The Lively Kernel
A Self-Supporting System on a Web Page

by
Daniel Ingalls, Krzysztof Palacz, and Stephen Uhler
Sun Microsystems Laboratories, Menlo Park, CA
and
Antero Taivalsaari and Tommi Mikkonen
Sun Microsystems Laboratories, Tampere, Finland

E-Mail: Lively@Sun.com

Abstract:
The Lively Kernel is a complete platform for Web programming written in JavaScript using graphics available in leading browsers. A widget set built from these elements provides a user interface kit, and the widget set is also extensible. A window-based IDE allows users to edit their applications and even the system itself. When a user visits the Lively Kernel page,

http://research.sun.com/projects/lively/index.xhtml

the kernel loads and runs with no installation whatsoever. The user can immediately construct new objects or applications and manipulate the environment.

The Lively Kernel is able to save its creations, and even clone itself, onto Web pages. In so doing, it defines a new form of dynamic content on the Web. Moreover, since it can run in today's browsers, it promises that wherever there is the Internet, there can be authoring of Web content.

Beyond its utility, the simplicity and completeness of the Lively Kernel make it a practical benchmark of system complexity, and a flexible laboratory for exploring new approaches to security, simplified graphics, and Web technologies in general.

Keywords:
Dynamic language, JavaScript, Morphic, self-supporting, Web programming, rich internet applications, widgets, Web 2.0

Note to Readers:
As of this writing, the Lively Kernel runs with no installation in the Firefox 3 beta and Safari 3 browsers. We are preparing an applet that will allow it to run in other browsers until their internal graphics are adequate for install-free operation.

I. Time for a Change

There is no good reason for Web Programming to be more complicated, less general, or any less fun than other modes of programming. There are reasons, of course, mainly focusing on static content and markup languages and ignoring several decades of experience with lean computing kernels built around Lisp, Smalltalk, and other dynamic languages. That is history, but it need not hold us back. In this paper we describe a simple and general kernel for programming the Web. Its core is less than 10,000 lines of code (with comments), it runs in major browsers with no installation, and it performs well. We call it the Lively Kernel.

In this paper, we begin by the observation that, completely apart form the text-based world of HTML and its decorations, the now ubiquitous Internet browsers provide all that is needed for a rebirth of active objects in the Web context. Beginning with the JavaScript language and standard browser graphics, we trace trace the construction of a computing environment from basic shapes to widgets (active user interface components) to programming tools, ending with an environment is self-supporting and that supports general application development and deployment on the Internet.

Look at a typical Web page on a typical computer and you will see static graphics, most likely generated from a decades-old markup language, being presented by a computer capable of executing a billion instructions per second. There is something wrong with that picture. There is no reason that the entire page cannot be an active object, ready to respond in all the general ways that computers were built to support. This is the Lively Kernel view of the Web and Web programming. Ironically it is not even a new approach, but rather the tried and true approach of numerous dynamic programming environments that were in widespread use long before HTML was adopted as the standard of Web content presentation.

We observe that every browser supports a dynamic programming language, one or more graphics systems, and support for network communication. While JavaScript has been mainly shaped by its role as a scripting vehicle for HTML, it is actually a perfectly usable dynamic programming language. In the area of graphics, most browsers support HTML, a flat graphics model (Canvas) and a retained graphics model (SVG; see http://www.w3.org/TR/SVG11/). For communication, modern browsers offer XmlHttpRequest for access to remote hosts elsewhere on the Internet. To a self-supporting system builder, this is all one needs.

We inherit from the World Wide Web an architecture built around a text markup language. The Lively Kernel sets that architecture aside in favor of modern graphics and a dynamic programming language. We begin by turning the conventional Web programming "stack" upside down as shown in figure 1.

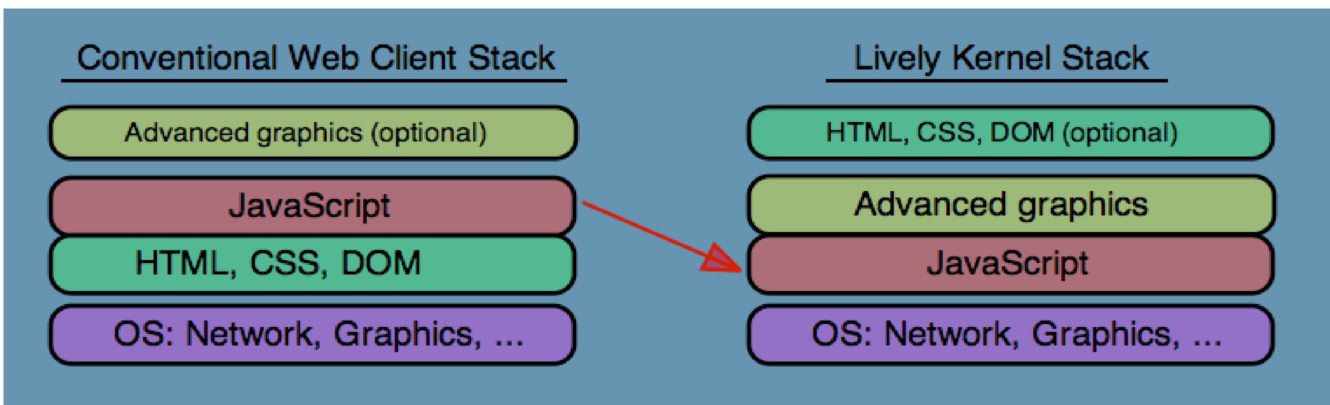


Figure 1: Turning Web programming upside down

The first priority of this architecture is to provide a world of active objects. This is accomplished by putting a dynamic language close to the operating system, which allows both the infrastructure (widgets, etc.) and the application to share the same pervasive generality and power. The compactness and capability of our system validates this approach.

We began with a few simple experiments with shapes on a Web page made active in small ways by attached JavaScript methods. Encouraged by the responsiveness of both JavaScript and our graphics layer, we set about implementing a more complete framework for active graphical objects on a Web page. We chose to follow the Morphic architecture, which we knew from both the Self and Squeak programming environments, and which we consider to be a model of simplicity and generality.

II. A Quick Summary of the Morphic Architecture

The Morphic architecture is very simple. It defines a class of graphical objects, or "morphs", each of which has some or all of the following properties:

A shape, or graphical appearance
A set of submorphs, used to construct the "scene graph" of the page or world
A coordinate transformation that affects its shape and any submorphs
An event handler for mouse and keyboard events
An editor for changing its shape
A layout manager for laying out its submorphs
A stepping protocol for time-varying behavior
A damage region and repainting protocol and double-buffered display
mechanism when this is not available in the underlying graphics

A few other high-level morphs serve to complete a meaningful graphical environment. WorldMorph captures the notion of an entire screen view (often a Web page); its shape defines its background appearance, and its submorphs comprise the remaining content of the page. A world has a scheduler for managing user input, external input, and timer-based events. A HandMorph is the Morphic manifestation of a cursor; it can be used to pick up, move, and deposit other morphs. Its shape may change to indicate different cursor states, and it is the source of user events.

