STEPS Toward the Reinvention of Programming, 2012 Final Report Submitted to the National Science Foundation (NSF) October 2012

(In random order) Yoshiki Ohshima, Dan Amelang, Ted Kaehler, Bert Freudenberg, Aran Lunzer, Alan Kay, Ian Piumarta, Takashi Yamamiya, Alan Borning, Hesam Samimi, Bret Victor, Kim Rose

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-2012-001

Viewpoints Research Institute, 1025 Westwood Blvd 2nd flr, Los Angeles, CA 90024 t: (310) 208-0524

2012 REPORT

STEPS Toward Expressive Programming Systems

"A Science Experiment"

by

(In random order) Yoshiki Ohshima, Dan Amelang, Ted Kaehler, Bert Freudenberg, Aran Lunzer, Alan Kay, Ian Piumarta, Takashi Yamamiya, Alan Borning, Hesam Samimi, Bret Victor, Kim Rose*

VIEWPOINTS RESEARCH INSTITUTE

*These are the STEPS researchers for the year 2012. For a complete listing of the participants over the length of the project see the section on the NSF site which contains this report.

TABLE OF CONTENTS

The STEPS Project for the General Public

Summary of the STEPS project

- Origin
- STEPS aims at personal computing
- What does STEPS look like?
- STEPS is a "science project"
- General approach
- "T-shirt programming"
- Overall map of STEPS
- General STEPS results
- Making "runnable maths"
- Assessing STEPS

Summary of STEPS research in 2012

- Inventing & building a language for STEPS "UI and applications", and rewriting Frank in it
- Producing a workable "chain of meaning" that extends from the screen all the way to the CPU
- Inventing methods that create automatic visualizers for languages created from metalanguages
- Continuing the start of "What vs How" programming via "Cooperating Languages and Solvers"

Reflections on the STEPS Project to Date

References

Outreach Activities for final year of NSF part of the STEPS project

Publications for the NSF part of the STEPS project

Appendix I: KScript and KSWorld

Appendix II: Making Applications in KSWorld

The STEPS Project For The General Public

If computing is important—for daily life, learning, business, national defense, jobs, and more—then *qualitatively advancing computing* is extremely important. For example, many software systems today are made from millions to hundreds of millions of lines of program code that is too large, complex and fragile to be improved, fixed, or integrated. (One hundred million lines of code at 50 lines per page is 5000 books of 400 pages each! This is beyond human scale.)

What if this could be made literally 1000 times smaller—or more? And made more powerful, clear, simple, and robust? This would bring one of the most important technologies of our time from a state that is almost out of human reach—and dangerously close to being out of control—back into human scale.

An analogy from daily life is to compare the great pyramid of Giza, which is mostly solid bricks piled on top of each other with very little usable space inside, to a structure of similar size made from the same materials, but using the later invention of the arch. The result would be mostly usable space and require roughly 1/1000 the number of bricks. In other words, *as size and complexity increase, architectural design dominates materials.*

The "STEPS Toward Expressive Programming Systems" project is taking the familiar world of personal computing used by more than a billion people every day—currently requiring hundreds of millions of lines of code to make and sustain—and substantially recreating it using new programming techniques and "architectures" in dramatically smaller amounts of program code. This is made possible by new advances in design, programming, programming languages, systems organization, and the use of science to analyze and create models of software artifacts.

Summary of the STEPS Project

Origin—the STEPS research project arose from asking embarrassing questions about many systems (including our own) such as: "Does this system have much too much code and is it messier than our intuition whispers?" Almost always the answer was "yes!" We wanted to write much smaller code, have it be more understandable and readable, and if possible, to have it be "pretty", even "beautiful".

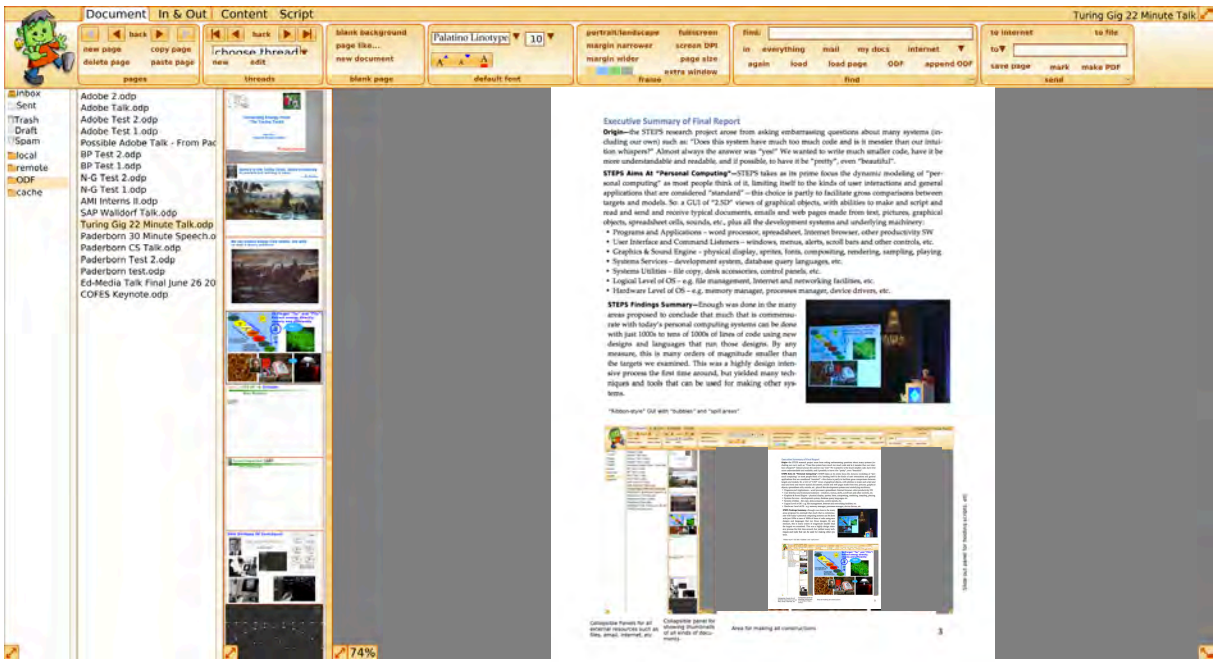
STEPS Aims At "Personal Computing"—STEPS takes as its prime focus the dynamic modeling of "personal computing" as most people think of it, limiting itself to the kinds of user interactions and general applications that are considered "standard"—this choice is partly to facilitate gross comparisons between targets and models. So: a GUI of "2.5D" views of graphical objects, with abilities to make and script and read and send and receive typical documents, emails and web pages made from text, pictures, graphical objects, spreadsheet cells, sounds, etc., plus all the development systems and underlying machinery:

- Programs and Applications – word processor, spreadsheet, Internet browser, other productivity SW
- User Interface and Command Listeners – windows, menus, alerts, scroll bars and other controls, etc.
- Graphics & Sound Engine – physical display, sprites, fonts, compositing, rendering, sampling, playing
- Systems Services – development system, database query languages, etc.
- Systems Utilities – file copy, desk accessories, control panels, etc.
- Logical Level of OS – e.g. file management, Internet and networking facilities, etc.
- Hardware Level of OS – e.g. memory manager, process manager, device drivers, etc.

What Does STEPS Look Like?—It looks and acts like familiar—but more integrated—personal computing! It has "universal documents" that can model all document forms, a user interface authoring approach that is like an extended form of presentation authoring, and an independent repertoire of ways to find, fetch, send, save, broadcast, etc., the documents. This subsumes the normal scattered "productivity suite" intractable stovepipes often provided instead of a truly integrated system (see below).

Presentation at Turing Centenary done in STEPS

"Ribbon-style" GUI with "bubbles" and "spill areas"



Slide out panel for holding scripts, etc.

Collapsible Panels for all external resources such as files, email, Internet, etc.

Collapsible panel for showing thumbnails of all kinds of documents

Area for making all constructions

The STEPS "Cute Frank" General Authoring and Document Interface

STEPS Is A "Science Project"—STEPS is not aimed at producing a new practical alternative to existing PC operating systems and applications. We do want to create a *model* that is much more compact, understandable, explanatory and easy to work with than renderings in normal code. In addition we'd like to have the model validated by being able to run fast enough to be interacted with as a real personal computing system. In other words, we are most interested in what it takes to find and represent the *meaning* of a large runnable system. Thus the STEPS project is a kind of "science experiment" rather than an engineering project (which still required a considerable amount of careful software engineering to pull off). For details, we suggest looking at the original proposal [NSF] and the yearly reports [STEPS Reports].

The original plan for lines of code counting was only to count "meaning-laden" code, not optimizations. Our aim here was to see what could be done with about 20,000 lines of meaning-code for everything "down to the metal". This would be slow on a conventional computer but could be tested in real-time using a supercomputer, or by adding optimization code (but no new meanings) "on the side".

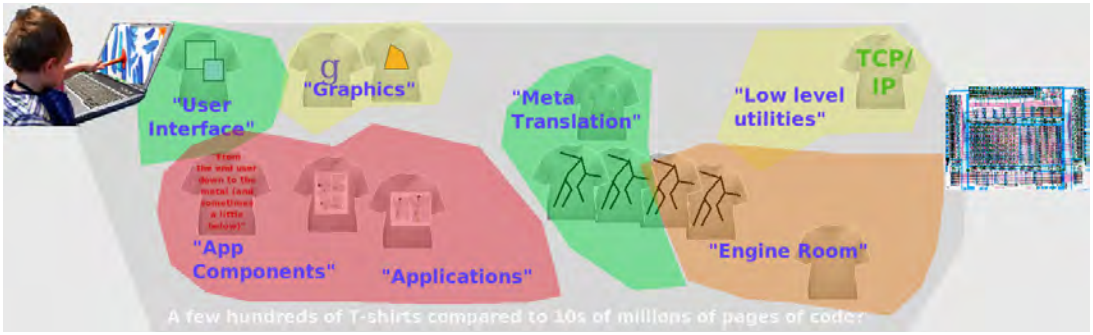
However, one of the early major parts of the system we wrote ran faster than anticipated, and this got us thinking about making it run reasonably on a conventional laptop. This would require some optimizations, but we thought we could find ways to do this without adding an enormous amount of code. This is how the last few versions of STEPS were built, and we have been counting the optimization code that we've had to add.

