# A World of Active Objects for Work and Play

## The First Ten Years of Lively

### Daniel Ingalls

Y Combinator Research
San Francisco, CA, USA
Dan.Ingalls@ycr.org

### Tim Felgentreff

Hasso Plattner Institute
Potsdam, Germany
tim.felgentreff@hpi.de

### Robert Hirschfeld

Hasso Plattner Institute, Potsdam,
Germany
robert.hirschfeld@hpi.de

### Robert Krahn

Y Combinator Research
San Francisco, CA, USA
robert.krahn@ycr.org

### Jens Lincke

Hasso Plattner Institute
Potsdam, Germany
jens.lincke@hpi.de

### Marko Röder

Y Combinator Research
San Francisco, CA, USA
marko.roeder@ycr.org

### Antero Taivalsaari

Nokia Technologies
Tampere, Finland
antero.taivalsaari@nokia.com

### Tommi Mikkonen

Tampere University of Technology
Tampere, Finland
tommi.mikkonen@tut.fi

## Abstract

The Lively Kernel and the Lively Web represent a continuing effort to realize a creative computing environment in the context of the World Wide Web. We refer to that evolving system simply as *Lively*. Lively is a live object computing environment implemented using JavaScript and other techniques available inside the browser. When first built in 2006, it was a grand accomplishment to have created such a system that would run in any web browser and that could be saved and loaded simply as a web page. Since that time we have learned a lot about the goals we had, the challenges and opportunities that come with life in the browser, and the exciting possibilities that still lie ahead.

***Categories and Subject Descriptors*** D.2.6 [*Programming Environments*]: Interactive environments; D.2.m [*Miscellaneous*]: Rapid prototyping; D.3.3 [*Language Constructs and Features*]: Frameworks

***General Terms*** Design, Experimentation

***Keywords*** Web programming, Software as a Service, Live Object System, Lively Kernel, Lively Web, Lively, JavaScript, Morphic

## 1. Live Object Systems

Lively [12] is a *live object system* which provides a web programming and authoring system to its users. By live objects we mean entities that can usually be seen, touched, and moved and that will react in a manner prescribed by some set of internal rules. A live object system is thus a kernel system for creating, manipulating, and composing live objects. A live object system is less and more than a web programming environment. That kernel is much less than a web programming environment; it may only allow the manipulation of a few shapes and a few rules of behavior. Yet, from such a kernel can be built an entire web programming environment, a complex data visualization system, a visual programming system, or even the kernel itself. In that sense it is much greater than a web programming environment.

Producing a live object system is more of an artistic challenge. One must choose a suitable set of atomic objects to begin with, a mechanism for composing them, a paradigm for their behavior, and a simple yet general framework for controlling all these properties. This may sound more technical, but the goal is an artistic one: to support the entire flow from a sketch of an idea to a concrete manifestation to animation of parts to a simulation of the whole and then to a presentation or publication.

Many live object systems have come before Lively: Sketchpad [26], Smalltalk-72 [2], ThingLab [1], Fabrik [13], Smalltalk-76 [6], Squeak [11], and Etoys [14], to name a few. The earlier of these are live constraint systems, live dataflow systems, and live educational systems. Only Squeak succeeds in being simple and yet general enough to implement its entire user interface and ultimately its own virtual machine. In that sense Squeak can be said to be a universal live object system. Our goal for Lively was and is to provide such a universal live object system that runs in any web browser.

We consider the *Morphic* graphics architecture [22] – originally pioneered in Self [30] – to be a uniquely powerful framework for producing a simple live object system. It defines a scene graph and paradigm for scripting behavior and a scheduler for coordinating dynamic behavior of many objects in complex relationships on the screen.

This paper begins with a brief history of live object systems and summarizes what it took to build the first such one running in a web browser. We then trace a number of interesting additions that have made Lively an ever more powerful tool for exploring the possibilities of live object systems on the Internet, and some notable applications that have been built and deployed using Lively. We note a number of experiments that we have tried and not yet incorporated into the Lively environment, and we observe some areas where we feel there is work yet to be done. Finally, we put things in a broader context of web programming and discuss ways to make Lively and similar systems ever more useful to users and relevant in the evolving world of the Internet.

## 2. How Lively Came to Be

Year 2005 found several of the authors at Sun Microsystems Labs, the birthplace of the Java programming language, feeling stifled by the static nature of this language and its development systems. By comparison, web programming looked like fun, and certainly more *au courant*. However, the Web too seemed burdened by the complexity of HTML, CSS, PHP and other technologies developed by non-programmers for non-programmers.

With the growth of the World Wide Web, it was obvious to us that the web browser would became a universal platform for graphical display. The JavaScript language, slipped into the web standard almost as an afterthought, turned out to have intriguing strengths. JavaScript was a garbage-collected dynamic language that could behave much like Smalltalk, only with the syntax of C and Java. It was tempting to consider re-purposing this language and the web browser to recreate the conditions for creative programming in the context of what was becoming a universal platform. To investigate this possibility, we equipped a browser with general (Java2D) graphics hooked to JavaScript, and added a small class library (Prototype.js and some graphics support). This allowed us to duplicate simple Java test programs in the browser framework, which was very motivating. JavaScript may not be the best programming language, but it was a refreshing change from the complex Java frameworks available at the time.

It was exciting to see a much simpler system duplicating these test programs, but it still lacked the live object feel of our earlier experience with Squeak. Feeling this frustration, and armed with this essential language and graphics support, one of the authors over Christmas vacation 2006 implemented a Morphic system in JavaScript, and in a matter of weeks we had a working set of widgets, a live code browser, in fact all the essential characteristics of a live object system, running in a web browser.

Once we had Morphic running in a browser, we could sense success within our reach but much remained to be done. While we could edit code live in the system, it could not be saved back into the source code base, so we had to devise a bridge from the live code edits back to our source files. At first we had direct access to source code files on our local disks, but we knew that Lively, to be live web software, must work entirely from a server. To that end we devised a mechanism using WebDAV on our Apache server to access an SVN repository with our code in it. Around the same time, we wrote a Smalltalk-style code editor and file format convention that allowed us to browse source code by class and method name, even though they were stored as JavaScript files that could be loaded at full speed in any browser.

