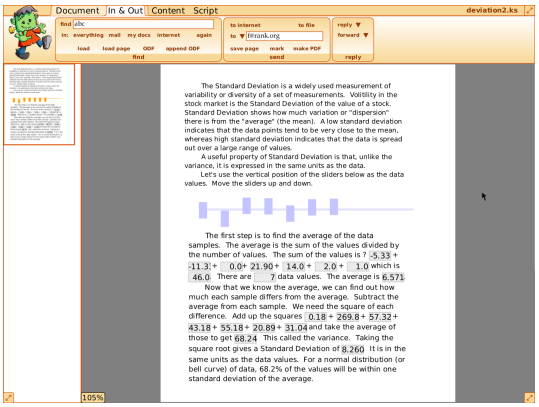


Figure 6. An Active Essay on Standard Deviation.

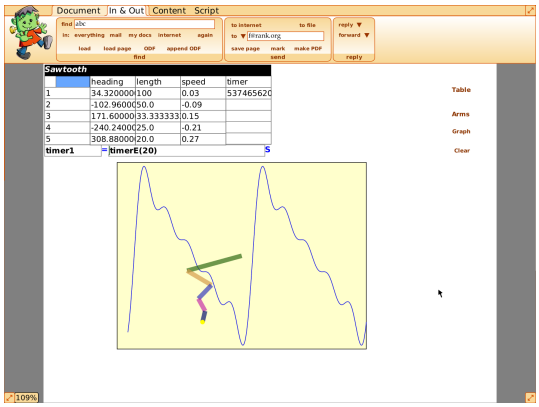


Figure 7. An Active Essay on The Fourier Series.

erated from the two-dimensional grid of a typical spreadsheet application. The spreadsheet-like nature of FRP makes it straightforward to write for each cell a stream that generates the cell's value in terms of other values, both for the cells that follow the number sliders directly and for those that represent steps in the calculation of the standard deviation.

8.2 An active essay on Fourier series

One of the interesting ways of visualizing the concept of a Fourier series is through a Phaser, a computerized animation developed by Danny Hillis based on an idea from Viki Weisskopf. The key is that a sine function can be visualized with a rotating line segment. When it is rotating at a constant rate around one end, the vertical position of the other end represents the sine function. Summing a series of sine functions of different amplitudes and frequencies can be achieved by visualizing each function as a line segment with appropriate length and rotation rate, and pinning the start of each line to the end of the one before. The oscillating vertical position of the end of the last line represents the moment-to-moment sum of the series.

Figure 7 shows an editable Phaser setup for use in explaining Fourier series. It includes a spreadsheet with columns `heading`, `length` and `speed`. Each row provides the data for one of five line segments instantiated in this example. The `heading` cells contain formulas that depend on a value held in the `timer1` cell. When the formula for this cell is set to be a steadily increasing time provided by a stream `timerE(20)`, the animation starts and the arms show the visualization of the Fourier series. The line graph is plotting the vertical position of the tip of the final arm.

Both of these examples were created in the Document Editor, making use of a set of commands for instantiating customized forms of `Box`. For example, the New Cell command creates a new (free-floating) spreadsheet cell that can be embedded into a another `Box`, such as a text field, where it can be further customized with the help of the Inspector tool.

9. Complexity and Performance

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 this system. The elements that are summarized in the first subtotal (10,055) are considered to be the essential part of the system for implementing the Document Editor. The next entry, "Gezira Bindings", is semi-essential. The remaining parts are not essential for making the Document Editor, but help with optimization and development.

Detailed discussion of each of the table items is beyond the scope of this paper (refer to [17] for more detailed information), but here we would like to make a few remarks:

First, note that KSWorld is currently hosted in the Squeak Smalltalk 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. The primitive objects in KScript, such as Numbers, Strings, Arrays, and Dictionaries are mapped to those of Squeak. Note that programs written in KScript use limited set of the protocols supported by these classes.

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 in Smalltalk. Dataflow-based features are considerably more compact. As one example, a comparable implementation of the FileList explained in Section 6.1 took about 250 lines in Smalltalk, as opposed to 50 lines in KScript.

Lines of code is a metric of static complexity. But how about dynamic complexity? When an empty Document Ed-

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

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

itor is started, it contains about 530 `Boxes`, including all the characters in the labels in buttons, bubbles, and other controls. Each such `Box` contains 13 to 18 streams, so the dependency sorter must handle about 8,800 streams. Because each character in a document is also handled as an individual `Box`, when there are 4,000 characters in the current page (as in Figure 1) there are roughly 80,000 streams involved. Most of these, however, are mundane value streams that represent the `transformation`, `shape`, `container`, etc. The number of streams with non-trivial expressions is quite low.

9.1 Performance

Even though our work emphasizes simplicity of the model over run-time performance, when running on a MacBook Pro computer with an Intel i7 2.3GHz processor the system is comfortably responsive (achieving 30 to 50 fps) in common cases.

Two major tasks consume most of the execution time. One is the graphics rendering. For simplicity we have so far omitted damage region management, and the system can spend 70% of its time rendering `Boxes` when the display tree is complex (noting once again that each character `Box` is rendered individually).

Another major taks is that of sorting streams topologically. As long as the set of streams in the system is steady, the sorted result can be cached, but when a new object or new stream is introduced the cache is discarded and reconstructed. If this happens often (such as when editing text from the keyboard) the system does become sluggish, sometimes achieving as little as 2 fps.