This "siren's song" (that is so well known to computerists) has created a project that is a bit different than the original proposal, and some of the differences and tradeoffs are detailed ahead.

General Approach—Look for *needs* and identify *kernels* that are close to *needs*, then *create languages* to implement the kernels. Finding kernels is heavily design intensive. Creating languages cleanly and relatively easily is one of the central *needs*, and so several of our kernels are aimed at easily creating new languages. "Powerful principle" heuristics are used to aid the design process. E.g. (a) "Math Wins!", (b) build throwaways to "find the arches", (c) "particles and fields", and (d) "Simulating *time*". And so forth.

"T-Shirt Programming"—One part of the "Math Wins!" principle is drawn from the aesthetic delight of Maxwell's equations on a t-shirt—the basic idea that it is often possible to find/invent mathematics to represent a model which brings all the important relationships into one cosmic eyeful, t-shirt size. This brings forth fanciful and motivational questions about domain areas, such as "How many t-shirts does it (should it) take to model this situation?" In the STEPS project, we ask that question about the whole enterprise, and about the parts of the system, for example "How many t-shirts will be required to define TCP/IP (about 3), or all the graphical operations needed for personal computing? (about 10)".

Overall Map Of STEPS—A rough map of STEPS can be shown in terms of t-shirts clustered near the end-user for the ones that deal with human interaction and end-user objects such as documents, near the CPU for the "engine-room" ones that make the executables for the CPU, and the systems t-shirts in between that support the system but are generally not visible.

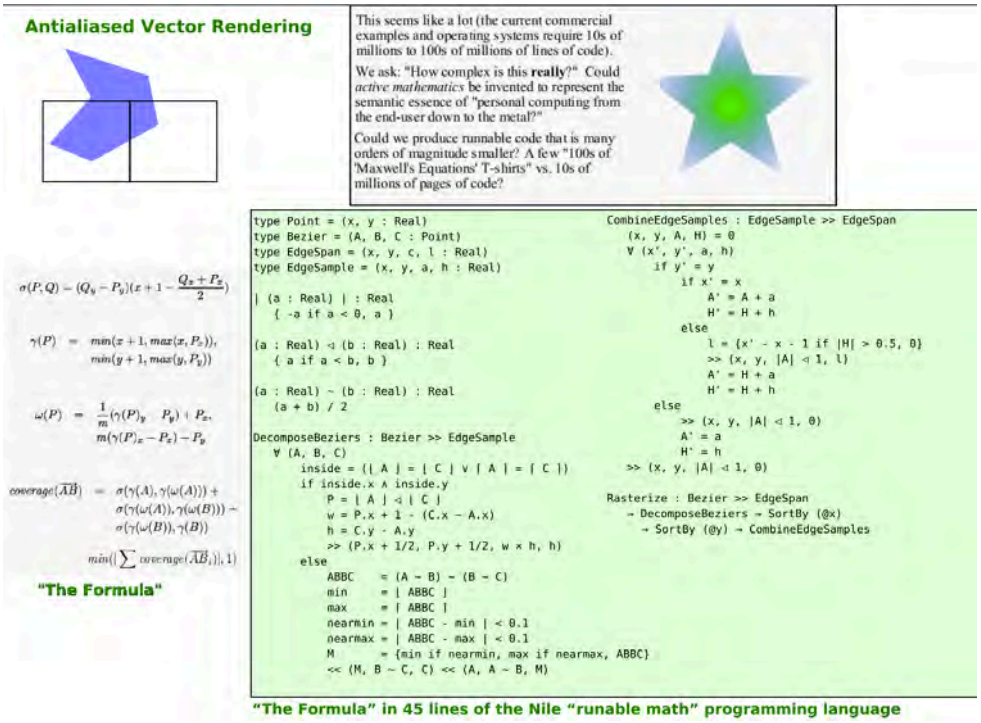


Overview Of General STEPS Results—Many different parts of personal computing were modeled in "runnable maths" and integrated into a workable "cute Frankenstein's monster" personal computing system—called "Frank"—which is complete enough—and speedy enough—to give all our presentations and

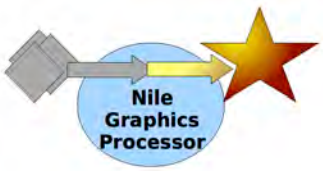
to write this report. By having universal documents and orthogonal fetch and storage modes, much of the functionality of the standard apps is covered very compactly. The universal document idea also allows the user interface to be modeled as a document, and also provides a much richer and simpler vehicle to superset the world wide web.

As expected, the approach of inventing runnable maths allows very compact and clear programs to be written. For example, the Nile Language creates standard graphics rendering, compositing, filtering, etc., which cover virtually all personal computing graphics functions in just 495 lines of code.

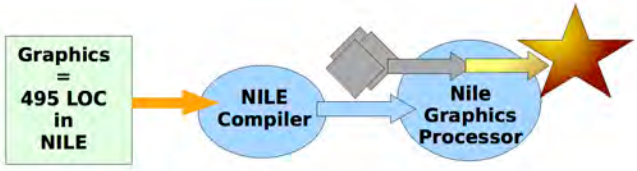
Below is an illustration of simple rendering and filling defined in Nile. The original mathematical formula is in the lower left, and the runnable math Nile language version in 45 lines of program code is to the right. Typical output of rendered text and a star filled with a gradient is shown at the top.



Making "Runnable Maths"— "Mathematics" is a plural form for "devising representations and inference schemes to help think and reckon with". Some useful forms of mathematics provide few hints for running them on a computer, while others can be run more or less directly, or after being transformed by analysis programs. For example, Nile rendering was originally conceived as a formula that could calculate the overlap of a polygon to a square (pixel). The Nile language was designed to be as close in spirit to this and other formulas of computer graphics as possible. To actually convert data into an image—for example, a description of a gradient-filled star into an image on the screen—an executable (**Nile Graphics Processor**) has to be made from the 495 lines of Nile code that define all graphics (lower left). This executable is made from another executable—**Nile Compiler**—which compiles the 495 lines of code. (below: human written code in green, executables in blue, data in gray, results in color.)



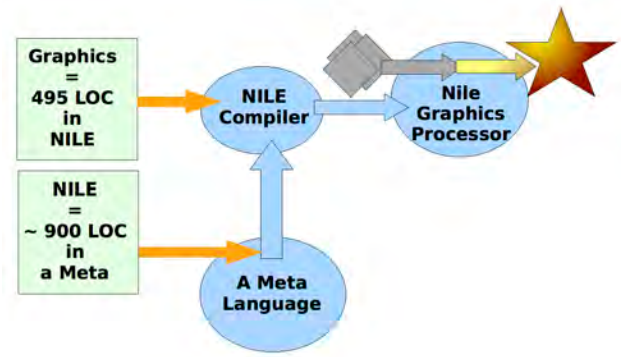
A rendered graphic image is made by executable code interpreting a description



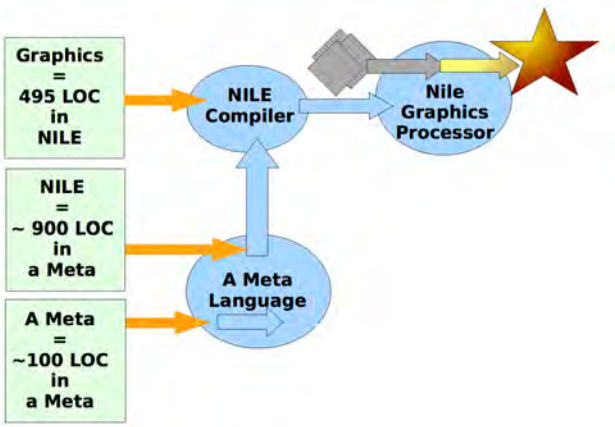
The graphics processor is made by an executable compiler that interprets a hand-written description of all of 2.5D computer graphics

However, we still have to make the **Nile Compiler** executable. This is made by sending a definition of the **Nile Language** and its meanings written in a **Metalanguage** through a **Metatranslator** executable (see below right). The form of the language requires about 130 lines of code to define it, and the meanings and optimizations about another 750.

This leaves the **Metatranslator** executable to be made—its job is to make Language Processors, so we will use it to make itself. This requires about 100 lines of definition code plus a variety of other modules (below right—described in more detail ahead). Over the life of the project, we have made and experimented with a number of metatranslators for this task.



The compiler for the Nile language is made by a "Meta" executable that interprets a description of the Nile language



The Meta executable can make itself when fed a description of itself

All the "runnable maths" languages are made in a similar fashion, though some of them generate virtual machine code instead of (as Nile does for speed) binary machine code directly for the local CPU. Ahead (page 10) is a more detailed description of the work in this area done in the last year.

Assessing STEPS—Since we are modeling personal computer systems as they are generally understood, our primary aim was not to improve existing designs either for the end-user or at the architectural level. This is partly because, in many respects, the closer STEPS appears to resemble existing systems, the easier it is to assess. But in practice, quite a bit of design improvement was needed to achieve better results of "smaller, more understandable, and pretty" (this is because some of the causes of the enormous code counts in standard systems come from poor design choices). Introducing improved designs creates "apples and oranges" problems for comparing what we've done to already existing systems. For example, it's crazy that "Office Apps" (including the Web) aren't just use-patterns manifested from a single "universal document type" with a spectrum of tools and a variety of ways to "fetch and store" them. By just cleaning this up we eliminate enormous amounts of special purpose code, and the need to write any of this ourselves.

One measure will be what has been achieved functionally within the initial target 20,000 lines of code budget. Another measure will be typical lines of code ratios compared to existing systems. We aim for large factors of 100, 1000, and more. How understandable is it? Are the designs and their code clear as well as small? Can the system be used as a live example of how to learn and do this kind of design and building? Is it clear enough to evoke other, better, approaches?