We needed a way to save Lively worlds, both for continuity of project development and for the release of useful creations as web pages. For that, we implemented a convention whereby every morph in the scene graph of a Lively world held its persistent state in a known format. This allowed Lively worlds to be saved by walking the scene graph tree and storing the state in a format similar to JSON.

Based on our experience with Squeak, we were committed to providing a completely general graphics system, with lines and curves and the ability to translate, scale and rotate any of the graphical elements in a general manner. For this we had relied on our Java2D plugin, but we knew that many users would not have access to such a plugin. Fortunately, at about this same time, browser support for SVG was becoming available, and we were able to rewrite our graphics implementation layer on top of SVG.

With the ability to access code and stored worlds in a repository and support for SVG graphics available in the browser, Lively was finally ready to be released as a live object system that runs in any browser. It was released to the public on October 1, 2007.

The following key points are from the press release notes from the October 2007 release of Lively:

1. *It comes live off a web page*. There is no installation. The entire system is written in JavaScript, and it becomes active as soon as the page is loaded by a browser.
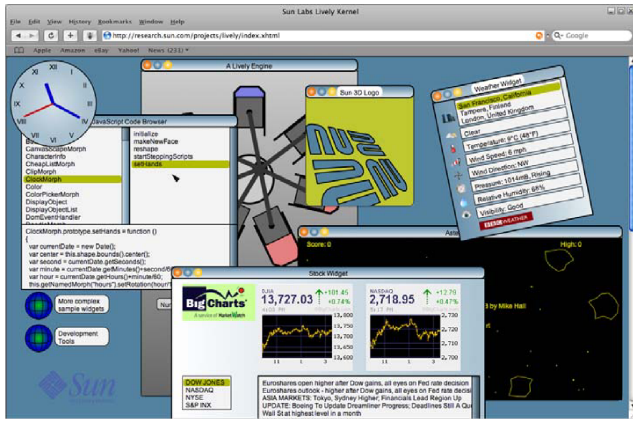
**Figure 1.** Lively Kernel from October 2007.

2. *It can change itself and create new content*. The Lively Kernel includes a basic graphics editor that allows it to alter and create new graphical content, and also a simple IDE that allows it to alter and create new applications. It comes with a basic library of graphical and computational components, and these, as well as the kernel, can be altered and extended on the fly.

3. *It can save new artifacts, even clone itself, onto new web pages*. The kernel includes WebDAV support for browsing and extending remote file systems, and thus has the ability to save its objects and "worlds" (applications) as new active web pages.

Back in mid-2000's, the concept of Software as a Service had not been popularized widely yet, so the idea of running an entire software development environment within the web browser was seen as radical by our peers. However, we actually wanted to go further and turn the entire Web into a playground for dynamic objects.

## 3. The Evolution of Lively

While the first release of Lively was somewhat of a triumph, much work remained to be done, some by choice, some by necessity. Almost immediately we had need of a decent module system. We were bound to the list of .JS files in the preamble of our stored pages, and it was very difficult to restructure our system for that reason. Also, as the system grew, there were many files that were only needed for special operations, so we wanted to be able to load parts of the system on demand. Exploring in advance of other development systems on the Web, we had to do this on our own in 2008, but it quickly repaid our efforts by simplifying organization and reducing our load times.

Next, feeling the need for a more forgiving storage system, we implemented a client-side Wiki system. In this manner, all files retained prior versions, so that it was always possible to revert changes if something went wrong. This robustness extended to stored web pages and applications as well as to the JavaScript files for the core system. At this

time a new more powerful version of the code browser for system classes was developed to partner with the new module system and the underlying SVN repository that we used at the time (see Figure 2).
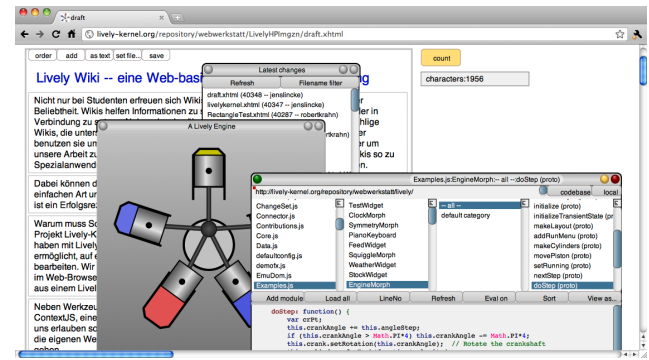


**Figure 2.** Lively Wiki from 2009. Lively Kernel pages can link to each other. Multiple users can asynchronously modify objects on a page, performing not only text editing but also graphical manipulation and programming. Full access to the version history of individual pages allows users to revert changes. Unlike traditional wikis, the entire environment including the wiki system itself can be edited from inside Lively, rendering the development of Lively Kernel self-supportive [15].

As Lively matured, so did much other dynamic content on the Web, most of it in HTML format. While mashup creation – the ability to combine code and content flexibly on the fly – was a key strength of Lively, we were constrained by our dependence on SVG for a rendering architecture. Therefore in 2010 we undertook a somewhat traumatic rewrite of the entire system to convert our Morphic rendering system to be compatible with HTML. This was complicated by the fact that in our original SVG-based implementation we had had to implement our own text system from scratch. There were no live text editors for the Web at the time of our release, so all of this code had to be rewritten from SVG to HTML.

As we left the convenience of SVG's excellent graphical transformation architecture, we had to subvert CSS to perform those functions, while leaving the Morphic programming layer as clean and simple as ever. We also undertook to go even further and support not only simple HTML, but also embedded SVG elements for lines and curves, and Canvas elements for arbitrary sketches and bitmap images. The fruit of this labor was that we were soon able to include web content such as Google maps and embedded videos in Lively, all as easily as any other components. While this was a huge amount of work (born almost entirely by one of the authors), it is probably the single factor most responsible for work in Lively remaining relevant to this day.