One way to address the latter problem would be to make characters no longer have their own `Boxes`, or to use fewer reactive streams in their implementation. Less draconian changes, such as finer-grained use of caching, could also achieve the necessary performance improvements.

10. Related Work

There has long been an interest in time-aware computation, with a history going back to John McCarthy's Situation Calculus [18]. Recently, Dedalus [19] provides a clear model that scales to distributed execution based on Datalog. For making a GUI framework for a single-node computer, however, we need a more orderly execution model, as we expect that what we see on screen is the snapshot of "quiescent" states at regular intervals. Also, we needed to allow the side-effecting `set` operation for streams. This is certainly a great area for future research.

Lucid provided a nice syntax for describing the concept of a variable being a stream of values, and provided a clean formulation of the concept. Lucid lacks the distinction between continuous values and discrete values; we found this distinction very useful in thinking about graphical applications.

FRP can be seen as an equality-based, uni-directional constraint solver. In the past, there have been attempts to apply constraints to GUI frameworks. Most notably, Garnet [20] provided a similar feature set to KScript and KSWorld, such as being able to have uni-directional constraints, or formulas, in the slots of graphical objects that the system then satisfies (it also had an equivalent of the `set` operation). Garnet had an interface builder as well. However, it did not have a time-aware execution model, and the system was not designed for exploratory system construction.

On the cleaner semantics front, the Constraint Imperative Programming language family, such as the versions of the Kaleidescope language [21], are notable. They used multi-directional constraint solvers that can handle non-equality, and the concept of assignment is incorporated in the framework. On the other hand, the cleaner semantics has some limitations. When the data involved in the framework ranges over colors, transformation matrices, bounding boxes encompassing Bézier curves, etc., we don't see that a multi-directional solver would give reasonable results. (A multi-directional solver could emulate one-way constraints when necessary; the challenge is finding a good trade-off between expressiveness and simplicity.)

Animus, by Duisberg and Borning [22], was an early constraint-based GUI framework with a theory of time.

Another GUI framework that had the idea of being based on spreadsheet-like uni-directional constraints was Forms/3. Compared to Forms/3, our system provides higher-level organization concepts such as embedded graphical objects to support applications and projects that are much bigger. Also, more importantly, our system aims to be **self-sustained**. The

editors, inspectors, etc. used in the authoring system should be written on top of the same system.

One direction for further work is to investigate tighter integration of the FRP concepts of behaviors and events with multi-way constraints, and with more recent solvers such as Cassowary [23].

Modern GUI frameworks often have a binding mechanism with a special language or syntax for specifying notifications and describing simple constraints among variables. Examples of them include the SML language for Qt [24], JavaFX 1, etc. However, they are mostly designed for making form-based applications that do not involve adding or removing new user objects.

11. Conclusions

We presented a new language called KScript for writing interactive graphical applications and a universal document editor written in it.

The FRP-derived model worked well for our goal. We wanted to describe a highly interactive exploratory application building system that is written with a time-aware execution model in declarative and compact style. With the late-bound variable lookup scheme, the system can be written as a set of dataflow graph nodes. The example application, the Document Editor, is a good validation of the language's expressiveness and flexibility to support dynamic changes for exploratory programming.

While the base model is functional we allow a non-pure operation to update the value in a stream for facilitating direct manipulation and breaking circular dependencies. A consequence of this was glitches in the timing of evaluation. In the future we would like to have a better formalization in this regard.

The pull-based implementation strategy was the result of the characteristics of the underlying implementation system, where it has a periodic event loop. We also would like to explore different implementation strategies.

Acknowledgments

We would like to thank the reviewers for providing us insightful comments.

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 early design discussions and the implementation of the `KSObject` inspector; Alex Warth for the language-building ideas and creating OMeta; Dan Ingalls for proof reading; 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.

This paper is based upon work supported in part by the National Science Foundation under Grant No. 0639876.

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

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

[4] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for Ajax applications. *SIGPLAN Not.*, 44(10):1–20, October 2009.

[5] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

[6] Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *J. Funct. Program.*, 11(2):155–206, March 2001.

[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] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

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

[11] Mark Guzdial and Kimberly Rose. *Squeak: Open Personal Computing and Multimedia*, chapter 2: An Introduction to Morphic: The Squeak User Interface Framework, pages 39–68. Prentice Hall, 2002.

[12] Ian Piumarta. Lessphic. `http://piumarta.com/software/cola/canvas.pdf`.

[13] Andreas Raab. Tweak. `http://wiki.squeak.org/squeak/3867`.

[14] Dan Amelang. Gezira. `https://github.com/damelang/gezira`.

[15] Edwin B. Kaehler, Alan C. Kay, and Scott G. Wallace. Computer system with direct manipulation interface and method of operating same, 12 1992. US Patent: 5,515,496.

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

[17] Yoshiki Ohshima, Aran Lunzer, Bert Freudenberg, and Ted Kaehler. Making Applications in KSWorld. Technical report, Viewpoints Research Institute, 2013. VPRI Memo M-2013-003.

[18] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[19] Peter Alvaro, William Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, Russell C Sears, Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. Technical report, University of California, Berkeley, 2009.

[20] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, David S. Kosbie, Edward Pervin, Andrew Mickish, Brad Vander Zanden, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, 1990.

[21] Bjorn Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 268–286, June 1992.

[22] Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Trans. Graph.*, 5(4):345–374, October 1986.

[23] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, UIST '97, pages 87–96, New York, NY, USA, 1997. ACM.

[24] The Qt Project. `http://qt-project.org`.

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)
```