Note that a final result of 20,000± lines of code is still a 400 page book (at 50 lines per page), and, even if this is more than a 1000-fold reduction in size from standard systems, it is still not easily presentable in a report, or as the book itself. In practice the system is dynamically active with ways to examine the working processes and the code that gave rise to them. Some of the visualizations are quite striking (see ahead), and others could use much more work. This is a future research topic of great interest.

Summary of STEPS Research In 2012

The four main areas of work in 2012 were:

1. Inventing & building a language for STEPS "UI and applications", and rewriting Frank in it
2. Producing a workable "chain of meaning" that extends from the screen all the way to the CPU
3. Inventing methods that create automatic visualizers for languages created from metalanguages
4. Continuing the start of a deep investigation of "What vs How" programming via "Cooperating Languages and Solvers".

1. Inventing & building a language for STEPS "manipulables and applications", and rewriting the Frank UI and Universal Editor, etc. in it

The language is called KScript and the GUI framework is called KSWorld. The goal of KScript and KSWorld is to try to reduce the "accidental complexity" in GUI framework writing and application building. The aim was 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.

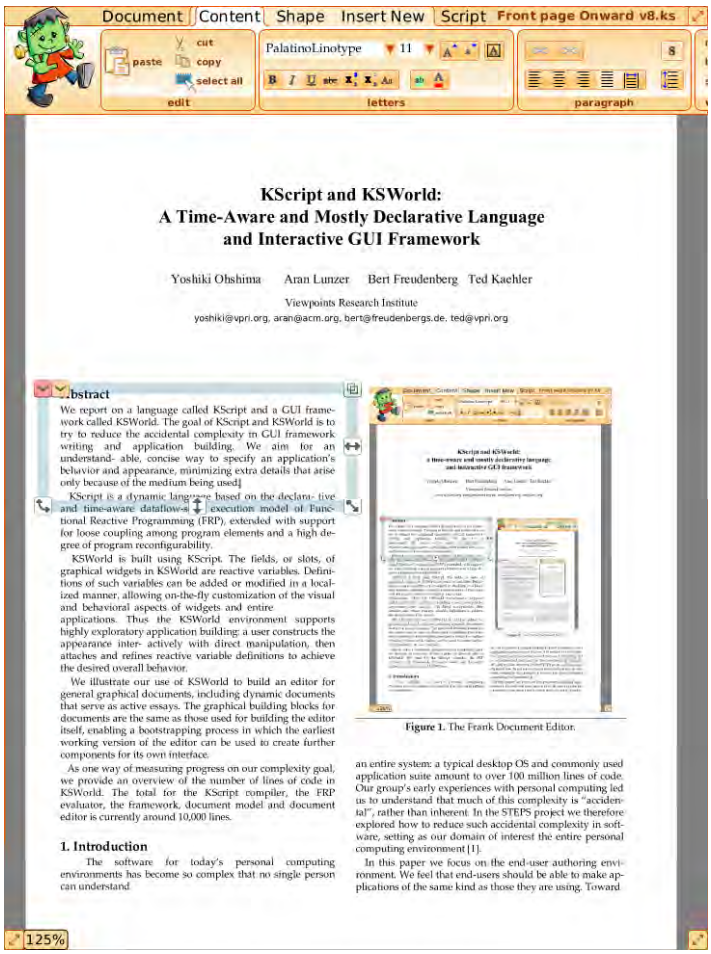
KScript is a dynamic language based on the declarative and time-aware dataflow-style execution model of Functional Reactive Programming (FRP) [FRP], extended with support for loose coupling among program elements and a high degree of program reconfigurability.

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 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.

This involved replacing 50,000+ lines of Smalltalk that had been used as a scaffold for the Frank system with about 10,000 lines of code in a newly invented language—KScript. KScript was partly inspired by Ivan Sutherland's *Sketchpad* in allowing programming to be done "by hand" assisted with dynamic relations. As with the Nile language for creating graphic rendering and composition processes, it had to be both as expressive as possible, and efficient enough to allow the newly remade Frank to again function as a useful system in real-time.

The three main categories that have to be handled well are:

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



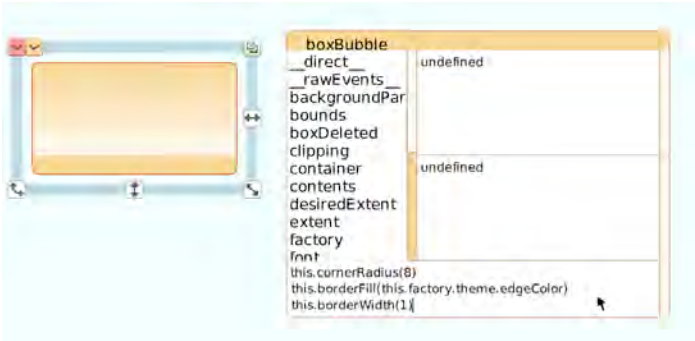
The first page of the summary paper of KScript and KSWorld in Frank. The paper is included as Appendix I to this report.

In the new system, the Frank user interface and document system are actually "made by hand" from a few graphics primitives with dynamic relations (see the summary papers in **Appendices I, II** (also [55] and [56]) for more details. Following is an example of how the "bubbles" in the user interface are made.

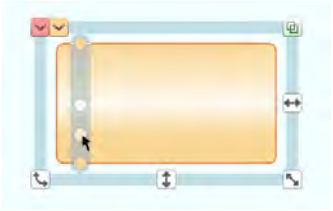
The first step is to create a Box to be the bubble, give it rounded corners and a border, and an appropriate gradient fill.

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:

Our goal: to make a box-editing bubble (here as it appears when no box is selected).

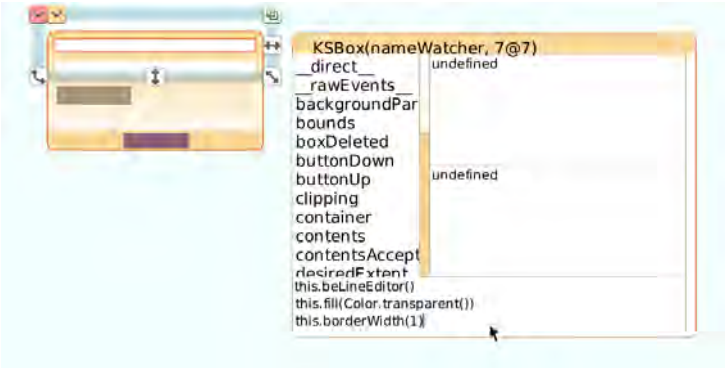


Rounding the corners, etc.