A property, that can be enabled or not, causes dropping of one morph upon another to make the first a submorph of the second. Mashups and new widgets or complete user interfaces can be assembled in this concrete manner.

In the Lively Kernel, a Morphic world may have several hands active at the same time, corresponding to multiple collaborating users of that world, and multiple worlds may be linked in the manner of linked Web pages.

Interested readers are referred to the original papers on Morphic [http://wiki.squeak.org/squeak/2139], and to the Lively Kernel technical documentation.

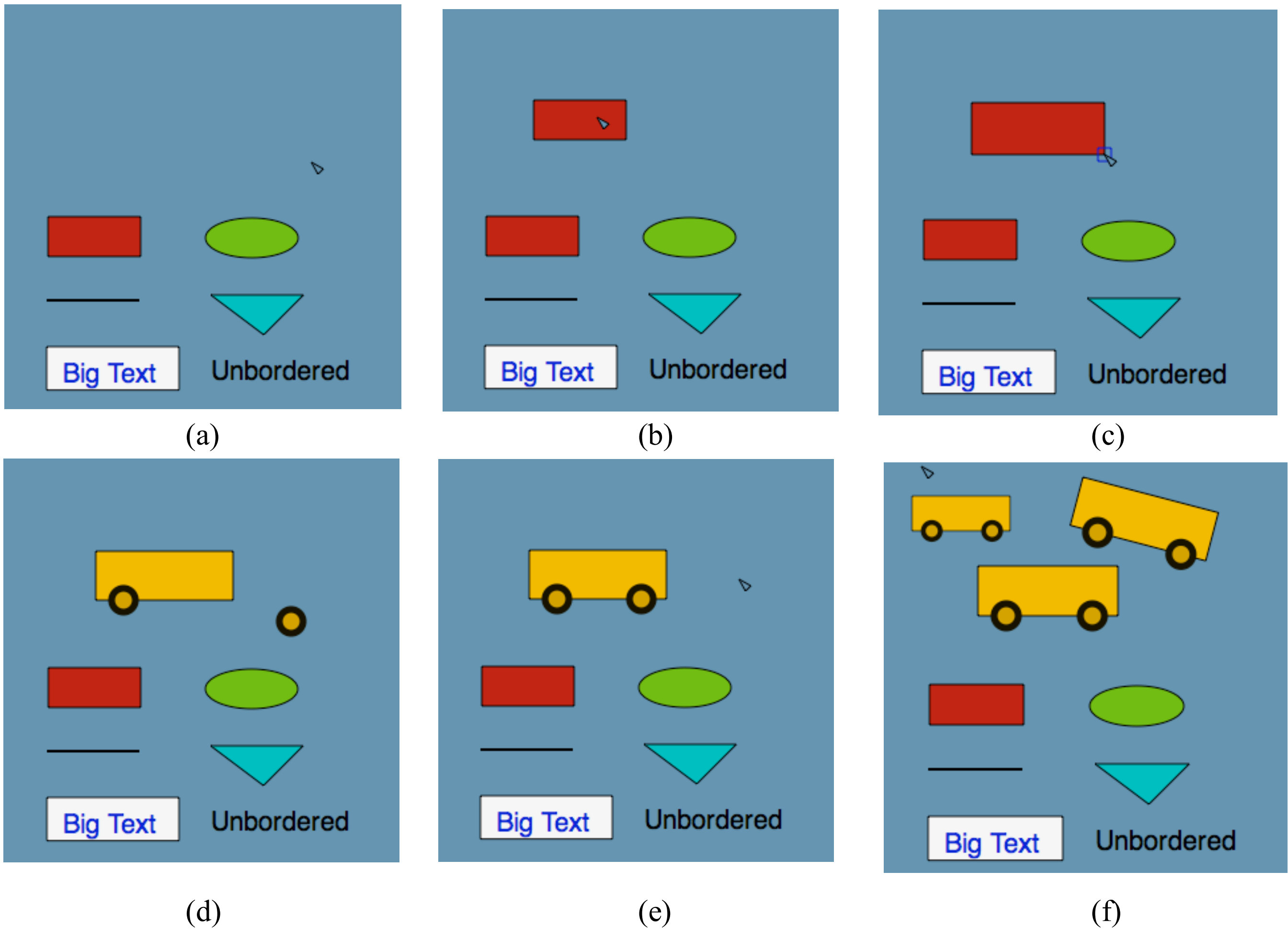
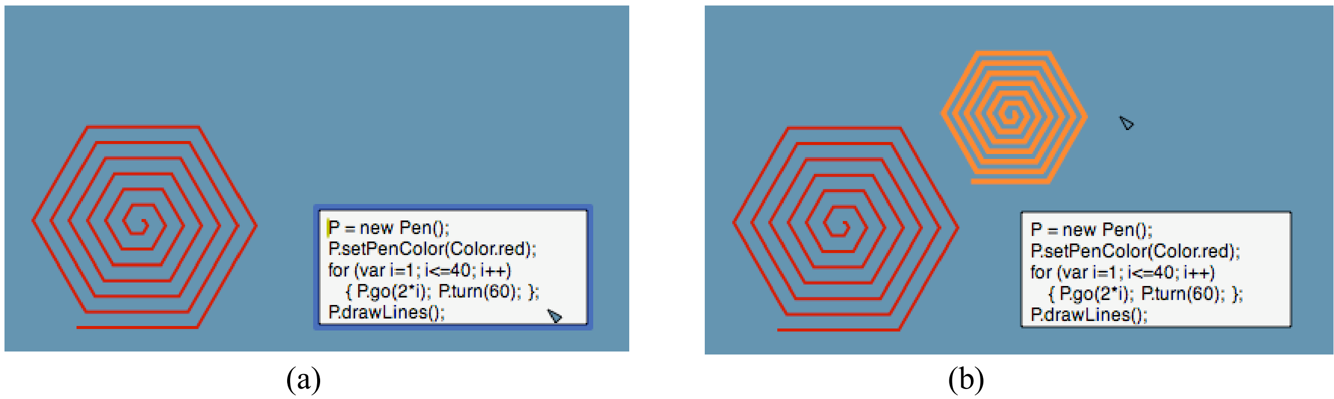


Figure 2: Drag-and-drop construction of simple objects in the Lively Kernel

III. A Lively Construction

In contrast with the static elements of most Web pages, each element of a Lively Web page is a Morphic object able to be picked up, moved, duplicated and reshaped. Thus, at its simplest, the Lively Kernel functions as a rudimentary graphics editor. The sequence shown in figure 2 illustrates the construction of a simple truck shape by concrete manipulation. In figure 2a we see a palette of useful shapes. In figure 2b, the rectangle has been copied, and extended in figure 2c. In figure 2d, the rectangle has been colored yellow, an ellipse has been copied, resized, and colored, and has been given a thick black border to resemble a tire, and a second copy has already been affixed to the bus. In figure 2e, the truck is complete, and figure 2f shows a family of trucks, copied and rotated from the new master, all operations that can be accomplished with simple gestures in the Lively Kernel's graphic editor. The next section will show how similar structures are built programmatically.