Also in 2010, two of the authors were asked to prototype a visual guidance system for Daimler that would help drivers to find the nearest gas station or, better, the one that would be near enough but require the least extra driving (see Figure

3). To make for a smooth demonstration, they constructed a palette of useful components that could easily be dragged in to the evolving application as required. Soon they were also using the palette to save new components for later use and, naturally, wanting to save these components into the Lively repository. This facility was enormously valuable and quickly evolved into the *PartsBin* that we know today in Lively (see Figure 6) [21].
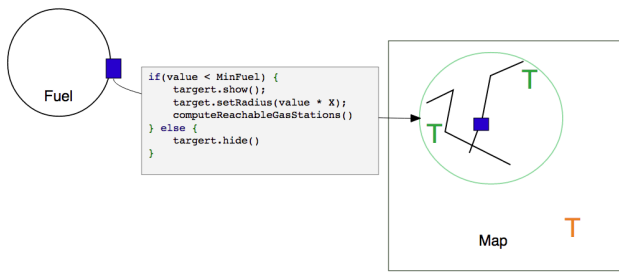


**Figure 3.** First draft of an interactive fuel consumption demo shown in Figure 5 from 2010. The Connections visualization tool in the final simulation evolved while building the application itself.

In the very same prototyping effort there was a need to quickly and simply add scripts to simple shapes and elements of the demonstration without having to go through the system browser and its JavaScript class definitions stored in module files. For this, in an afternoon, they used Lively itself to build a simple code window in which they could attach simple JavaScript methods to any object in the world. This too was immediately useful and quickly evolved into the *Object Editor* that we know in Lively today (Figure 7). The fact that two of the most useful tools in the Lively environment came about in a week of working on customer prototypes and demos illustrates the fact that nothing helps to improve a system more than a real-world challenge and real users.

By 2011 we had become aware of the Node.js project and started following it with interest. We shared the same need for a decent module system and good JavaScript coding standards and libraries. We experimented with a proxy from our Apache server to a Node.js server, and finally in 2012 we moved all our system over to use Node.js. The wonderful thing about Node.js for us was that it looked and worked so much like Lively; it was built on the V8 JavaScript engine, and was compatible to the point of being able to run much of our code unaltered. We quickly developed the "Subserver Viewer" that allowed us to create and test new server processes in a minute, without having to reload either client or server. Suddenly we were able to work with our server as easily as with any other Lively component. In Lively, it is a simple matter to have one window open on client code, and another on server code, and to make and test corresponding changes in both in seconds.
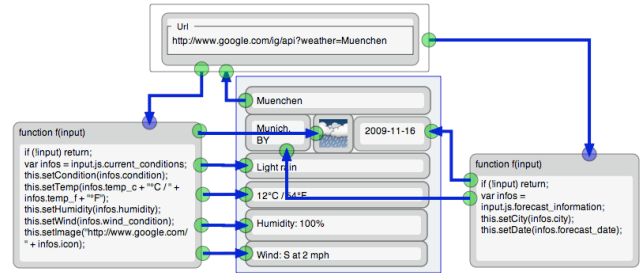


**Figure 4.** Lively Fabrik from 2008, a data flow based end-user programming environment in Lively. The whole system consists of UI components, scripts, components and wires establishing data flows between them.
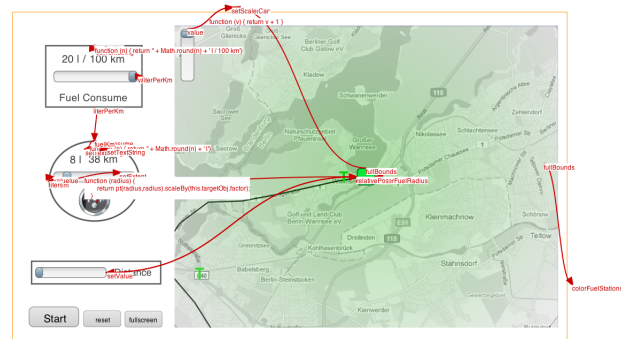


**Figure 5.** Dataflow visualization of an interactively scripted auto fuel consumption simulation from 2010. Leveraging Lively Fabrik, Lively Connection bindings allow JavaScript objects to observe each other. With this generalization we were able to extend the domain-specific dataflow development style into a general Lively Kernel feature for application and tool development.

A part of our vision for Lively has always been to escape the browser *per se*, and to spawn and manage independent processes anywhere on the Internet. With the introduction of support for WebSockets in browsers beginning in 2012, we started experimenting with new possibilities for Lively. Soon a framework was developed for messaging between active Lively Web sessions, called *Lively-to-Lively*, or L2L. L2L promises to open Lively way beyond its current bounds. We can begin to think not just of worlds of active objects, but an entire connected universe of active objects as simple and malleable as the Lively objects we know today. This offers the possibility to incorporate an Internet of Things with almost nothing new to learn or build. We have already experimented with a real-time interconnection of all Lively users that lets each know (and even see) what they are doing, controlled by various privacy preferences of course.

A Lively-to-Lively message is a JSON object that carries fields for *sender, receiver, selector and arbitrary payload data*. Sender and receiver are UUIDs identifying the Lively
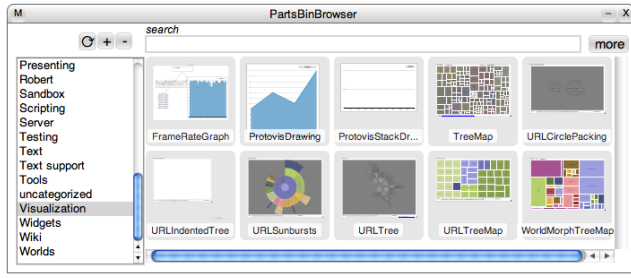
**Figure 6.** Lively PartsBin from 2011, a repository to share live objects. Parts can be taken out, used, explored, modified, and shared again. Unlike in conventional source code repositories, all those interactions are based on direct manipulation of actual objects and not their abstract source code representation to better understand complex applications, tools, and systems only by inspecting their visual representation [21].
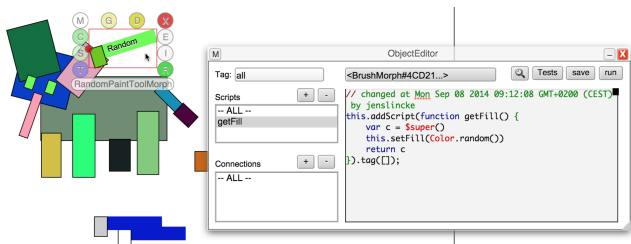


**Figure 7.** Using a Halo context menu, shape and position of Lively objects can be directly manipulated and more complex editors opened. For example, the Object Editor can be used for adding and modifying object-specific behavior as scripts or connections.