Using the halo's gradient tool

Then 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 to be a text editor

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
```

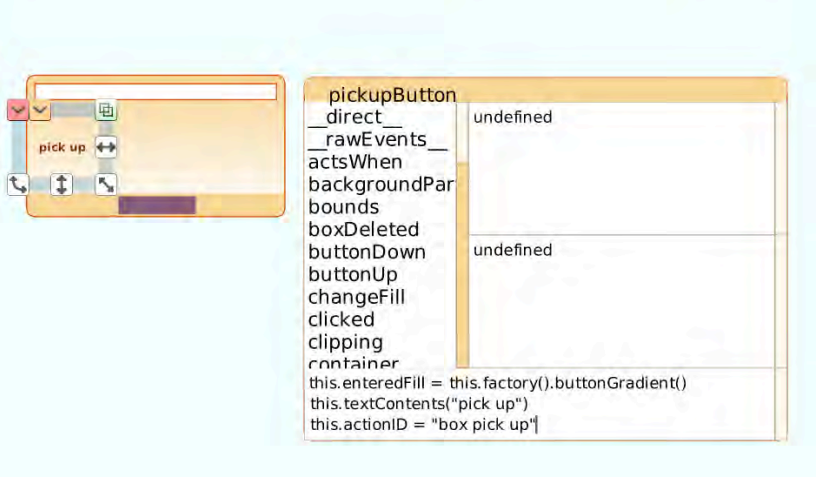
refers to the selected Box, and the result is converted to a string and shown in this Box.

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 entered-fill along with the button's label and action identifier:

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.



Setting up one of the bubble's command buttons

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.

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 (more details of breakdown in the Appendix).

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

KScript Lines of Code count for the essential part of implementing the Frank UI and Authoring System

We had anticipated that the "application parts" of STEPS would be the most difficult to fit into our initial lines-of-code targets, and this was indeed the case. This is despite the re-design of the productivity apps into one app with one kind of universal document that could serve all document purposes and an orthogonal approach to finding, fetching, storing and sending that subsumes the major modes required for docs, emails, the web, etc. The KScript renderings of the meanings of the applications and the user interface seem quite compact—they are pretty much what needs to be said—and we conclude that "removing the accidental" was done pretty thoroughly here, even though we wound up with more lines of code than we were hoping.

Two comprehensive recent papers about KScript, KSWorld, and applications building do a good job of covering this work in the last year. They have been included with this report as **Appendix I** and **Appendix II**.

2. Producing a workable "chain of meaning" that extends from the screen all the way to the CPU

This is yet another of many different low-end systems we've built during the STEPS project to investigate the best ways to deal with "life in the engine room". As before, we thought it very important (and illuminating) for this project to not use existing low-level tools—such as C, which often is quite large with fairly good optimizers—but to look at what can be done from scratch when dealing with the (often strange) architectural ideas of a CPU vendor (for example the Intel *86 line of CPUs). We like to include the tools code count in our totals, even though this is rarely done in industrial renderings of personal computing.

Our main test cases over the years have been to take the definition of "personal computing graphics" as written in less than 500 lines of Nile—and as used in the practical version of Frank—and invent transformations that reformulate the very high level descriptions into binary machine code formats that can run directly on CPU hardware (see the general summary on page 5). Nile is used for several main reasons: (a) it is small and very high level, (b) it is used to make all graphics processing in the Frank system which is used day to day, (c) it has to run quite efficiently to allow it to be put into practical use.

The main effort this past year has been to use the ultracompact Maru system (summarized in the sidebar and described in previous yearly reports) to make the complete meaning chain for Nile all the way to generating direct code to be executed by the CPU.

To give a picture of this, we will use the same form as the illustration on page 5 and show the Maru modules and code counts needed to produce the Nile system in Maru.

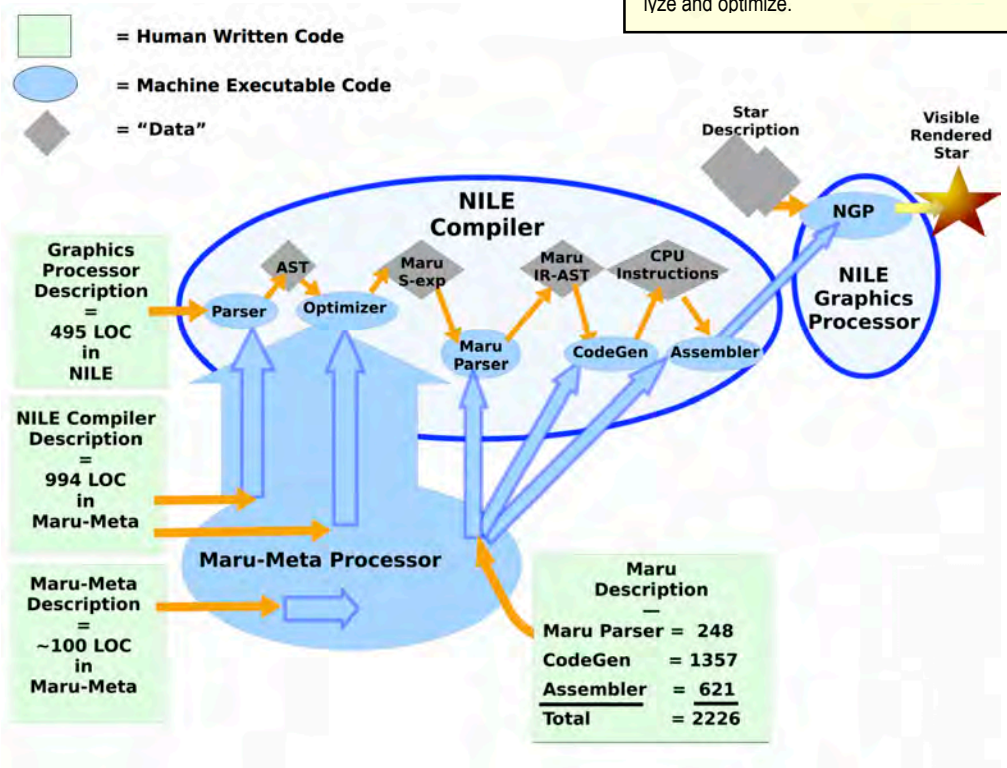
Maru is a vertically-extended description language that bridges the gap between desirability of high-level representations and easy manipulation of low-level details. It is simultaneously an extensible, high-level language for strategy and coordination, and an intermediate representation for programs and semantics from a wide range of high- and low-level languages.

Most critical is its metacircularity: it provides low-level mechanisms that can implement new high-level abstractions for describing new high- and low-level mechanisms.

High-level features include higher-order functions, extensible composition mechanisms and extensible data types. These are used throughout Maru's description of itself and to build mechanisms for compiling that description to an implementation and runtime support system.

Major components designed to be reusable include a PEG-based parser generator, an AST abstraction (supporting objects and primitive types, and operations on both), an evaluator for compile-time execution of ASTs (to analyze and manipulate ASTs or other data structures), and extensible mechanisms for code generation (including instruction selection and register allocation).

The latter phases provide several points of entry for programming language implementations. The compile-time AST evaluator provides hosted languages with a low-cost mechanism to provide their own metalinguistic and introspective facilities during compilation, aided again by the system's metacircularity: analyses and optimizations are described using the forms and processes that they analyze and optimize.



More details of how the executables used by Nile are made

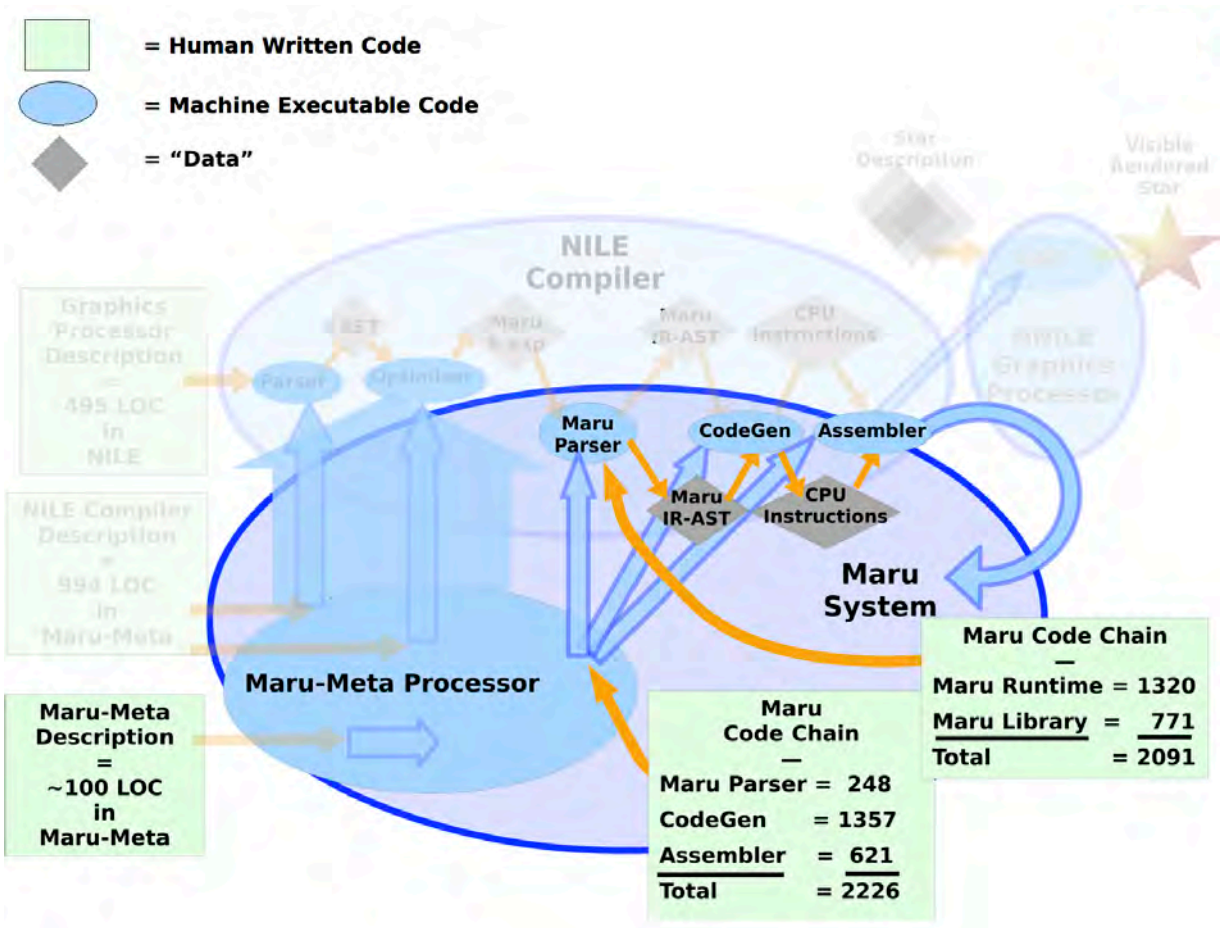
Above we see a schematic of the chain of transformations from the 495 lines of code that describe standard 2.5D personal computer graphics to eventually make executable machine code "NGP" in the upper left that can transform a description of a graphical object—in this case a star filled with a gradient—and put it in the display buffer for eventual display on the screen. The executable code segments that make up the Nile Compiler are made from code written in the Maru-Meta language—the Parser and the Optimizer—plus generally usable executables for Maru itself—Maru Parser, CodeGen, and Assembler—which are made from the Maru Description code.

The Maru-Meta Processor is made similarly (see below).

Below we dim out the Nile Compiler chain to more easily look at how Maru makes itself. This includes all that is necessary to make each of the specially designed languages that make up the STEPS system.

The basis here—which can be the foundation of each language—is about 4400 lines of hand-written code. This would be smaller by about 500 lines if we only included the machine instructions actually used for Maru and Nile: the current count is the code needed to assemble all the X86 instructions.

The Nile example shows that about 495+900+200 = 1585 lines of code were added to create the full range of 2.5D computer graphics plus the compiler for the language. Similar schemes for other languages have similar code counts. For example, the code count for the KScript compiler is about 753 lines. Other parts of KScript are more apple and orangy—see the "Applications in KScript" paper in **Appendix II**.

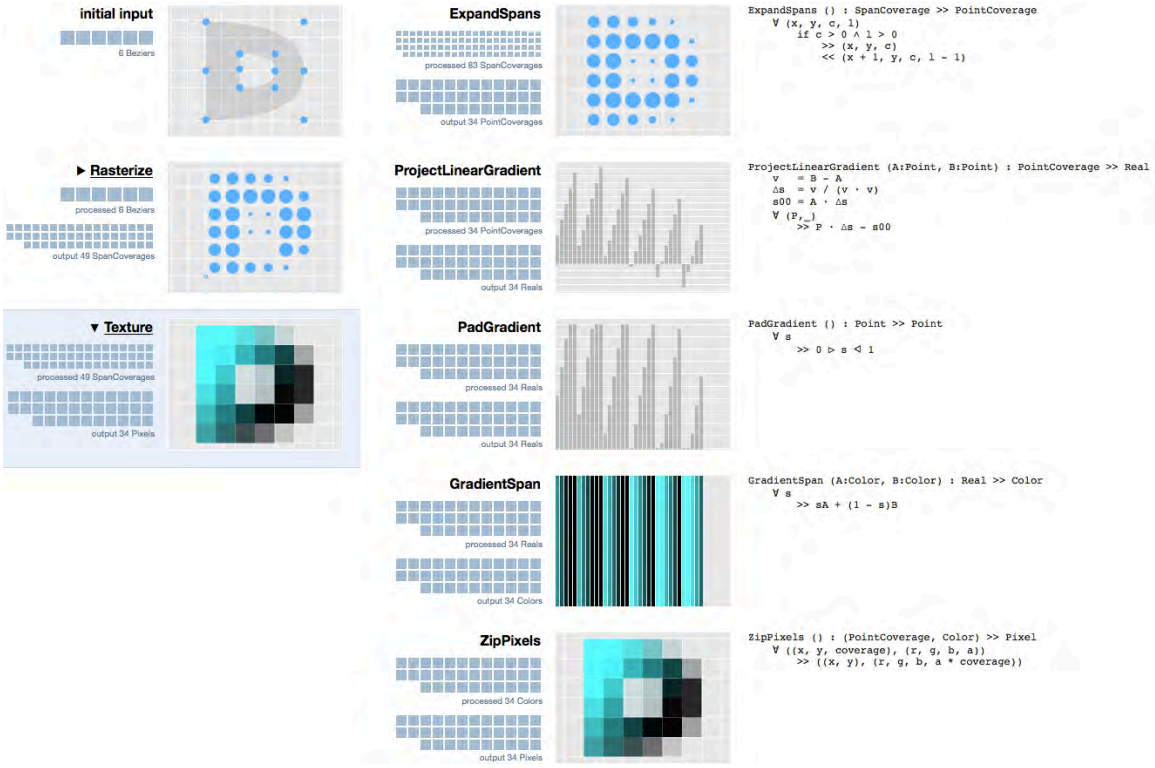


In the same system, suppressing the Nile details to show how the Maru foundation language and system is made from itself

3. Inventing methods that create automatic visualizers for languages created from metalanguages

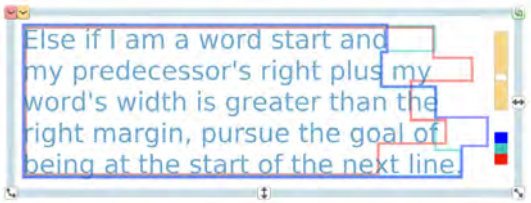
We addressed the issues surrounding the understanding of "quick tools and languages" built using metatechniques. For example, just as it is now much simpler and easier to create a running and workable language using metatools—and even do enough high level optimization to allow the new systems to be used in a practical manner—it should be possible to automatically (or semi-automatically) create visualizers of the processes that the new tools give rise to.

A good example is to again take the Nile language as a test case. The code for making graphical processing elements is short and simple, and consists of functional elements connected by dataflow streams. By looking at the (decorated) Abstract Parse Trees for Nile it is possible to derive useful viewers recursively through the code that dynamically relate actions both downstream and upstream to each other. One can place the cursor over any part of any visualization and the whole will respond to show how that part got made and what it affects.

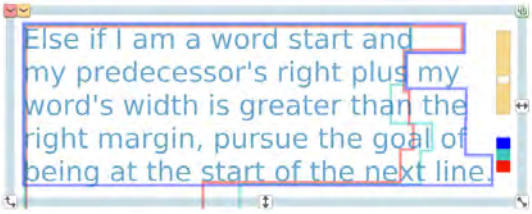


Automatically produced visualizer for the Nile programs that convert a Bezier curve (at top left) to shaded pixels (bottom of both columns). The right column is an expansion of the "Texture" code block.

Another viewing exploration of interest was to compute multiple "worlds of solutions" from slightly different constraints, and then to show them in a way that allows easy comparisons of the "goodness" of the solutions. A good example is to take the formatting of a text paragraph under different measures of "fittedness" and show the solutions superposed so the outlines of the layout solutions can be easily seen.



A text paragraph showing three simultaneous solutions (red, green, and blue outlines) under different constraints for a given right margin



The same text paragraph showing different solutions (red, green, and blue) for a different right margin

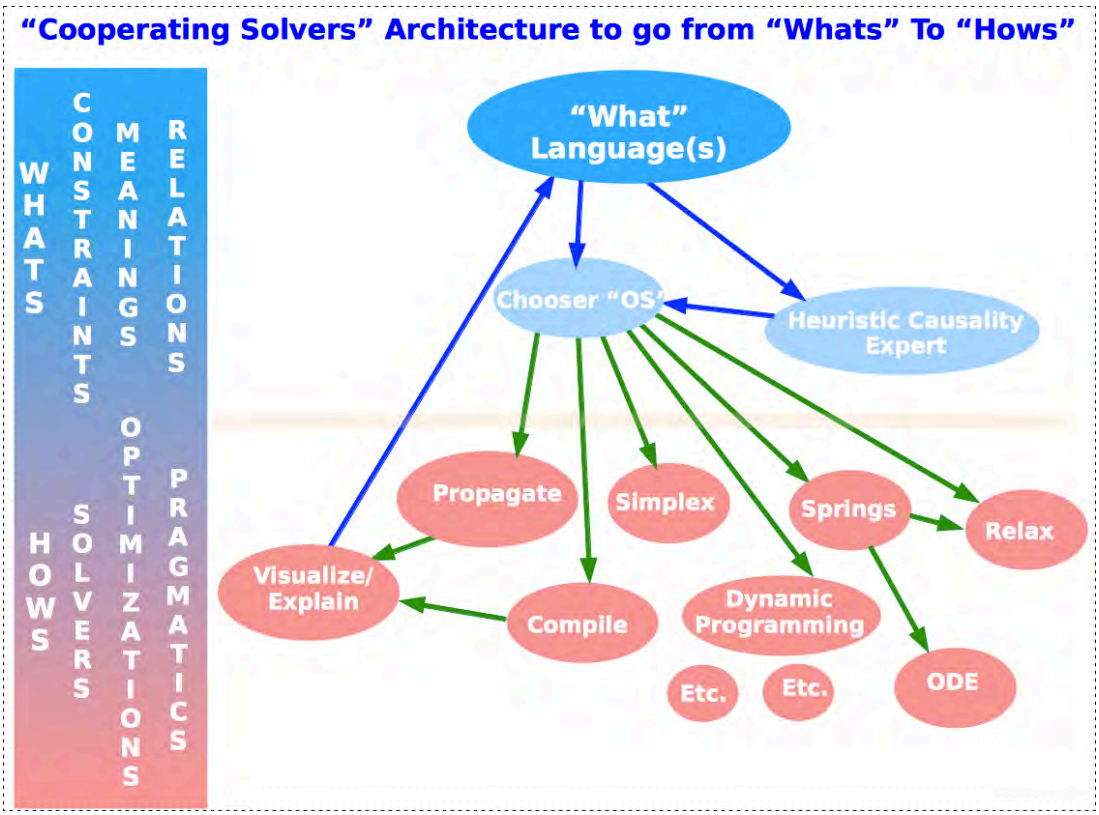
4. Continuing the deep investigation of "What vs How" programming via "Cooperating Languages and Solvers".

As discussed in previous sections of this report, one of our overall goals for the STEPS project is to have a very small kernel of code (20,000 lines) to capture the essentials of a complete personal computing system from the hardware up. To enable this, we take the position that:

•carefully crafted domain specific languages (also known as problem oriented languages) will be essential to capture the essentials of different aspects of the system's functionality, and

•these DSLs should be very high level, and in many cases declarative, focusing on goals and strategies rather than low-level details.

To support this, the various DSLs need a clean, well-defined way to interact with each other, one that admits easy integration of new DSLs as they are developed. Constraints and constraint solvers are one powerful way to support declarative, high-level DSLs. However, we believe that it is most unlikely that we will be able to find or devise a single powerful constraint solver that meets all our needs – rather, we will need multiple kinds of solvers, and a way for them to interoperate to solve systems of constraints. In the same vein, we want a way for different small languages to interoperate to build a larger system.



From "Whats" to "Hows". A good result would to have a single "specification/requirements" language at the top in the best form for human beings to use and reason in, served by many heuristic and pragmatic modules that can realize the meanings of the specifications and requirements, both for testing but also to generate practical implementations.