(a) (b)

Figure 3: Extending the graphical vocabulary

Beyond the basic vocabulary of the underlying graphics support, the programmability of Morphic shapes provides an unlimited range of graphical idioms. Figure 3 shows a small snippet of JavaScript that draws a polygonal spiral. Beyond mere marks on the screen, this code produces a fully active object that can be copied, scaled, and colored, as shown, and that could be set to spinning with one more line of code, such as

```
this.startStepping(50, "rotateBy", 0.1);  // 0.1 radians every 50 ms
```

IV. A Lively Clock

Here we present a simple application written in the Lively Kernel. The purpose is to illustrate the style of code written, and the advantages derived from the underlying architecture. Most of the code is in the "`makeNewFace`" method which centers the hour labels at equally spaced points around the face, and creates the three hands. Once the clock is created, the remaining task is to make the hands move. This is accomplished in the "`setHands`" method. Note that no code is required to update the image. It suffices to set the rotation of the hands appropriately; the architecture takes care of any required redrawing. The "`setHands`" method is scheduled to be called every 1000 milliseconds by the "`startSteppingScripts`" method, which is invoked whenever a new morph is placed into the world.

```
// ================================
// A Lively Kernel clock
// ================================
ClockMorph = Class.create(Morph, {
    defaultBorderWidth: 2,
    type: "ClockMorph",

initialize: function($super, position, radius) {
    $super(position.asRectangle().expandBy(radius), "ellipse");
    this.openForDragAndDrop = false;
    this.linkToStyles(['clock']);
    this.makeNewFace();
    return this;
},

makeNewFace: function() {
    var bnds = this.shape.bounds();
    var radius = bnds.width/2;
    var fontSize = Math.max(Math.floor(0.04 * (bnds.width + bnds.height)),2);
    var labelSize = fontSize; // room to center with default inset

    for (var i = 0; i < 12; i++) { // Place the 12 labels...
    var labelPosition = bnds.center().addPt(
             Point.polar(radius*0.85,((i-3)/12)*Math.PI*2)).addXY(labelSize, 0);
        var label = new TextMorph(pt(0,0).extent(pt(labelSize*3,labelSize)),
        // (i>0 ? i : 12) + "");  // English numerals
```

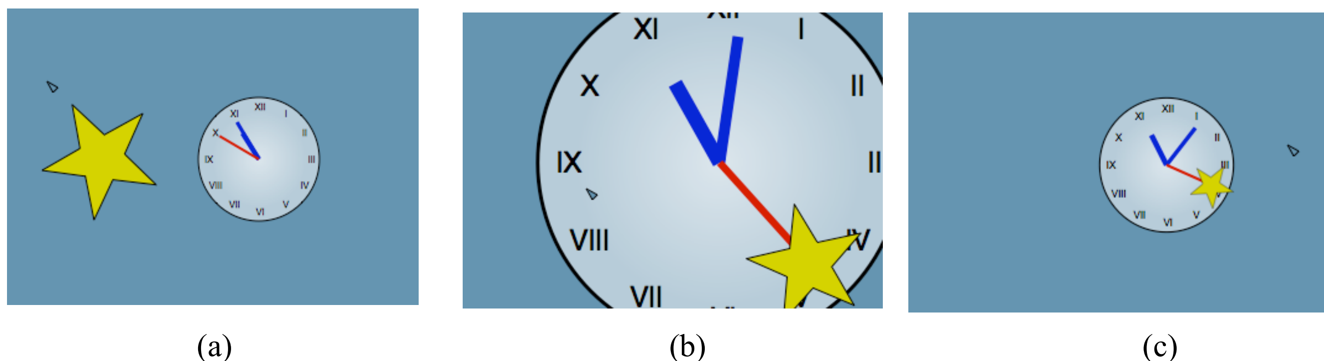
```
        ['XII','I','II','III','IV','V','VI','VII','VIII','IX','X','XI'][i]); // Roman
        label.setWrapStyle(WrapStyle.SHRINK);
        label.setFontSize(fontSize);    label.setInset(pt(0,0));
        label.setBorderWidth(0);        label.setFill(null);
        label.align(label.bounds().center(),labelPosition.addXY(-1,1));
        this.addMorph(label);
    }
    this.addMorph(this.hourHand = Morph.makeLine([pt(0,0),pt(0,-radius*0.5)],4,Color.blue));
    this.addMorph(this.minuteHand = Morph.makeLine([pt(0,0),pt(0,-radius*0.7)],3,Color.blue));
    this.addMorph(this.secondHand = Morph.makeLine([pt(0,0),pt(0,-radius*0.75)],2,Color.red));
    this.setHands();
    this.changed();
},

setHands: function() {  // Set hand angles from time
    var now = new Date();
    var second = now.getSeconds();
    var minute = now.getMinutes() + second/60;
    var hour = now.getHours() + minute/60;
    this.hourHand.setRotation(hour/12*2*Math.PI);
    this.minuteHand.setRotation(minute/60*2*Math.PI);
    this.secondHand.setRotation(second/60*2*Math.PI);
},

startSteppingScripts: function() {  // Will be called when placed in a world
    this.startStepping(1000, "setHands"); // once per second
}
});
```

Listing 1: A Morphic Clock in the Lively Kernel

We have already pointed out the architectural advantage provided in redrawing of the hands and periodic execution of the scheduled behavior. In addition, the entire construct inherits the ability to be scaled and rotated arbitrarily.



(a) (b) (c)

Figure 4: Composition of dynamic objects in the Lively Kernel

Figures 4a to 4c illustrate the flexibility of the Lively application architecture. In figure 4a we see a clock next to a (spinning) star. In figure 4b, the clock has been expanded to a large size (note that the text looks better, not worse), making it possible to drop the spinning star onto the end of the second hand. In figure 4c, we see the clock shrunk back to its normal size, but still sporting a spinning star on the end of its second hand. Of course, it is always possible to disable this kind of fanciful manipulation, but at this point we are exploring flexibility, not trying to prevent it.

The clock is a graphical assembly of text, lines and an ellipse, together with a simple script that endows the assembly with real clock-ness. In an equally simple manner, a basic set of “widgets” (common active user interface components) can be built up from the basic shapes plus a few simple methods. If the earlier truck example illustrated the construction of new molecules, then this is a bit like chemistry since, besides the mere assembly of parts, there is a meaningful model under each widget, and the interaction of those underlying values is the beginning of open-ended computing and self-support.

V. Lively Development Tools

Within the Morphic context, we chose to implement the Lively Kernel's widget set with a model/view separation along the lines of many Smalltalk systems. This choice was influenced by experience with GUI-builder applications and the Fabrik visual programming system. Besides allowing for multiple views of a given model, the model/view separation makes it easier to infer appropriate model structure from a given concrete assembly of UI components. It also turns out to provide a flexibility that is vital for migration of functionality between client and server where this is desired.