component from which the message originates (to allow for answer messages) and to whom which component is sent, respectively. The selector is the name of a method that the receiving side can implement. If such a method exists, it is called with the message data in the receiver's runtime upon message delivery. If no matching method is available, the system automatically returns a does-not-understand answer message. With this direct and extensible messaging scheme, Lively applications a) do not have to limit network-based communication around the HTTP server-client response model, and b) can dynamically add and remove network services as required by applications. By connecting Lively components via both WebSockets and WebRTC channels (allowing direct browser-to-browser connections), the Lively-to-Lively model allows practical peer-to-peer communication. Moreover, since the messaging mechanism is based on a simple JSON format, external systems can be connected to the Lively-to-Lively network as well. Using these facilities, a number of Lively applications such as the user visualization and chat shown in Figures 8 and 9 as well as remote development tools were created.
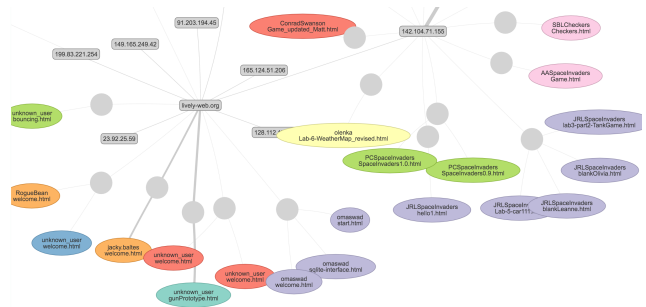


**Figure 8.** Part of a network visualization showing Lively-to-Lively sessions of users and servers.
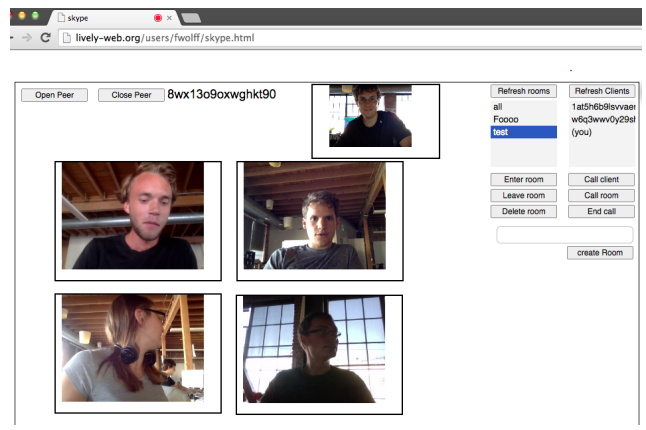


**Figure 9.** Lively-to-Lively group chat experiment using WebRTC.

In parallel to the brief historical summary provided above, Lively Kernel evolved into a number of variants supporting different rendering technologies, such as the Qt framework and WebGL. A more generalized mobile version of the system – *Cloudberry HTML5 mobile phone platform* – was also built at Nokia Research Center [29]. A short history and summary of the different versions is provided in Table 1.



**Figure 10.** Collaborative HyperCard Application [4].

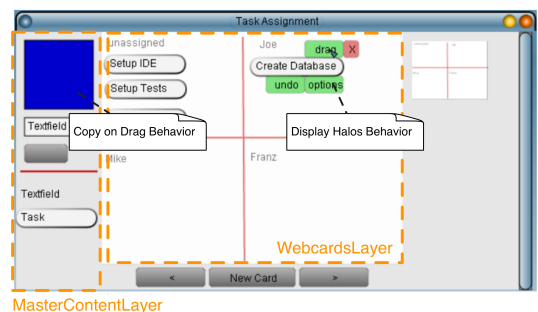| System | Overview | Features | Applications |
|---|---|---|---|
| Lively Kernel 1.0 [12] | The original baseline and showcase implementation demonstrating the feasibility of building a self-contained programming environment running inside a generic web browser (Figure 1). | JavaScript source code browser, reflective evaluation capabilities, lively objects, Morphic graphics framework implemented on top of SVG. | Morphic windowing system [23], Asteroids game, Clock, 3D maze walker, stock viewer, weather widget, and many others. |
| Lively Kernel 1.5: Lively Wiki [15] | Smalltalk-like source code browser that provides a class and method-centered view on top of JavaScript modules. | Self-supporting development, Apache SVN-WebDav used as backend. | CPU Visualization. Quick Brown Fox Game. Fuel and Gas Station Demo (Figure 5). Lively Journal. Development Layers [18]. Web Cards (Figure 10) |
| Lively Kernel 2.0: Lively Webwerkstatt [19] | Fully self-supporting development. New rendering architecture developed inside the Webwerkstatt Wiki. | Create and adapt content, applications, and tools in the same way. Direct manipulation and scripting of objects and publishing them as parts in a shared PartsBin [19]. Runtime adaptatation via Development Layers [18] and Context-oriented Programming [9]. | Presentation Tools (Figure 12). Lively note taking App. Block heat and power plant Simulation and Simulation Environment (Figure 13). Interactive Explanations for Algorithms (Figure 11). Neo4J Query Workbench in D3 Visualization. |
| Lively Kernel 3.0: Lively Web [21] | Implemented a backend service based on Node.js [3] that enables running JavaScript also on the server. | Server-side Lively development through Node.js. Development of Lively Kernel as GitHub Project. Introduction of debugging tools through source code transformation. | Traffic Simulation, Introduction to Programming Courses. |
| Lively for Qt [25] | A variant of the Lively Kernel that uses Qt APIs for rendering and accessing native platform features. | Feature-compliant with Lively Kernel 1.0; SVG replaced with Qt Graphics API. In addition, access to various system APIs enabled via Qt APIs. | Morphic windowing system [23], Asteroids, Clock, 3D maze walker, stock viewer, map widget, various mashup applications. |
| Lively 3D [32] | 3D enabled version of the core ideas of the Lively Kernel implemented on top of WebGL. | 3D rendering engine for Lively applications based on WebGL and 3D libraries that implemented features that are beneficial for 3D visualization. | 3D Golf Simulator, number of 3D arcade games inspired by game consoles. |
| Cloudberry [29] | Fully functional smartphone implemented using HTML5, CSS, and JavaScript, demonstrating the feasibility of a "zero-installation" application platform in the mobile context. | Connectivity features needed for a smartphone, cloud backend to support multi-device operations. HTML5 App Cache leveraged to avoid excessive application (re)loading. | All the generic smartphone applications – even system apps such as the Phone Dialer – implemented as web pages. |