During the initial phase of this work, we devised an architecture for cooperating solvers that includes multiple levels of cooperation: at the highest level, making very minimal demands on the different solvers, but also allowing more tightly coupled sets of solvers to cooperate in richer ways on subparts of the problem. We represent the constraints and constrained variables as a bipartite graph, with constraints in one set and variables in the other, and the edges connecting variables with constraints on those variables. For the top level, we partition the constraints into regions. The different regions can share variables (but not constraints – each constraint belongs to exactly one region). The variables shared by

two or more regions must be read-only for all but one of the regions. (See reference [Cooperating Solvers] for more details.) Given a way to solve the constraints in each region, this architecture suggests an easy way to solve the constraints as a whole: it is simply a dataflow problem. Then, within a region, we can either use a single solver, or else have another layer of cooperating solvers that use a more sophisticated architecture to join them, for example by allowing constraints on the variable values (e.g., bounds) to be passed back and forth between the solvers.

We also experimented with integrating five different constraint solvers: Cassowary, an incremental solver for linear equality and inequality constraints that include both hard and soft constraints [Cassowary]; OCassowary, a wrapper for Cassowary supporting references to real-valued attributes of arbitrary objects; Kodkod, a relational first-order SAT-based constraint solver [Kodcod]; Z3, a SMT–SAT Modulo Theories–solver [Z3], and a naive depth-first search solver.

We soon evolved this into a framework for cooperating languages [Cooperating Languages], generalizing from the framework for cooperating solvers. This supported a program as being composed from several sub-programs written in multiple languages, e.g., an imperative part, a linear constraint solving part, and a first order constraint solving part. We also experimented with several small languages within this framework, including a language for "Things" (constraint-based declarative descriptions of geometric objects, electrical circuit components, and the like, derived from the examples in ThingLab [ThingLab]), a language for springs and struts, and several others.

As the work progressed, we realized that Datalog, and specifically Dedalus, a subset of Datalog with an explicit representation of time [Dedalus], could provide a clean, flexible way to specify the communication among solvers and also among DSLs. We are currently reworking and significantly extending our first efforts, using this new framework, and at the same time developing a set of compelling applications to drive the work. We have implemented a first example of two constraint solvers (Cassowary and a solver for uninterpreted functions), running in different processes, cooperating via Bloom [Bloom], a practical implementation of the Dedalus semantics as a DSL in Ruby. There are a set of reasons that supporting distributed solvers is likely to be highly beneficial:

- speeding up finding solutions by running on multiple machines
- using a server with a specialized constraint solver on it
- addressing security issues in distributed systems by only sharing a subset of the information among machines (as is done in the University of Pennsylvania work on Cologne [Distributed Constraint Optimization])

We have also designed an implementation of Bloom in the Squeak language, which will fully interoperate with the Bloom/Ruby implementation. Bloom will thus handle all of the distribution, communication, and coordination among the programs in the different languages. We plan to implement this in the next months; and after that to integrate the Functional Reactive Programming model from KScript with it.

Reflections on the just ended NSF part of the STEPS project

The STEPS project was funded by NSF and several other funders. This report marks the end of the NSF part of the funding, but it is presented as a yearly report because the STEPS project itself continues. However, the previous six years of the project has produced much to reflect on.

STEPS Overall Finding—Enough was done in the many areas proposed to conclude that much of today's personal computing systems—and many other kinds of programming goals—can be built with just 1000s to tens of 1000s of lines of code using new designs and languages that run those designs. By any measure, this is many orders of magnitude smaller than the targets we examined. This was a highly design intensive process the first time around, but yielded many techniques and tools that can be used for more easily making similar and other systems.

The results included several working interactive systems that covered a wide range of "standard" personal computing.

We were correct in thinking that a lot could be done within our guessed code budget of 20,000 lines for the *meaning* of the entire system. However we wound up covering a bit less ground that we had expected, partly because we got interested in seeing if we could run in real-time on laptops, and this part of the design and optimization fought some of the original goals.

The Centrality Of Design—It was not a surprise that STEPS has been a heavily design intensive project—the NSF part was funded as part of a "Science of Design" inquiry—and "design" is a wide word that can be used for many different kinds of processes, from those that involve mostly organization and reorganization of existing materials, to the invention of brand new materials when that is needed. Still, we were a bit surprised at how effective was "simple design": just taking fresh looks at what "machineries" are actually needed to do the many parts and tasks of personal computing made a tremendous difference to the size and real complexity of the entire system. It is also not surprising that the language designs we judged to be the best were the ones which followed the most design, implementation, and redesign.

As anticipated, the two most difficult parts of the project (in terms of lines of code required and number of redesigns and rewrites) revolved around (a) applications and their frameworks, and (b) the "engine room" and how best to harness the CPUs available. Our change of goal to see if we could get real-time performance and still have a tiny code budget on a laptop made both of these even more of a design task. The biggest change in (a) from the original scheme was to come up with a language for dealing with "control and features" that was not just compact but could also make all the dependencies run in real-time. In (b) the trickiest cases were ones that could not be handled by an interpreter, but—e.g. for graphics rendering—had to generate decent enough machine code and a runtime without requiring a lot of code for various kinds of optimizations.

Other parts of the project—especially the ones that were "more like math"—required a lot of thinking but behaved well when implemented. These included the Nile graphics system, TCP/IP, and the implementation of the various languages and metalanguages we used.

In parallel, the best explanatory visualizations we came up with—for example, see page 12—were rather like the sketches one might make when trying to carry out good design in the first place—and this raised the question: in what order things should things really be carried out? E.g. perhaps a better way to do a project like this (and any complex software project) would be to start by making a "design studio" IDE that could help designers think through designs and then gradually develop the detailed pragmatics needed. This got us to think more deeply about "sketching and constraining" as a good process for finding the design and the problems that needed to be solved (this in fact was just how Sketchpad—the original personal computer environment—was set up).

In the middle of the project we were very happy to find the "Berkeley Orders Of Magnitude" (BOOM) project, which had goals very similar to ours, but had decided to look at different application areas (many of them having to do with distributed computing on the Internet). We were interested to see the parallels in how they approached solving their problems and were encouraged at some of the similarities. For example, they did the right thing (in our opinion) by making a system that would do some of their goals,

and then analyzing the system to come up with a Domain Specific Language that would allow the system to be rewritten very compactly. The main language design of theirs—Dedalus—also incorporated a strict internal model of time—an approach we think is absolutely necessary for scaling and concurrency.

STEPS Production Summary—We have been able to count more than 60 languages that we implemented as part of our learning process—perhaps 20 of these have been discussed in the STEPS reports. Some of the 60+ were existing language designs—such as Prolog, Javascript, Logo, etc.—but most were home-grown designs aimed at a wide variety of areas of need, including: making graphics, documents, user interfaces, TCP/IPs, languages themselves, fast runtimes, etc. Most of the invented languages we made were tested to some extent by writing example systems in them, and some of these were pressed into major service to construct the "Frank" systems that were put into daily use as personal computer systems.

Similarly, the several Frank systems were reimplemented a number of times, and some of the Frank parts (such as the universal document system) were also heavily reimplemented.

One kind of multiple implementation involved targeting a number of back-ends—for example: Nothing, Maru, C, Smalltalk, JavaScript—for various kinds of convenience in running and testing.

Most of the parsing systems we made used some form of PEG parsing, and these were relatively easy to make and to deal with. It will be no surprise that it was generally much easier to implement a language than to design it (see the section below comparing "Computer Science" to "Software Engineering"), and that quite a bit of the overall work in making languages is not just dominated by design, but by optimizations, especially when real-time performance is needed. In our original plans we thought about alleviating optimizations by (a) running on supercomputers, and/or (b) making custom hardware (especially from the fruitful FPGA technologies of the current day). In the actual project we did neither. Instead we created software architectures—for example the multiple core/multiple thread architecture for Nile, and the multiple coordinated stream architecture for KScript—that were speedy enough for our purposes.