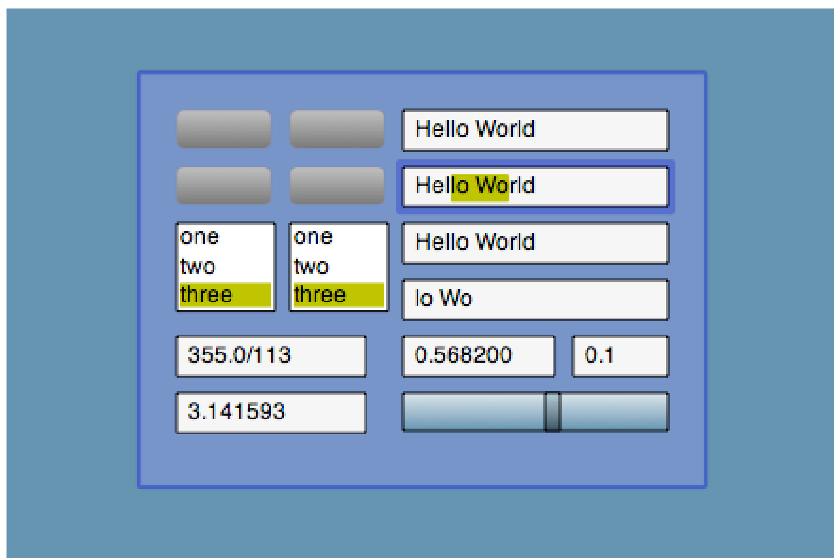


Figure 5: A text panel for widgets with shared models

Figure 5 shows a number of simple widgets arranged in a test panel. It is not much to look at as a picture, but if one runs the Lively Kernel from our site, one finds that the buttons, text boxes and lists are in groups that exhibit bidirectional coupling through their shared models. For instance the slider is hooked to a numerical model that is bidirectionally connected to one of the text views with a read/print converter.

We made the claim above that even a rudimentary assembly of widgets is the makings of open-ended computing. This should not surprise any of our readers, and we see in Figure 6 a piece of almost professional-looking software which is little more than an assembly of text boxes, lists, a clipping component, and the same slider shown in Figure 5, now doing service in (almost) vertical orientation as a scroll bar.

If we look more closely at the list on the left -- “ButtonMorph”, “ClipMorph” -- these are names of classes in the system itself, as are the selections, “ClockMorph”, and “setHands”. It is in fact a code browser (as its title bar confirms) written in the system, and viewing code in the system; in fact the very code exhibited earlier for the clock.

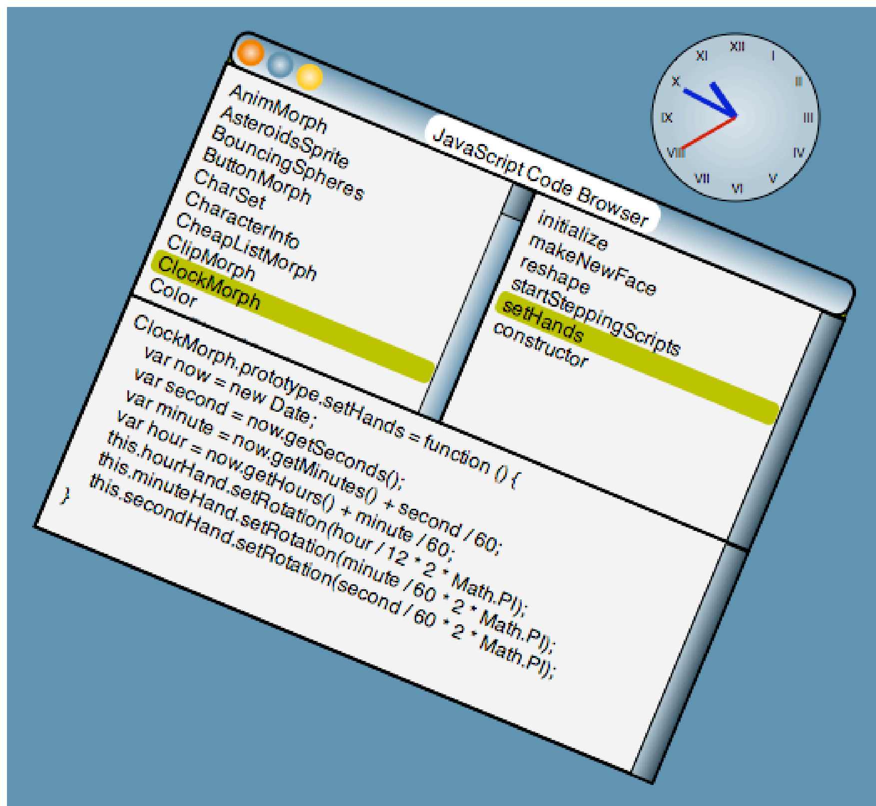


Figure 6: A Code Browser viewing the Clock application

Owing to the uniform graphics architecture, the browser application too can be used at any scale or rotation. If you are running the Lively Kernel, you may find this browser in the "Development Tools" world. Browse to this same method, add a minus sign to the parameter of the last setRotation call, and you will have a clock whose second hand runs backwards.

The code browser shown above is scarcely larger than than the ClockMorph class it is editing. We exhibit it here, if only to show that, having built up a rudimentary set of widgets, self-support with a graphical user interface is not so difficult to achieve.

```
// ===============================
// A Lively Kernel code browser
// ===============================
Widget.subclass('SimpleBrowser', {
    defaultViewTitle: "Javascript Code Browser",
    pins: ["+ClassList", "-ClassName", "+MethodList", "-MethodName", "MethodString"],

    initialize: function($super) {
        var model = new SyntheticModel(this.pins);
        var plug = model.makePlugSpecFromPins(this.pins);
        $super(plug);
        this.scopeSearchPath = [Global];
        model.setClassList(this.listClasses());
    },

    updateView: function(aspect, source) {
       var p = this.modelPlug;
       if (!p) return;
       switch (aspect) {
       case p.getClassName:
           var className = this.getModelValue('getClassName');
           this.setModelValue("setMethodList", this.listMethodsFor(className));
           break;
```

```
        case p.getMethodName:
            var methodName = this.getModelValue("getMethodName");
            var className = this.getModelValue("getClassName");
            this.setModelValue("setMethodString",
                            this.getMethodStringFor(className, methodName));
            break;
        case p.getMethodString:
            this.getModelValue("getMethodString"));
            break;
        }
    },

    listClasses: function() {
        var list = [];
        for (var i = 0; i < this.scopeSearchPath.length; i++) {
            var p = this.scopeSearchPath[i];
            var scopeCls = [];
            Class.withAllClassNames(p, function(name) { scopeCls.push(name);});
            list = list.concat(scopeCls.sort());
        }
        return list;
    },


    listMethodsFor: function(className) {
        if (className == null) return [];
        var sorted = (className == 'Global')
             ? Global.constructor.functionNames().without(className).sort()
             : Global[className].localFunctionNames().sort();
        var defStr = "*definition";
        var defRef = SourceControl &&
                    SourceControl.getSourceInClassForMethod(className, defStr);
        return defRef ? [defStr].concat(sorted) : sorted;
    },

    getMethodStringFor: function(className, methodName) {
        if (!className || !methodName) return "no code";
        else return Function.methodString(className, methodName);
    },

    setMethodString: function(newDef) { eval(newDef); },

    buildView: function(extent) {
        var panel = PanelMorph.makePanedPanel(extent, [
            ['leftPane', newTextListPane, new Rectangle(0, 0, 0.5, 0.5)],
            ['rightPane', newTextListPane, new Rectangle(0.5, 0, 0.5, 0.5)],
            ['bottomPane', newTextPane, new Rectangle(0, 0.5, 1, 0.5)]
        ]);
        var model = this.getModel();
        panel.leftPane.connectModel(  {model: model, getList: "getClassList",
                                       setSelection: "setClassName"});
        panel.leftPane.updateView("getClassList");
        panel.rightPane.connectModel( {model: model, getList: "getMethodList",
                                       setSelection: "setMethodName"});
        panel.bottomPane.connectModel({model: model, getText: "getMethodString",
                                        setText: "setMethodString"});
        return panel;
    }
});
```