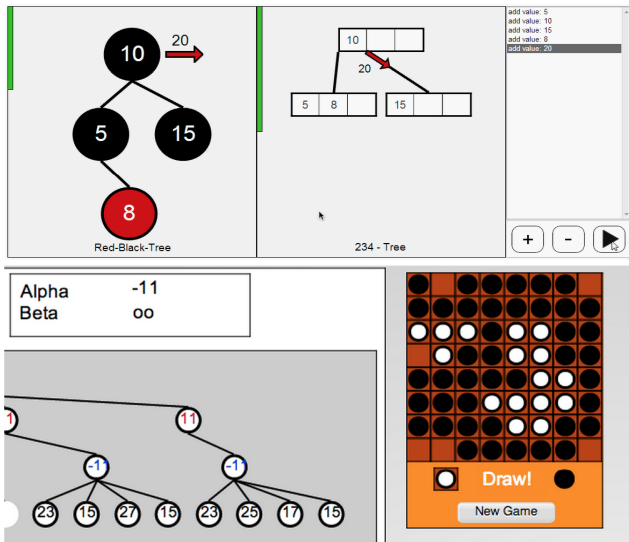**Table 1.** Versions and derivatives of the Lively Kernel.

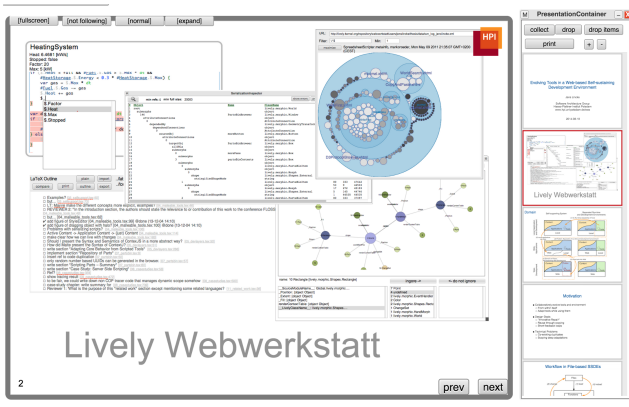**Figure 11.** Interactive tutorial examples illustrating Balanced Search Trees and Alpha Beta Pruning.



**Figure 12.** An interactive presentation tool built using Parts in Webwerkstatt.

## 4. Lively Applications

The evolution of the Lively system made it possible to construct rich, complex applications collaboratively without the users ever having to leave the confines of the web browser. In this brief paper we will not be able to dive deeply into any specific applications. However, we have included a number of interesting applications in screen snapshots sprinkled throughout the paper. Extended figure captions provide some details on those applications.

## 5. Example Capabilities of a Live Object System

We began by stressing that a live object system is more than a web development environment. It is really an operating system for simple (and also complex) components that can easily be composed, scripted and assembled to provide useful services or produce any number of different artifacts. The
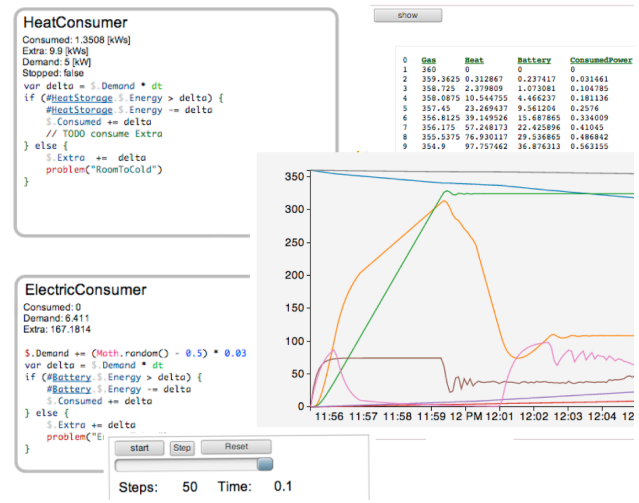


**Figure 13.** A simple simulation environment built from Parts using D3.js for visualization.

critical difference in every case is that the end result remains live, and thus able to meet unanticipated demands. In this section we mention a few actual examples.

**Presentations**. A good example is the Lively presentation at JSConf [10]. Here the system is used to present itself in real time, including the creation of several simple applications and demonstrating simultaneously running simulations and real-time music synthesis.

**Lifting**. Lifting is a term we use for attaching a non-live system to Lively and endowing it with a new dynamic personality. Typically this involves making a communication interface to Lively, and then building a simple user interface for the functions available in the system being lifted. Typically one begins with a simple text pane and the equivalent of a *read-eval-print* loop, and then adds buttons and other enhancements suited to the application being produced. We have done this with *Cyc* (Doug Lenat, http://www.cyc.com), *Virtual Worlds Framework* (David Smith, https://virtual.wf), and a *Clojure* system (http://clojure.org), but the most comprehensive example is a connection to the *R Data Analysis Program* as illustrated in Figure 14.

**Quick real-time visualizations**. A good example of real-time visualization is Figure 15 showing how a few components were wired together to visualize geographic origins of accesses to Lively servers. Another is a visualization of all the active Lively browser sessions in the topology of our servers in Figure 8.

**Serious web/JavaScript development**. The SqueakJS project (https://bertfreudenberg.github.io/SqueakJS/) uses Lively Kernel as the development environment for building a Squeak Virtual Machine in JavaScript [8]. The VM itself does not need Lively Kernel to run, but Lively made it possible to visualize all the state of the virtual machine, essentially providing a Lively debugger for the simulated machine.
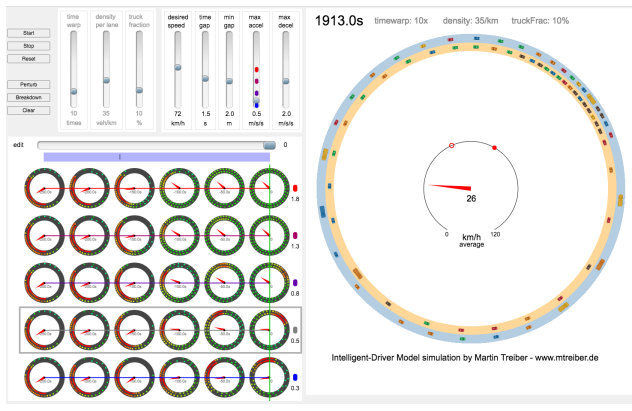
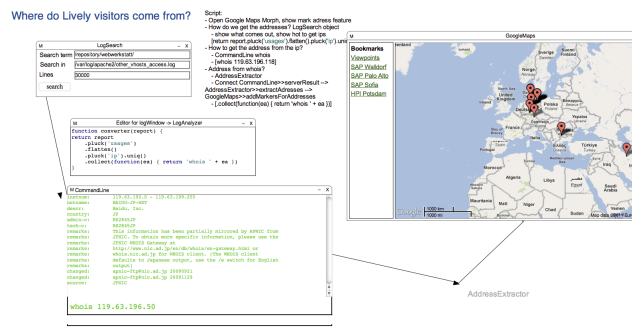**Figure 14.** Simple traffic simulation using the R Data Analysis Program.