Because we counted the code to make the languages and their runtimes, a useful motivation was provided to keep the language specifications clean and small. We conclude here that if a project like this were to be done again by the same rather small number of people, then it would be well worthwhile to employ some form of the original supercomputer strategy.

"Computer Science" and "Software Engineering"—As mentioned in the beginning, this was a "science project"—we tried to find good models from phenomena—and a "math project"—we had to do a lot of "maths inventions" to find suitable forms in which to express the good models. The combination of these two took the project methodology out of the bounds of what could be hoped for as an improvement for current "software engineering". If there were an existing wide-spectrum "maths language" (which paralleled the existing framework of differential equations, tensor analysis, etc. that is used in physics), then a methodology for software engineers could be set up as a new kind of design process (somewhat analogous to "patterns" in SE today). We could think of this as a "Programmatica" (as an analogy to "Mathematica".

Today's use of "pattern languages" in programming can be criticized as codifying software methods that are still too limited. To make a methodology such as "patterns" work, we need to identify much stronger ways to build software and then help software developers by making these into a suite of heuristic approaches that has as much strength of application as possible.

We conclude that the "special language for each application area" is a powerful way to make progress in design, but that—in the end—we would prefer to distill the most important ideas into a more general specification language. This led to a renewed effort to try to make some progress with the "Whats" to "Hows" pyramid—this still seems like a very good way to make progress.

Differently Than Planned—The depth and length of this project gave rise to new ideas and goals as we proceeded. The original plan was to "capture meaning" in a way that could be separately optimized to run in real-time (or would run in real-time on a sufficiently powerful supercomputer). However, the first formulations of the new approach to graphics rendering ran surprisingly well on standard laptops and

we gave into the temptation of doing a few optimizations that resulted in the graphics model running fast enough to build a real-time system on ordinary computer systems. This injected interesting "realities" into what started as a more theoretical plan of attack. The universal media structures built on top of the graphics engine also were made to serve the dual role of exhibiting tiny clear models of their processes and to run well enough to be usable. As a result we wound up with a different artifact than was originally planned.

Another change of emphasis came from getting fascinated with a number of the "powerful principles" that were used as strategic organizers for the design and taking them beyond what was needed for vanilla personal computing—for example, the "Worlds" mechanism for fine grain context capturing, parallel experiments and redoing. This led to many experiments in "possible worlds reasoning" and visualizations of multiple solution paths.

The largest shift in emphasis came in the last 18 months of the project: the desire to approach the entire system "a la Sketchpad" as problems posed to a wide variety of "cooperating solvers" organized by a knowledge based system with expertise in both programming and solving. Fortunately for progress, we decided not to abandon the results from the previous years, but to use them as target examples and goals for the solvers to do as well. This change effectively produced parallel paths of effort in the STEPS project, and the new direction will be continued onward after the NSF part of this project ends.

More Than Planned—The comprehensive "Worlds" mechanism for fine-grain "possible worlds" computing has already been mentioned. Another area where a wider scope was adopted and a lot more work was done that originally planned was in the pragmatics of the graphics system. This included efficient chains of meaning, very fast execution, comprehensive experiments with dataflow and multiple cores/tasks, several elaborate visualizers, etc.

Less Than Planned—Several important goals where we did less than originally planned, included: (a) the comprehensive "bottom engine room" for the entire system, (b) numerous features of the productivity "suite", and (c) the system as an "exploratorium".

(a) We had guessed that the "bottom engine room" would be challenging if the lines of code count were to be kept small. But much can be done via minimal interpreters and other devices (such as the techniques often used in FORTH bootstraps). Still, a major concern was just how big this and its associated helper packages (garbage collector, dynamic linker, debugging support, etc.) would turn out. Then we compounded the constraints by deciding in mid-project that it would be fun to run as much of STEPS as possible in real-time on standard laptop computers. This led to many designs and trial implementations of parts of the engine room. And this in turn led to a primary focus on creating a "chain of meaning" for the code that had to run most rapidly e.g. convert the highly abstract, compact and mathematical Nile programs into a runable state that would manifest real-time rendering. With a different set of goals and more time, we probably would have tried a "metatracing" approach to optimization similar to that done in PyPy [PyPy]

(b) We did enough to show how "universal" documents combined with a wide variety of storage and transmission mechanisms could subsume word processing, desktop publishing, presentations, email, web-pages, and Hypercard-like pages. We mainly fleshed out the desktop publishing and presentation aspects of this, and (for example) did little more than to show how much more comprehensive versions of email, web-pages, etc. could be handled.

(c) Exploration was planned to be a major part of the STEPS system. We did make it explorable—the system is dynamic and has many kinds of "inspectors"—for the structures and behaviors of the components. In several cases—for example, the Nile Explorer (see page 12)—a fleshed out interactive viewer was made that provides great insights into what a complex running system is doing. Similarly, we also produced several "active essays" about some of the programs and facilities for their (re)construction by interested end-users (e.g., see **Appendix I** in [STEPS 2011 Progress Report]). Many other visualization experiments were done. But we had to leave a comprehensive high quality exploration of the entire system for another time

and another project.

A Significant Problem Still To Be Solved—Massively scalable intermodule coordination and communication has not been achieved via any means in personal or any other kind of computing. Thus in one sense—we are looking at typical personal computing systems for phenomena to model—this is out of the scope of the STEPS project and its proposal. On the other hand, we had expectations that a real advance could be made in this more than 50 year old problem which has just gotten disastrously more serious as the scope and size of software systems has been enabled to grow (a) via Moore's Law, and (b) by the super-scaling of hardware systems via the Internet. This work—along with the "Whats" to "Hows" project—will be the two main projects carried forward in the next years of the STEPS Project.

References

[Bloom] Peter Alvaro, Neil Conway, Joseph Hellerstein, and William Marczak. Consistency Analysis in Bloom: A Calm and Collected Approach. In Proceedings 5th Biennial Conference on Innovative Data Systems Research, pages 249–260, 2011.

[Cassowary] Greg J. Badros, Alan Borning, and Peter J. Stuckey, "The Cassowary Linear Arithmetic Constraint Solving Algorithm," ACM Transactions on Computer Human Interaction, Vol. 8 No. 4, December 2001, pages 267-306.

[Cooperating Solvers] Alan Borning, "Architectures for Cooperating Constraint Solvers", VPRI Memo M-2012-003, 18 March 2012.

[Cooperating Languages] Hesam Samini, "Architectures for Cooperating Languages", VPRI Memo M-2012-001, 28 March 2012.

[Dedalus] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears, "Dedalus: Datalog in Time and Space," Datalog Reloaded, Lecture Notes in Computer Science, Vol. 6702, 2011, pages 262-281.

[Distributed Constraint Optimization] Changbin Liu, Lu Ren, Boon Thau Loo, Yun Mao, and Prithwish Basu. Cologne: A Declarative Distributed Constraint Optimization Platform. Proceedings of the VLDB Endowment, 5(8):752–763, 2012.

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

[Kodcod] Emina Torlak and Daniel Jackson, "Kodkod: A Relational Model Finder," Lecture Notes in Computer Science, Vol. 4424, 2007, pages 632-647.

[NSF] STEPS Proposal to NSF - 2006

[PyPy] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09). ACM, New York, NY, USA, 18-25.

[STEPS Reports]

- STEPS 2007 Progress Report
- STEPS 2008 Progress Report
- STEPS 2009 Progress Report
- STEPS 2010 Progress Report
- STEPS 2011 Progress Report

[ThingLab] Alan Borning, "The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory," ACM Transactions on Programming Languages and Systems, Vol. 3 No. 4 (October 1981), pages 353-387.

[Z3] Leonardo de Moura and Nikolaj Bjørner, "Z3: An Efficient SMT Solver," Lecture Notes in Computer Science Vol. 4963, 2008, pages 337-340.

Appendix I

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 writing 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 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 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.

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.

1. Introduction

The software for today's personal computing environments has become so complex that no single person can understand

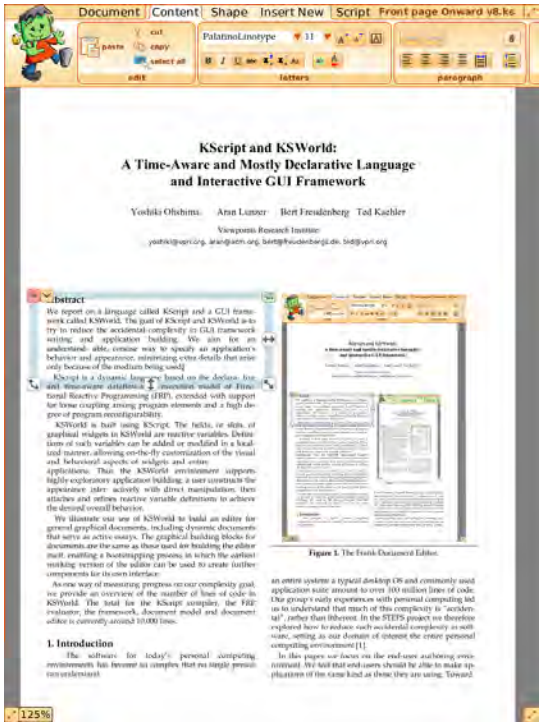


Figure 1. The Frank Document Editor.

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. Analogically, 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 is for reacting to changes in objects and dealing with 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 project's 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 fully takes 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 lets the user make various kinds 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 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:

```
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:

```
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

```
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 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. 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 transformation 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.

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. 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.

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-style reactive programming. The ideas on how to structure graphical objects are drawn from Morphic [10], Lessphic [11], and Tweak [12]. 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

We fully embrace vector graphics. The `shape` property of a `Box` holds an object that packages quadratic Bézier contour definitions along with fill and stroke data.

From the STEPS project, there is a full-featured vector graphics engine called Gezira [13]. We 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 the external device, the framework 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 framework is mostly written in a procedural rather than reactive style; event routing is an example of this. The basic 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 frontmost `Box` containing the event location should react. Trying 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`.

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 argu-

[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

ment 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.

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 [14]. 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`. 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()
```

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

```
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'
```

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 describing a button 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.

Figure 4. The File List.

6.1 File list

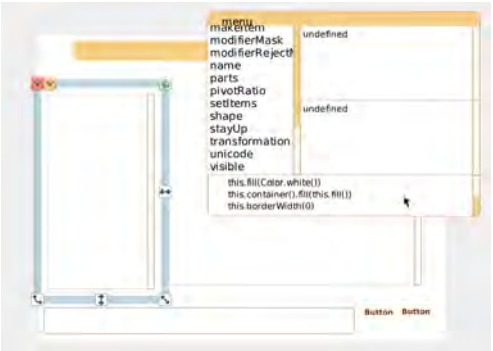
The first tool we are going to make is the File List, shown in Figure 4. In a File List there is a list of directories, a list of files in the selected directory, and a field to show the selected directory and file. Pressing the Accept button will trigger some action, while pressing the Cancel button 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:



Set fills.

At this point we can also use the Inspector to attach certain behaviors to `Box`es 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.

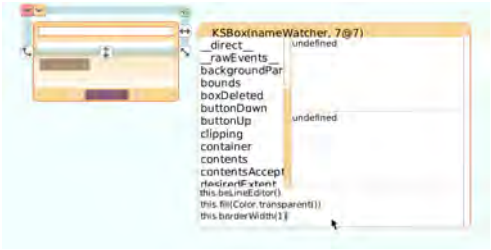
6.2 A panel for the tool bar

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, 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 the `selectedDocBox`).

The panel contains a number of buttons, making it conceptually similar to the way we defined a menu. Like 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 [15]. 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).

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` 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 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 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()
```

7.3 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.

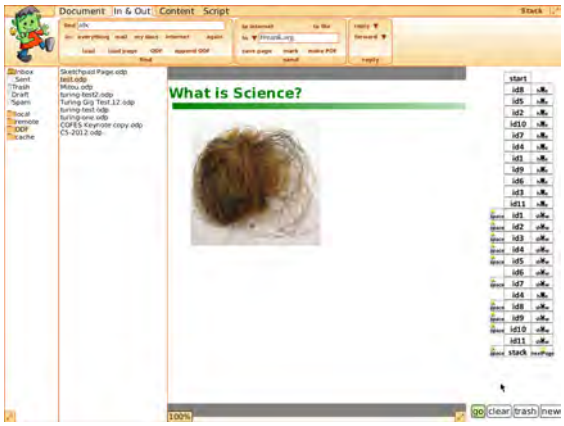


Figure 5. The Directory Browser on the left, and the Scripting Pane on the right.

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 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).

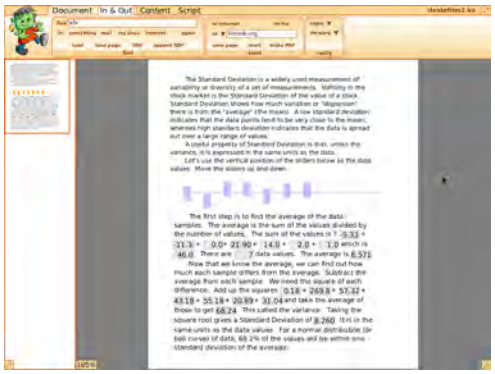


Figure 6. An Active Essay on Standard Deviation.

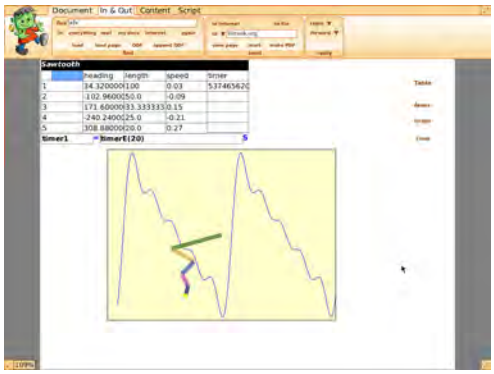


Figure 7. An Active Essay on The Fourier Series.