Listing 2: A simple code browser.

Most of the browser code should be fairly self-explanatory. We point out that the reference to SourceControl allows this same browser to function stand-alone, reflecting on the sources of the running system (JavaScript functions will print themselves in response to `toString()`) or, in a team programming environment, it will access the original source code files in a repository. The `connectModel()` protocol in the `buildView` method provides for “pluggable” views so that, for example, the two list panes are connected to different aspects of the underlying model.

VI. More Tools for Self-Support

While a source code browser is the hallmark of self-support in any system, a number of other reflective tools are useful in the maintenance and evolution of a software system. The Lively Kernel provides a number of these, including, an Object Inspector, a Stack Viewer, and a Profiler. Examples of these tools appear in figure 7.

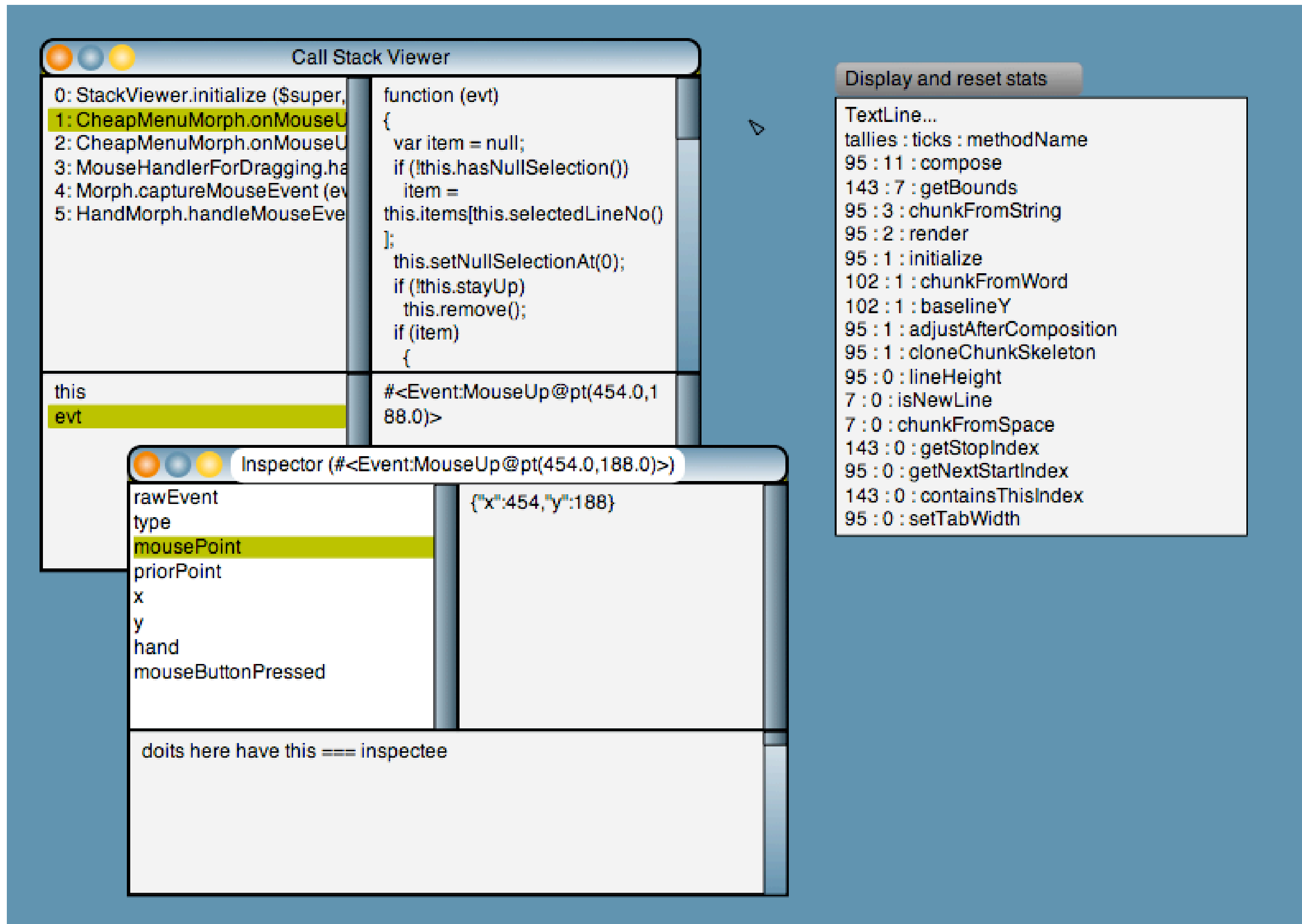


Figure 7: An Object Inspector, Stack Viewer, and Profiler

The Profiler and Stack Viewer are perhaps the most interesting of our reflective tools, because the normal JavaScript environment is missing the necessary reflection to provide them. However, the resourceful software engineer will find just enough reflection to provide these tools.

Consider the problem of execution time analysis: we wish to know exactly how many times each method is invoked in the course of some computation, either for rigor, or to understand where most of the time is spent. In the latter case, we would ideally like to see an accounting of the real time spent in each method as well. JavaScript engines provide neither of these reports, but they do, at least, give us a millisecond clock.