**Figure 15.** Mapping Lively users to the globe using Google Maps.

**Collaboration**. Lively provides an interface to both audio and video streaming as shown in Figure 9. We plan to integrate this with the community network (Figure 8) to provide even more straightforward, lively ways to communicate between community members.

**Alternative programming metaphors**. Figure 4 shows Lively Fabrik, a dataflow model for programming Lively applications [20].

## 6. Lively in the Broader Context of Web Programming

From the very beginning, a key difference between the Lively Kernel and other systems in the same area is our focus on *uniformity* – our goal was to build a platform using a minimum number of underlying technologies [27]. This is in contrast with many web technologies that utilize a diverse array of technologies such as HTML, CSS, DOM, JavaScript, PHP, XML, and so on. In the Lively Kernel we attempt to do as much as possible using a single technology: JavaScript. Along the way, we discovered numerous shortcomings of the Web as an application platform [27], discussed in the following.

**Completeness of applications is difficult to determine**. Web applications are generally so dynamic that it is impos-

sible to know statically, ahead of application execution, if all the structures that the program depends on will be available at runtime. While web browsers are designed to be error-tolerant and will ignore incomplete or missing elements, in some cases the absence of elements can lead to fatal problems that are impossible to detect before execution. Furthermore, with scripting languages such as JavaScript, the application can even modify itself on the fly, and there is no way to detect the possible errors resulting from such modifications ahead of execution. Consequently, web applications require significantly more testing (especially for coverage) to make sure that all the possible application behaviors and paths of execution are tested comprehensively. The situation is further complicated by the lack of static verification and static type checking.

**Limited access to local resources or host platform capabilities**. Web documents and scripts are usually run in a sandbox that places restrictions on the resources and host platform capabilities that the web browser can access. For instance, access to local files on the machine in which the web browser is being run is not allowed, apart from reading and writing cookies. While these security restrictions prevent malicious access, they make it difficult to build web applications that utilize local resources or host platform capabilities.

**Fine-grained security model is missing**. A key point in all the limitations related to networking and security the need for a more fine-grained security model for web applications. On the Web today, applications are second-class citizens that are at the mercy of the classic, one size fits all sandbox security model of the web browser. This means that decisions about security are determined primarily by the site (origin) from which the application is loaded, and not by the specific needs of the application itself.

**Nano releases and continuous deployment**. A software release is the distribution of an initial or new and upgraded version of a computer software product. Traditionally, new software releases have occurred relatively infrequently, perhaps a few times per year for a major software product such as a word processor or spreadsheet application, or a few times per month for some business-critical applications in early stages of their deployment cycle. The instant deployment model has now changed all this, allowing new releases to be made dramatically more frequently – even in near real-time. Since the Lively Kernel was one of the very first systems to face this, we had no support from tools and techniques that have later been introduced in the context of Continuous Deployment [17]. Instead, all such complications were handled as a part of the manual development process.

**Incompatible browser implementations; lack and disregard of existing standards**. A central problem in web application development even today is browser incompatibility. Partly this is due to partial implementations of different features in new web-related standards, but in general the sit-

245

uation is much different from, e.g., Java development, where well-defined, organized JCP process was in place to ensure compatibility and standards compliance.

While Lively has evolved, the world around us has not stood still. Let us next take a look at the state of the art in web programming today, reflecting our original objectives and ideas to the present situation in the industry. We will also list some of the work in related areas.

**The Web and the Software as a Service (SaaS) model have redefined personal computing**. Today, the use of the Web as a software platform and the benefits of the Software as a Service model are widely understood. For better or worse, the web browser has become the most commonly used desktop application; often the users no longer open any other applications than just the browser. Effectively, for many desktop computer users today, the browser *is* the computer. The recent VisionMobile developer survey report confirms this observation, citing the following trends in year 2016 [31]:

1. The browser has become the default interface for desktop applications.

2. If the browser isn't used to run the desktop app, it is being used to distribute it.

Based on the points above, it is fair to say that the Web and the Software as a Service model have completely redefined the notion of personal computing in the past ten years. Desktop applications and their deployment model are now primarily web-based.

**Interactive, visual development on the Web has become commonplace**. From the viewpoint of the original Lively vision, it is interesting to note that interactive, visual development for the Web has become commonplace. There are numerous interactive HTML5 programming environments such as *Codepen.io* (http://codepen.io/), *Dabblet* (http://dabblet.com/), *JSBin* (https://jsbin.com/), *LiveWeave* (http://liveweave.com/) and *Plunker* (https://plnkr.co/) that capture many of the original qualities of the Lively vision – such as the ability to perform software development entirely within the confines of the web browser [16].

In addition, there are JavaScript visualization libraries, including designs such as *Chart.js* (http://www.chartjs.org/), *D3* (https://d3js.org/) and *Vis.js* (http://visjs.org/) that provide rich, interactive, animated 2D and 3D visualizations for the Web, very much in the same fashion as we envisioned when we started the work on Lively back in 2006. A central difference, though, is that these new libraries are intended primarily for data visualization rather than for general-purpose application development.

**Web browser performance and JavaScript performance have improved dramatically**. While the original versions of the Lively Kernel ran slowly, advances in web browsers and high-performance JavaScript engines soon changed the situation dramatically.

The emergence of the Chrome web browser and the V8 JavaScript engine kick-started web browser performance wars. Raw JavaScript execution speed increased by three orders of magnitude between years 2006 and 2013, effectively repeating the dramatic performance advances that had occurred with Java virtual machines ten years earlier. From the end user's perspective, today's web browsers are easily 10-20 times faster than ten years ago, and raw JavaScript execution speed (excluding UI rendering) can be thousand times faster. This has made it possible to run serious applications in the web browser, enabling the Software as Service revolution as a side effect.