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 liberated 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, but here we would like to make a few remarks:

First, note 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 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	2,330	Gezira Bindings
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.

The lines of code is a metric of static complexity. But how about dynamic complexity? When an empty Document Editor 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.

9.1 Performance

Even though our work emphasizes cleanness 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.

There are two major aspects that 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 aspect is 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 [16]. Recently, Dedalus [17] 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 [18] 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 [19], 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 question is finding a good trade-off between expressiveness and simplicity.)

Animus, by Duisberg and Borning [20], 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.

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

[10] 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.

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

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

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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)
```

Appendix II

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.

Figure 1. The Frank Document Editor.

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 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 `Boxes`. Traditionally this tends to involve writing code that instantiates `Boxes`, 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 `Boxes` that represent the characters in a one-line text-field `Box`.

Conceptually, all we want to do is to make a handful of `Boxes`, 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 `Boxes`. We provide a simple constraint-based layout mechanism (called `KSSimpleLayout`), along with vertical and horizontal layouts, to specify the locations and sizes of `Boxes`. 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 `Boxes` 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 "offside 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.

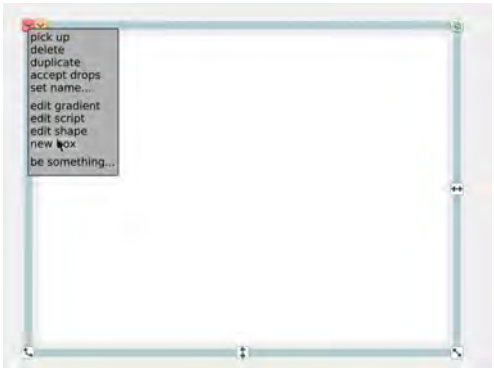
Figure 4. The File List.

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.

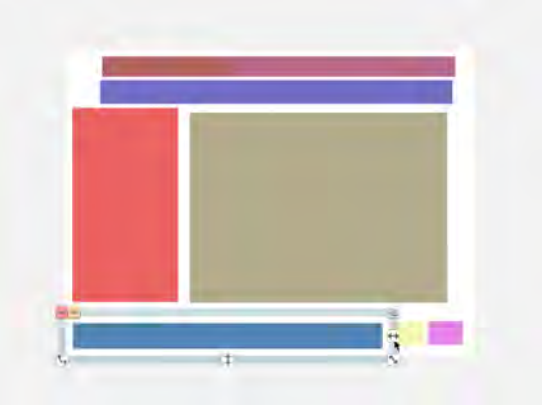
We start from an empty `Box`, then use its halo menu to add child `Boxes` within it:



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

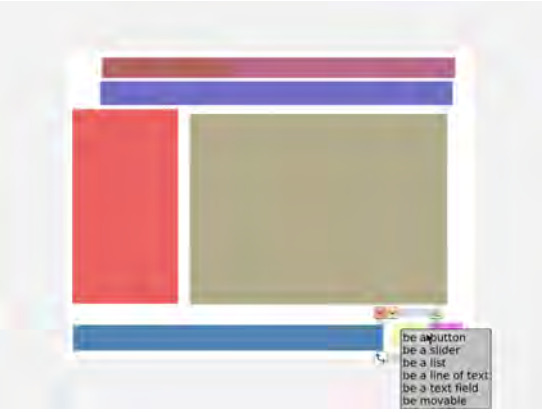
In all, we need eight such `Boxes`. We use the halo to resize and roughly place each one to get a feel for the eventual layout. Note that KSWorld initializes each new `Box` with

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 `Box`es 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:

Make each list `Box` scrollable.

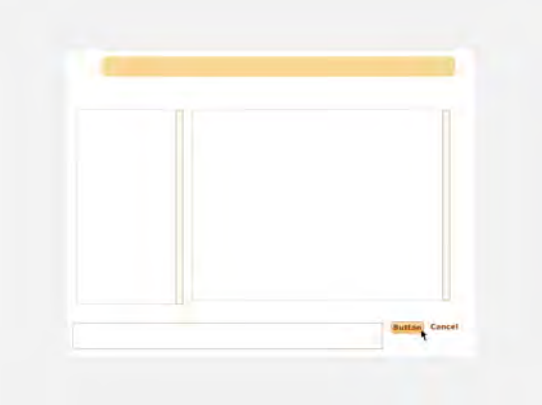
The Inspector can also be used to set all the `Box`es' fills and border styles as desired:

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:

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.

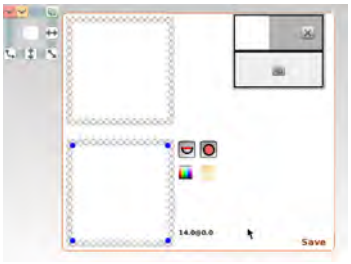
The Resize icon.

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

Invoking the Shape Editor.

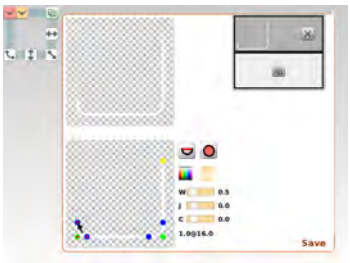
This opens up an editor for the Shape of this Box. A Shape comprises multiple paths, that can be considered as layers. In the editor, the layers appear in a scrollable list on the right, the currently selected layer in an edit view at bottom left, and the composite view of all layers at top left. In 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.



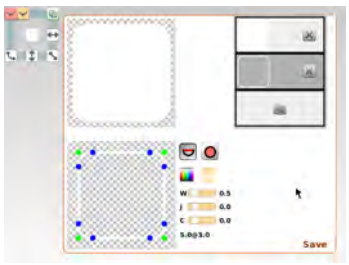
Initial shape, with a single filled path.

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.



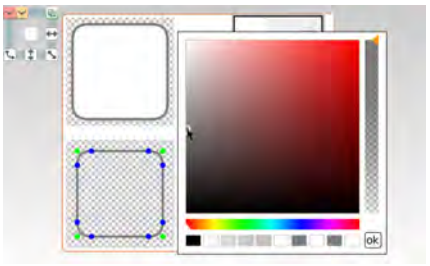
Rebuild the path with rounded corners.

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.



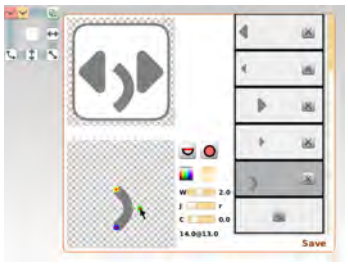
Duplicate the layer, and fill one copy.

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.

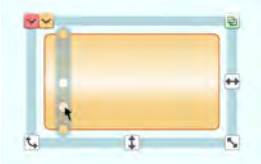
Recovering from this distraction, we move the path segment into its proper position and press the save button to store the completed composition as the new `Shape` for the original white `Box`. We can then save the `Box` to a file for later use.

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.

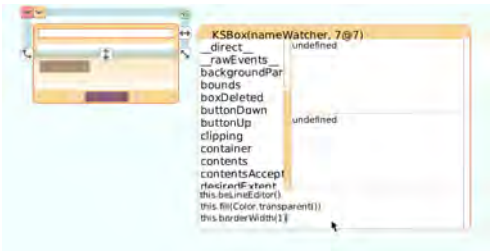
A box-editing bubble, as it appears when no box is selected.

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:



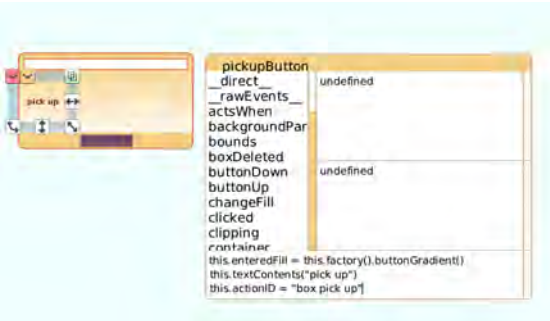
Customizing a part within the bubble.

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:



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

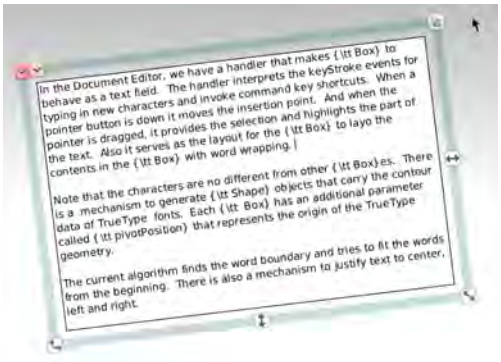


Figure 5. A Text Field. Each letter is a Box with vector graphics data representing the contour of a glyph.

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-

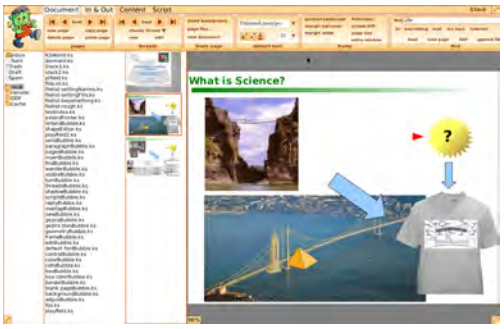


Figure 6. The Directory Browser on the left.

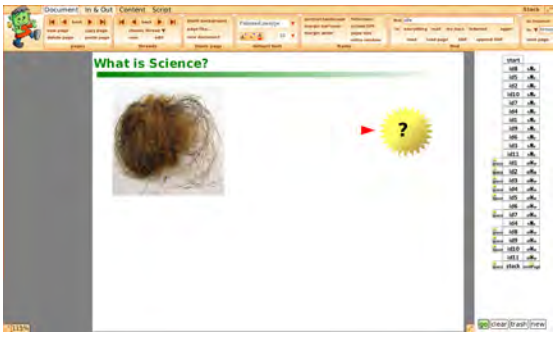


Figure 7. The Tile Scripting area on the right.

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)
```