Listing 3 shows how the millisecond time can be used to provide a relatively complete profiler in a dynamic language environment. The essence of this function is simply an enumeration of all the methods in a class. If invoked with the parameter "`start`", then it replaces every method with an anonymous wrapping function (`tallyFunc`) that, after some bookkeeping, calls the original method, and if called with "`stop`", it undoes this replacement. The bookkeeping, in this case, involves

incrementing a tally count by one, and a ticks count by the number of millisecond ticks between call and return of the method. The remaining parameters for the outer call use the same enumeration to reset the tallies, or to collect them for reporting, as in figure 7. The button above the profile is a very simple control: each time it is pressed, it reads out the tallies and the tick timings, displays its report, and then resets all the tallies.

```
// ================================
// The Lively Kernel Profiler
// ================================
Object.profiler = function (object, service) {
    // Invoke as, eg, Object.profiler(Color.prototype, "start")
    var stats = {};
    var fnames = object.constructor.functionNames();

    for (var i = 0; i < fnames.length; i++) {
        var fname = fnames[i];

        if (fname == "constructor") {} // leave the constructor alone
        else if (service == "stop")
            object[fname] = object[fname].originalFunction;  // restore original functions
        else if (service == "tallies")
            stats[fname] = object[fname].tally;  // collect the tallies
        else if (service == "ticks")
            stats[fname] = object[fname].ticks;  // collect the real-time ticks
        else if (service == "reset") {
            object[fname].tally = 0; object[fname].ticks = 0; // reset the stats
        } else if (service == "start") { // Replace original functions by tallyFunc wrapper
            var tallyFunc = function () {
                var tallyFunc = arguments.callee;
                tallyFunc.tally++;
                msTime = new Date().getTime();
                var result = tallyFunc.originalFunction.apply(this, arguments);
                tallyFunc.ticks += (new Date().getTime() - msTime);
                return result;
            }
            // Attach tallies, and the original function, then replace the original
            if (object[fname].tally == null)
                tallyFunc.originalFunction = object[fname];
            else
                tallyFunc = object[fname]; // Repeated "start" will work as "reset"

            tallyFunc.tally = 0;
            tallyFunc.ticks = 0;
            object[fname] = tallyFunc;
        }
    }
    return stats;
};
```

Listing 3: The Lively Kernel Profiler

The Profiler shows the degree to which a dynamic language environment can amplify even the simplest reflective capability. In this case the millisecond clock, and ability to wrap and replace methods yields a relatively powerful profiling tool in only half a page of code.

It is lamentable that the JavaScript standard provides almost no access to the runtime execution state, such as call stack, temporary variable values, and the ability to resume a suspended computation, but we can at least make the most of what is there. JavaScript does provide a pseudovariable "arguments" whose value is an array alias of the arguments passed on call. It also tacks a "callee" property onto that array that allows access to the function that is running. Is this enough for reasonable debugging? In some JavaScripts it is almost enough to provide a stack trace because a non-standard feature in some JavaScripts allows a function object to return its "caller", but this is not a proxy to the activation record, and thus is useless in the presence of recursion.

Wrap-and-replace to the rescue! In the Lively Kernel we support a debugging mode of execution that

wraps every method in the system with a function which appends a reference to the arguments array, as well as to the receiving object ("this") to a shadow stack that is created afresh each time through the Morphic event loop. This allows us to provide not only a stack trace, but also the ability to inspect the receiver and arguments at every level of the call chain, either at will or when an exception is encountered. It is worth noting that this management of our own stack allows us access to these values after an exception has been thrown, whereas our experience is that most JavaScript engines discard this state before giving control to the exception handling code. It can be viewed as a tribute to the power of today's computers that this level of simulation does not bring the Lively Kernel to a complete halt. In fact we hardly notice the impact on performance.

VII. Team Programming

One last tool is worthy of mention, given the context of this paper. The Lively Kernel includes a rudimentary file parser which provides a bridge between the source code file style of most Java and JavaScript developers, and the per-method management of source code such as we know from Squeak and similar systems. When viewing the system source files, each method has an associated sourceCodeDescriptor that delimits its location in the file. We keep a careful reckoning of changes for each file so that descriptors from earlier versions of a given file can still be used to make changes in later versions (we re-read the segment as a check before committing any change). This enables our source code browser to browse both the running code in the system and the shared sources in a repository. The source code file approach is useful for team programming, given the existence of external tools such as CVS, Subversion, and the like.

We talk here of files, but the Lively Kernel, being Web-borne software, makes no reference to disk files on the user's machine. Instead we use a basic WebDAV protocol (see http://tools.ietf.org/html/rfc2518) that allows Web sites to be treated as read/write file systems. Of course a user may run a local file server on his laptop in order to work away from the Internet, but this style of access to resources ensures that the Lively Kernel can be used anywhere on the Web.

These source code objects are useful in a number of different ways. For instance, one can type a search string in the Lively Kernel, and get an instant list of all occurrences in the source database that match the string. Such search results are presented as a change list viewer, and the methods so viewed can be edited there in place. It is a gratifying result of the Lively Kernel's compactness that these searches scan our entire code base and display their results in just one second.

Associated with each world in the Lively Kernel is a list of changes that have been made to the system. These can be viewed as a change list, which is handy both for viewing prior versions, and for ready access to just the work in progress. Most importantly, this log of changes is retained within the running system. If the user saves any world as a new Web page, the associated list of changes for that world will also be saved. At a later time, on a different machine, in another browser, that page can be reloaded, the change list replayed, and the work continued in a seamless manner. In this way, the Lively Kernel enables a Smalltalk-image style of evolutionary development.

VIII. Application Development for the Web

Is a system capable of self-support necessarily a good foundation for the development of general-purpose applications? We believe so. We have experimented with simple media, interactive games, RSS feeds, chats, and mashups and, in every case, the architectural substrate of the Lively Kernel has shown itself to be effective. Some early experience is recorded in our Sun Tech Report (TR-2008-175, January 2008 in refs).