**HTML, CSS and the DOM turned out to be much more "sticky" than we thought**. The browser and JavaScript performance improvements – while definitely impressive – were not really unforeseen to us. We were convinced that the performance problems of the browser and JavaScript would ultimately get resolved. However, what was unforeseen to us how sticky the original "holy trinity" of web development – HTML, CSS and JavaScript – as well as the use of the Document Object Model (DOM) would be. Our assumption was that software developers would prefer having a more conventional set of powerful graphics APIs instead of using tools that were originally designed for document layout rather than for programming.

Furthermore, when we gave presentations in web developers conferences, reminding web developers of traditional software engineering principles such as separation of concerns and the general importance of keeping specifications and public interfaces separate from implementation details, web developers shrugged and noted that the use of HTML, CSS and JavaScript already gave them the necessary separation. Likewise, the ability to manipulate graphics by poking the DOM tree was seen as a normal way of doing things rather than as something that would raise any concerns.

**The worlds of JavaScript and web programming are highly fragmented**. The number of JavaScript libraries and frameworks has grown almost exponentially in the past years. According to the latest estimates, there are now well over 1100 JavaScript libraries and frameworks available (https://www.javascripting.com/). Interestingly, there is still very little convergence yet, except for some annually changing trends, with some libraries and frameworks gaining momentum one year, only to lose their momentum to newer frameworks some time later. For instance, the once dominant Prototype.js and jQuery libraries are now being forgotten. While Angular.js seemed to capture the most developer mindshare only a year ago, it is currently the React.js ecosystem that seems more in focus.

**Mobile computing is still dominated by apps – for now**. During the original development of the Lively Kernel, some of us were heavily focused on making the system run well also on mobile devices. Although the feasibility of running the system on mobile devices was demonstrated, in

practice mobile devices and browsers were still so slow those days that no serious Lively applications could be built.

It is interesting to note that in the past ten years desktop computing and mobile computing have evolved in entirely different directions. While personal computers are now driven mostly by the Software as a Service model, mobile devices are still dominated by native apps. We claim that today this divergence is driven primarily by user interface needs. While screens of mobile devices have become considerably larger, mobile browser use still is not very feasible because of different input modalities and usage contexts (e.g., using devices while walking, running or while driving a car). In contrast, the other historical reasons for keeping personal and mobile software platforms separate – such as CPU performance, memory and network bandwidth limitations – have largely disappeared in the past years.

Our prediction is that in the next 5-10 years mobile and desktop operating systems will converge. This convergence will be driven by the emergence of multi-device computing environments in which average users will have a much larger number of Internet-connected computing devices in their daily lives. In such environments, the users will expect a totally seamless, *liquid* software experience that allows the users to pick the most applicable device and then effortlessly move to another device (e.g., with bigger screen or larger keyboard) to continue the same activities. A recently published *Liquid Software Manifesto* summarizes our predictions and expectations in this area [28].

## 7. Looking Forward

The computer is the ultimate dynamic medium of our civilization. Able to respond to our touch with billions of operations per second, it is infinitely changeable and always active. Lively is our latest attempt to preserve the power of this dynamic medium and bring it to users in a form that is universally accessible and simple to understand and control.

The field of web development still bears the imprint of pre-web development techniques. Spec the app, write a lot of code, debug it, and hope the users are happy with it. Agile techniques that have softened this rigidity and enabled more flow between implementers and target users, were actually inspired by early live object systems. Furthermore, approaches such as DevOps [5] have made live object systems commonplace, and we are more and more accustomed to systems that never sleep. Examples include massive online games, business information systems as well as sites such as Facebook whose evolution is defined by its users' live preferences [7]. Finally, while the adoption of Single-Page Application (SPA) development style has improved the overall user experience for web sites that want to behave like classic desktop applications, our desire is to simplify and generalize both the goals and how we move to achieve them.

Our wonderfully general computers are growing and changing, and the challenge of live object systems in this time should be to preserve the power of those changes and deliver them all seamlessly and effortlessly to our users, whether they are doing web programming, sketching the design of a water clock, giving a presentation, or playing drums in five places around the planet in real time. While we would be the first to cite the many improvements needed in Lively, we must balance this with a more important need to move forward and explore the world ahead for live object systems.

**Sketching.** Anyone who uses a designer's sketchbook will see it as the destiny of a live object system. Now that we finally have tablets with capable pens, it is time to support the creative flow from idea to sketch to scripts and simulation, and finally to presentation and publication. This should be a primary capability of any live object system.

**Touch.** Touch is a delightful enigma. On one hand it lacks the precision of the pen and the symbolic power of text. Yet on the other hand (so to speak) the finger is the most immediate and intuitive physical extension from the brain. How do we harness that immediacy and give it greater power than merely pushing buttons? The answer lies in extremely dynamic interfaces. Without the symbolic reference power of text, we must provide an array of choices, presented to our visual cortex, so the finger can navigate and press, to provide the next finer choices along a path to the desired tool or object, whatever it may be.

**Tiles.** Tiles appear to be the best adaptation of code to the world of touch. Tiles can represent detailed programmatic constructs as in EToys or Scratch, or they can represent higher level commands which we have used for decades as buttons. In the evolution toward simpler more general concepts in live object systems, program tiles, action tiles, buttons and menu items should surely all be the same kind of object. We are currently experimenting with recording the history of morphic actions in Lively as a sequence of active tiles, and making this available as an on-ramp to programming for end users.

**Collaboration and social media.** Social media are the vehicle for sharing in today's world. Surely any live object capability would add depth to such sharing, and conversely serve to greatly extend the universe of any live object system so connected. So far, probably the best example is *mashware* [24], where code and content from various locations are integrated into a single system on the fly.

**Internet of Things.** A live object system would appear to be the ideal interface to the Internet of Things. It offers the facilities for communicating and interacting with visual proxies for any object, and even integrating visual proxies for things that do not yet exist. Live object systems will have a major role in the *Programmable World Era* in which all sorts of everyday objects will be connected to the Internet and thus effectively have enough computing, storage and networking capabilities to host a dynamic programming interface integrated with systems such as Lively. Beyond just making the World Wide Web more lively, we foresee making

the world around us being a world of "Lively Things", each responsive in an effortless and natural fashion.

## 8. Conclusions

We have shown how a live object system is more than a web development environment. The Lively Kernel became the first fully interactive, self-sustaining web-based software development environment precisely because it began with the more general goals of a live object system. Ten years ago we recognized that these goals could be met by web browsers on the Internet, and it has been gratifying to make it happen. While Lively itself is not widely known or used, it did blaze the path for today's Software as a Service based systems and live web programming more broadly.

The promise of a live object system is to be an active design medium for beginners and experts alike; to empower our individual thought processes and also our communication with others. With many of the hard problems solved, and systems like EToys to inspire us, we can now concentrate on civilizing this instrument to be our most expressive tool from idea to sketch to simulation to presentation and deployment. No sooner do these goals appear within reach than more opportunities such as touch, collaboration and the Internet of Things come along to add more challenges. There is plenty to keep us busy for another ten years!

## Acknowledgments

## References

[1] A. Borning. ThingLab – an Object-Oriented System for Building Simulations Using Constraints. In *Proc. of the Fifth International Joint Conference on Artificial Intelligence*, pages 497–498, 1977.

[2] A. Goldberg, A. Kay, with The Learning Research Group. The Smalltalk-72 Instruction Manual. *Xerox Palo Alto Research Center Technical Manual*, March, 1976.

[3] M. Cantelon, M. Harter, T. Holowaychuk, and N. Rajlich. *Node.js in Action*. Manning, 2014.

[4] J. Dannert. WebCards – Entwurf und Implementierung eines kollaborativen, graphischen Web-Entwicklungssystems fur Endanwender. Master's thesis, Hasso-Plattner-Institut Potsdam, 2009.

[5] P. Debois. DevOps: A Software Revolution in the Making. *Journal of Information Technology Management*, 24(8):3–39, 2011.

[6] D. Ingalls. The Smalltalk-76 Programming System. *Proceedings of the 5th ACM Principles of Programming Languages Symposium*, Pages 9-16, Tucson, January 23-25, 1978.

[7] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[8] B. Freudenberg, D. H. Ingalls, T. Felgentreff, T. Pape, and R. Hirschfeld. SqueakJS: A Modern and Practical Smalltalk That Runs in Any Browser. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS'14, pages 57–66, New York, NY, USA, 2014. ACM.

[9] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3): 125–151, March - April 2008.

[10] D. Ingalls. The Live Web. JSConf 2012, https://www.youtube.com/watch? v=QTJRwKOFddc.

[11] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. *ACM SIGPLAN Notices*, 32(10):318–326, 1997.

[12] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. The Lively Kernel: A Self-Supporting System on a Web Page. In *Self-Sustaining Systems*, pages 31–50. Springer, 2008.

[13] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle. Fabrik: A Visual Programming Environment. *SIGPLAN Not.*, 23(11):176–190, 1988.

[14] A. Kay. Squeak Etoys Authoring and Media, 2005. as of Aug 01, 2005, `http://www.squeakland.org/pdf/etoys_n_authoring.pdf`.

[15] R. Krahn, D. Ingalls, R. Hirschfeld, J. Lincke, and K. Palacz. Lively Wiki: A Development Environment for Creating and Sharing Active Web Content. In *Proceedings of the 5th international Symposium on Wikis and Open Collaboration*, page 9. ACM, 2009.

[16] J. Lautamäki, A. Nieminen, J. Koskinen, T. Aho, T. Mikkonen, and M. Englund. Cored: browser-based collaborative real-time editor for java web applications. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1307–1316. ACM, 2012.

[17] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. The Highways and Country Roads to Continuous Deployment. *IEEE Software*, 32(2):64–72, 2015.

[18] J. Lincke and R. Hirschfeld. Scoping Changes in Self-Supporting Development Environments using Context-Oriented Programming. In *Proceedings of the International Workshop on Context-Oriented Programming*, COP '12, pages 2:1–2:6, New York, NY, USA, 2012. ACM.

[19] J. Lincke and R. Hirschfeld. User-Evolvable Tools in the Web. In *Proceedings of the 9th International Symposium on Open Collaboration*, page 19. ACM, 2013.

[20] J. Lincke, R. Krahn, D. Ingalls, and R. Hirschfeld. Lively Fabrik - A Web-Based End-User Programming Environment.

In *Proceedings of the Conference on Creating, Connecting and Collaborating through Computing (C5) 2009*, Tokyo, Japan, 2009. IEEE.

[21] J. Lincke, R. Krahn, D. Ingalls, M. Röder, and R. Hirschfeld. The Lively PartsBin–A Cloud-Based Repository for Collaborative Development of Active Web Content. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 693–701. IEEE, 2012.

[22] J. Maloney and R. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of UIST'95*, pages 21–28, 1995.

[23] J. Maloney and Walt Disney Imagineering. An Introduction to Morphic: The Squeak User Interface Framework. *Squeak: OpenPersonal Computing and Multimedia*, 2001.

[24] T. Mikkonen and A. Taivalsaari. The Mashware Challenge: Bridging the Gap Between Web Development and Software Engineering. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 245–250. ACM, 2010.

[25] T. Mikkonen, A. Taivalsaari, and M. Terho. Lively for Qt: A Platform for Mobile Web Applications. In *Proceedings of the 6th International Conference on Mobile Technology, Application & Systems*. ACM, 2009.

[26] I. E. Sutherland. *Sketchpad a Man-Machine Graphical Communication System*. PhD thesis, 1963.

[27] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web Browser as an Application Platform: The Lively Kernel Experience. Technical report, TR-2008-175, Sun Microsystems Laboratories.

[28] A. Taivalsaari, T. Mikkonen, and K. Systä. Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture. In *Proceedings of COMPSAC'2014*, 2014.

[29] A. Taivalsaari and K. Systä. Cloudberry: An HTML5 Cloud Phone Platform for Mobile Devices. *Software, IEEE*, 29(4):40–45, 2012.

[30] D. Ungar and R. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA'87*, pages 227–241, 1987.

[31] VisionMobile. Cloud and Desktop Developer Landscape. http://www.visionmobile.com/product/cloud-and-desktop-developer-landscape/. [Online; accessed 5-March-2016].

[32] J.-P. Voutilainen, A.-L. Mattila, and T. Mikkonen. Lively 3D: Building a 3D Desktop Environment as a Single Page Application. *Acta Cybern.*, 21(3):291–306, 2014.