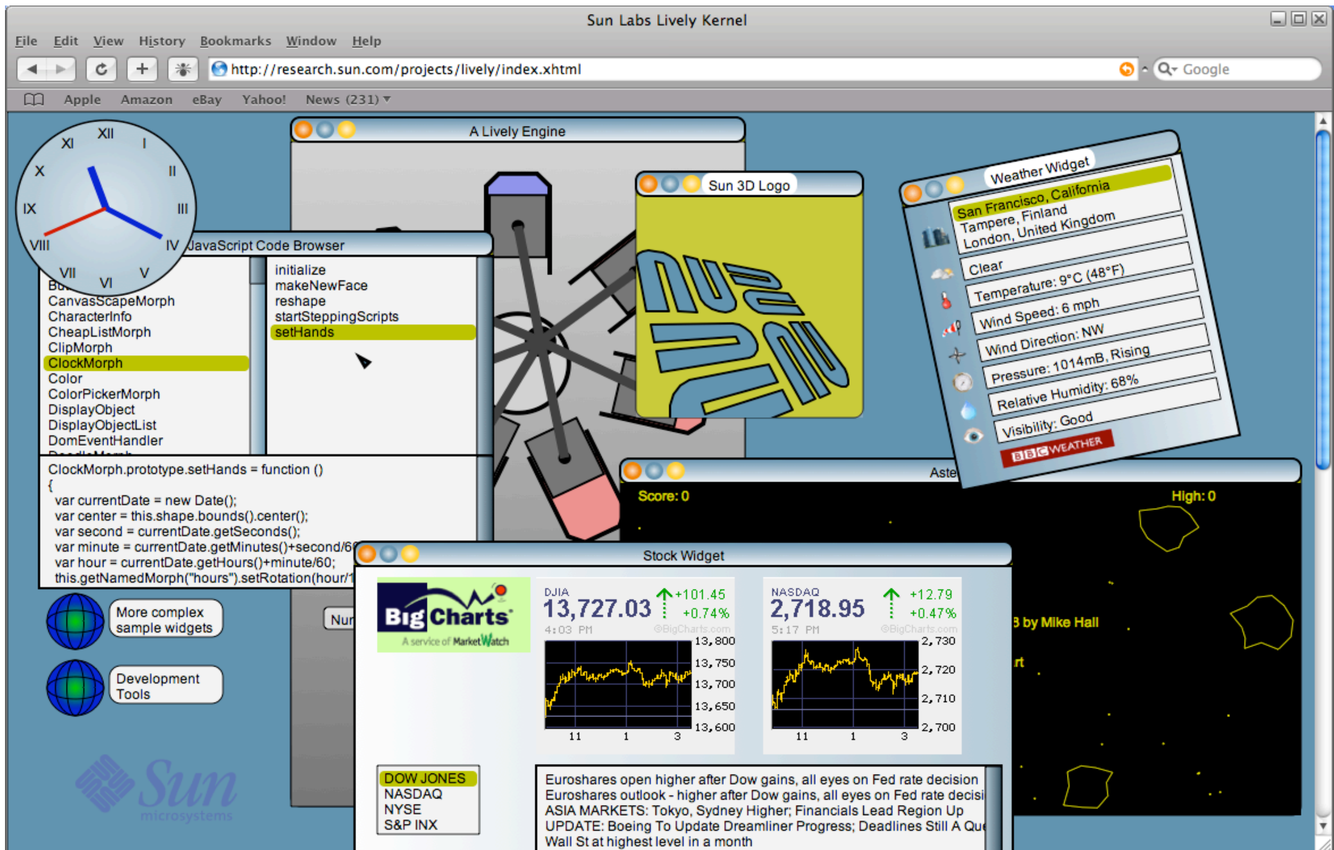


Figure 8: The Lively Kernel running numerous applications in an overlapping window framework. This is a Web page

Figure 8 shows a Lively Web page that is a mashup of a number of applications, all active and all manipulable. The clock and browser will be familiar from earlier in this paper. The other applications include an asteroid-blasting game, a Web weather viewer, a Web stock report, a demonstration 3-D viewer, and a simulation of a seven-cylinder radial engine (running). Also visible are a couple of links to other worlds with more applications including an RSS feed reader, a GoogleMaps viewer, and a personal information manager.

When the Lively Kernel stores a simple object, an application, or an entire world, it does so on a Web page. If one looks at one of these pages, one sees a link to import the core of the Lively Kernel, followed by markup describing the objects that have been stored. The link to the Lively Kernel both declares this to be a page of lively content, and provides the interpretive engine for bringing the stored objects to life.

Of course, similar things have been done on the Web for years, using plugins and applets to provide the engines of active content. The twist in the Lively Kernel is to use JavaScript for all the machinery of activity, thus avoiding the need to install a plugin. Other libraries, such as Dojo or Scriptaculous, can operate without installation, but the Lively Kernel goes several steps further. First, since its graphical library is built from the ground up in JavaScript, it sets the stage for a world without HTML and the epicycles that revolve around it. Second, it brings with it a world model in which everything is active and reflective from the beginning, a world of concrete manipulation that is immediately empowering to developers and users alike.

Figure 9 illustrates the simple modes of storing and retrieving Lively objects on a Web page

Load into...	Store as... One object	Entire world
Empty browser	Object in empty world	Entire world recreated
LK world	Object in current world	Link to new sub-world

Figure 9: Modes of storing and retrieving Lively Kernel objects

IX. A Benchmark Kernel

Beyond its immediate utility, the simplicity and completeness of the Lively Kernel make it a meaningful benchmark of system complexity. It provides a widget set and tools to construct an application from those widgets. It provides screen management and process management facilities, along with a modest IDE. In short, it provides all the tools needed for application development and deployment, as well as for evolution of the system itself.

We consider this accumulated functionality to be a meaningful unit of comparison. How many lines of code does it take to produce such a kernel? How is the performance, and by what is it most limited? These and other questions can be asked, and answered concretely with an artifact like the Lively Kernel as a benchmark.

The reader may be interested to know our experience so far in this regard. Figure 10 presents a breakdown of the Lively Kernel by functional category in terms of lines of code.

```
  319  Host Interface – Browser API plus XML utility functions
  515  Utility – Classes, collections, printing, etc. (plus 562 in Prototype.js)
  833  Basic Graphics – Point, Rectangle, Transform, Color, Gradient, Image
  550  Shapes – Host graphical objects (SVG node API)
 1667  Morph – Basic protocol
 1170  Morphic Core -- World, Hand, Event, Handles, Handlers
 1188  Basic Widgets – Button, List, Menu, Dialog, Slider, Selection, ImageMorph
 1921  Text – basic TextMorph plus composition and rich text
  747  Editors – Drag/drop manipulation, shape editors and text editing
                         also ColorPicker and StylePanel
  386  Model, Widget
 1295  High Level Widgets – Scroll panes, panels, windows, world links
                         also Panel and Browser support
 1311  Tools – Browser, inspector, stack viewer, change lists, profiler
  406  Serialization – Copier, exporter, importer (plus 202 in JSON)
  281  Network – URL, HTTP, WebDAV basics
  365  Storage – WebDAV, file browser
-----  -----
12954  Total
```

Figure 10: Breakdown of Lively Kernel by function with approximate code size

The figures above include comments and lines with single bracket characters. Prototype.js is an open source set of useful JavaScript extensions, of which we use only a small number [see http://www.prototypejs.org/], and JSON is a nice encoding for JavaScript objects by Doug Crockford [see http://www.JSON.org/].

It is surprising to some that text should be the largest module in a kernel such as ours. In our

experience, it is often the handling of text that makes the difference between a toy and a serious tool. The Lively Kernel text is built from the ground up, so it can run where no native text support is available. The tally includes all the functions for mouse tracking, line composition, selection, rich text encoding, font changes, and on-the-fly layout. It seemed to us the only approach consistent with a world of active objects.

Clearly this system goes beyond the minimum functionality required for self-support. Windows, nested worlds, rich text, arbitrary scaling and rotation are all in some sense frills. Our intention in carrying the work this far was to make it more likely that people might pick up the work and do surprising things without needing to build a lot more infrastructure.

As with other benchmarks, we see the Lively Kernel as a starting point The challenge is to build an even simpler graphical model, an even more general processing model, a smaller complete kernel, and so on. The standard to be met, in every case, is a kernel capable of building itself, and altering and saving itself again as a Web page.

X. Related Work

It will be obvious to most readers that the Lively Kernel inherits much genetic material from the Squeak Smalltalk system and the Smalltalks that preceded it as well. Also the Morphic architecture, while most directly inherited from the (class-based) Squeak version, began as part of the Self project at Sun.

We know of no other self-supporting JavaScript development system, let alone one that runs directly off a web page without installation. However numerous web sites use the underlying JavaScript engine to provide an execution facility for JavaScript snippets, usually as part of a tutorial environment. The nicest we have seen is Takashi Yamamiya's live JavaScript Wiki (ref). Also there are some interesting examples of similar tutorial or exploratory execution environments for Logo (Colin Putney and Alan kay, LogoWiki ref?), Squeak (Alex ref?), and prolog(ref?), hosted on top of JavaScript. In this regard, Alex Warth's OMeta system (ref) is relevant, as it facilitates this kind of emulation in the Web environment.

We have learned much about JavaScript as a programming language. While it is beyond the scope of this paper, some of our early impressions are recorded in a Tech Report (Sun Labs TR-2007-168, October 2007).

XI. Future Work

This kernel we have described is lively in yet one other respect: It is small and simple, and thus easy to change and port. So if we imagine a secure subset of JavaScript, or a nice 3D graphics system for the Web, it should be straightforward to port the Lively Kernel to these environments and to observe immediate results in the practical world of active Internet content. We have already gone through several complete rewrites of the system and have found it to be a tractable task. For instance we rewrote the entire graphics substrate from one built on a flat graphics model (Java2D) to a retained graphics model (SVG) in roughly two months. Our experience suggests that many meaningful transformations of the Lively Kernel could be done by a graduate student or other serious programmer in a month or two. Simple experiments can, of course, be tried in much less time.

We list here some areas that we have identified for future work:

Caja[http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf] is a secure subset of JavaScript. At the time of this writing, it is a research project that has not yet been tried on real applications. But if the Lively Kernel can be ported to the Caja model, it will be an existence proof of an entire application platform with known modularity and security properties. We hope to produce a version of the Lively Kernel that is consistent with the Caja rules for security, thus ensuring that Lively mashups and other cooperating applications will be well-behaved

Lessphic[http://piumarta.com/software/cola/canvas.pdf], as its name suggests, is an alternative to the Morphic architecture with various desirable properties. We are investigating a port of the Lively Kernel to the Lessphic model for the purpose of validating some of the apparent benefits of this design.

GUI Builder for the Internet. The current Model and Widget framework of the Lively Kernel has been designed to facilitate extremely simple (drag-and-drop) construction of useful panels to control all sorts of Web-based resources. We hope to demonstrate a number of these in the future. [Fabrik ref?]

End-user programming. All of the required elements to support an Etoy-like environment already exist in the Lively Kernel. We believe that could enable the creation of interesting active Web objects conceived and built by end users.

Beyond JavaScript. We have only used JavaScript because it is available in every browser. We find that we have no need for a number of features in the language, and this suggests the possibility of simpler host implementations that are smaller and run faster. [COLA ref]

Beyond SVG. We have similarly been using SVG because it is available in many browsers. Here again, we find no need for many features in the standard, and this suggests the possibility of simpler implementations that are smaller and run faster. [Gezira ref?]

Beyond Browsers. Having turned Web programming upside down in order to achieve a simpler and more general world within the browser it is hard not to ponder, from time to time, going all the way and building a complete browser within the Lively Kernel.

XII. Conclusion

During the Lively Kernel project, we have learned some things about the kernel as a concept and as a vehicle. Rather than complain about the languages available or the inconsistencies between various browsers, we have done our best to pick one viable solution and to preserve every bit of liveliness for the developer and ultimately for end users. Rather than dwell on perfection in one area or another, we have pressed for the ability of end users to immediately publish and share their their creations. Having glimpsed the possibility, our passion is now to enable such authoring and sharing for every user of the Internet. It is our hope that, seen in this fresh perspective, and now available as a tangible artifact, the Lively Kernel may inspire further progress toward simplicity, generality and liveliness in Web programming.

XIII. Acknowledgements

The authors wish to acknowledge the help of Charles Jackson, Alan Lancendorfer, and Mary Holzer for their help in setting up the Lively Kernel Web site, and Pekka Reijula and Mikko Kuusipalo for their contributions as interns, including much early testing of the Lively Kernel application framework. Also Richard Ortiz for his help with a number of application and porting experiments, Kristen

MacIntyre for contributions to text display and affine transforms, and Mario Wolczko, Bob Sproull and Greg Papadopoulos for their enthusiastic support of this project.

XIV. References

Antero Taivalsaari, Tommi Mikkonen, Dan Ingalls and Krzysztof Palacz.
Web Browser as an Application Platform: The Lively Kernel Experience.
Sun Microsystems Laboratories Technical Report TR-2008-175, January 2008.

URL to PDF: http://research.sun.com/techrep/2008/abstract-175.htmlMaloney, J.H., Smith, R.B., Directness and liveness in the Morphic user interface construction environment. Proceedings of the 8th annual ACM Symposium on User Interface and Software Technology (UIST), Pittsburgh, Pennsylvania, 1995, pp. 21-28.

Maloney, J.H., Morphic: The Self User Interface Framework. Self 4.0 Release Documentation, Sun Microsystems Laboratories, 1995.

Ingalls, D., Kaehler, T., Maloney, J.H., Wallace, S., Kay, A., Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. Presented at the OOPSLA'97 Conference.
http://ftp.squeak.org/docs/OOPSLA.Squeak.html.

Mark Miller, Mike Samuel, Ben Laurie, Ihab Awad, Mike Stay, Caja: Safe active content in sanitized JavaScript, January, 2008
http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf

Mikkonen, T., Taivalsaari, A., Using JavaScript as a Real Programming Language. Technical Report TR-2007-168, Sun Microsystems Laboratories, 2007.

Piumarta, I., COLA whitepaper: Albert, VPRI Research Note RN-2006-001-a
http://vpri.org/pdf/colas_wp_RN-2006-001-a.pdf

Piumarta, I., Lessphic: A disposable, light-weight graphical enviroment for FoNC
http://piumarta.com/software/cola/canvas.pdf

Tommi Mikkonen and Antero Taivalsaari, Using JavaScript as a Real Programming Language.
Sun Microsystems Laboratories Technical Report TR-2007-168, October 2007.
http://research.sun.com/techrep/2007/abstract-168.html

Various, History of Morphic
http://wiki.squeak.org/squeak/2139

Warth, Alessandro and Piumarta, Ian
OMeta: an object-oriented language for pattern matching
Proceedings of the ACM 2007 Symposium on Dynamic languages, pp 11-19
http://portal.acm.org/citation.cfm?id=1297081.1297086