Explorative Authoring of Active Web Content in a Mobile Environment

Conrad Calmez, Hubert Hesse, Benjamin Siegmund, Sebastian Stamm, Astrid Thomschke, Robert Hirschfeld, Dan Ingalls, Jens Lincke

Technische Berichte Nr. 72

des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam | 72

Conrad Calmez | Hubert Hesse | Benjamin Siegmund | Sebastian Stamm | Astrid Thomschke | Robert Hirschfeld | Dan Ingalls | Jens Lincke

Explorative Authoring of Active Web Content in a Mobile Environment

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2013
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Das Manuskript ist urheberrechtlich geschützt.

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2013/6405/
URN urn:nbn:de:kobv:517-opus-64054
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-64054

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-232-2

Abstract

Developing rich Web applications can be a complex job – especially when it comes to mobile device support. Web-based environments such as Lively Webwerkstatt can help developers implement such applications by making the development process more direct and interactive. Further the process of developing software is collaborative which creates the need that the development environment offers collaboration facilities.
This report describes extensions of the webbased development environment Lively Webwerkstatt such that it can be used in a mobile environment. The extensions are collaboration mechanisms, user interface adaptations but as well event processing and performance measuring on mobile devices.

Zusammenfassung

Vielseitige Webanwendungen zu entwickeln kann eine komplexe Aufgabe sein – besonders wenn es die Unterstützung mobiler Geräte betrifft. Webbasierte Umgebungen wie Lively Kernel können Entwicklern helfen Webanwendungen zu entwickeln, indem sie den Entwicklungsprozess direkter und interaktiver gestalten. Zudem sind Entwicklungsprozesse von Software kollaborativ, d.h. Enwicklungsumgebungen müssen so gestaltet sein, dass sie mit kollaborativen Elementen zu unterstützen.
Diese Arbeit beschreibt die Erweiterungen der webbasierten Entwicklungsumgebung Lively Webwerkstatt, so dass diese in einer mobilen Umgebung genutzt werden kann. Die Reichweite dieser Erweiterungen erstreckt sich von Kollaborationsmechanismen und Benutzerschnittstellen bis hin zu Eventbehandlung und Performanzmessungen auf mobilen Geräten.

Table of Contents

1 Self-supporting web-based programming on mobile devices with Lively Kernel 1
1.1 Introduction 1
1.2 Related Work 3
1.3 Approach - Identify Workflows in Lively Kernel 4
1.4 Touchscreen interaction vs. mouse and keyboard interaction 9
1.5 Implementation - Bring Lively's Workflows to Mobile Devices 10
1.6 Results and Future Work 17
2 Handling Touch Events on Mobile Devices for Lively Kernel 20
2.1 Introduction 20
2.2 Events on Mobile Devices 21
2.3 Handling Events in Lively Kernel 24
2.4 Implementing Touch Events in Lively Kernel 29
2.5 Related work 41
2.6 Conclusion and Future Work 42
3 Diffing and Merging of Lively Kernel Parts 44
3.1 Introduction 44
3.2 Identifying the problem areas: An example setting 45
3.3 Approaches and decisions 50
3.4 Implementation 60
3.5 Evaluation 68
3.6 Future work 73
3.7 Related Work 74
3.8 Conclusion 74
4 Design and Implementation of Shared Workspaces in a Mobile and Desktop Environment 75
4.1 Introduction 75
4.2 Approach / Problems 77
4.3 Implementation 84
4.4 Examples and Scenarios 89
4.5 Performance Evaluation 91
4.6 Related Work 96
4.7 Future Work 97
4.8 Conclusion 99
5 Benchmarking Lively Kernel on Various Systems 101
5.1 Introduction 101
5.2 Benchmarking 102
5.3 Profiling 116
5.4 Results of the profiler 124
5.5 Future Work 125
5.6 Related Work 125
5.7 Conclusion 127
A Specifications of test sytems 128
A.1 iMac 128
A.2 Windows 128
References 129

1 Self-supporting web-based programming on mobile devices with Lively Kernel

1.1 Introduction

Lively Kernel is a web application that allows users to create their own applications in the web browser. This includes the users' applications and even the system itself. Lively is a self-supporting web system [30] that means the development environment, e.g. editor, debugger is part of the system and always live. Instead of working on text files containing source code, programmers interact and modify vivid objects.

Lively Kernel provides a user interface that enables users to prototype and create their web applications quickly.

Web browsers are not only available on desktop devices but also on smaller devices such as the iPad, smart-phones and other mobile devices. Due to the limited screen space and different input device paradigms a different approach of user interaction is necessary.

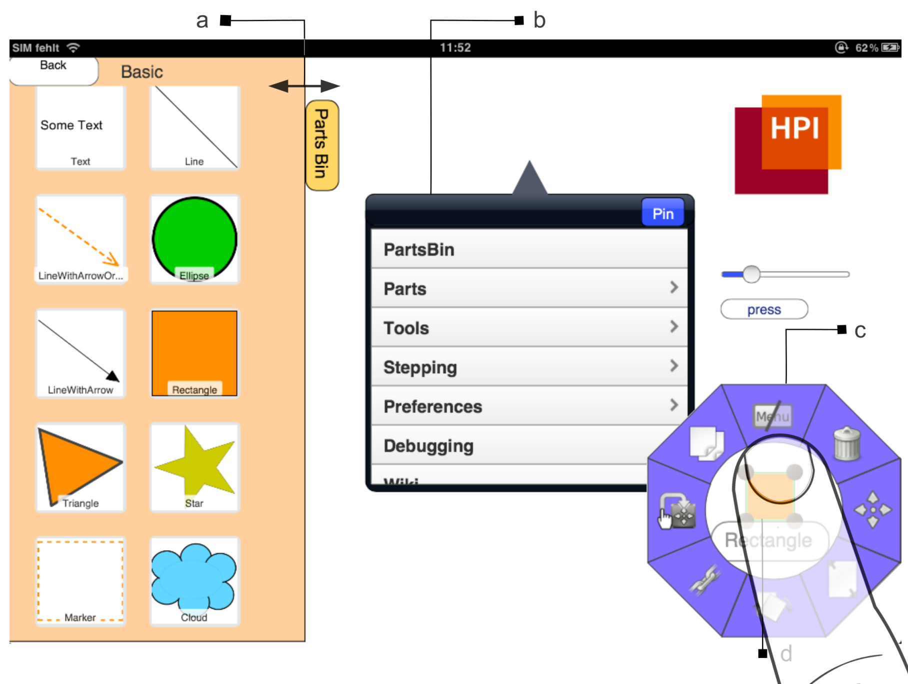


Figure 1: a current Lively Kernel World on an iPad 2 with prominent user interface elements: a) flap b) world menu c) pie menu d) selection

Motivation Today web applications provide a wide variety in functionality typically reserved for native desktop applications such as email, creation of presentation and documents. This is a trend which can be seen in mobile computing as well [58].

Similar to the PC market this is based on the many "good reasons" [37, p. 1] A first reason for this is the inherent platform independence of web apps. In mobile computing this *fragmentation problem* exist across different browsers-specific implementations [58]. This is stressed by the emergence of Sencha Touch, jQuery Mobile and other mobile web development frameworks, which introduce a compatibility layer and offer a uniform interface to web development technologies across different browsers.

Another good reason is that no software installation is necessary to run wherever a web browser is available. Moreover software updates can be performed immediately at one central system. Finally, content of web applications and access to mobile devices is not limited by an application distribution system and other gatekeepers.

The Lively Kernel is built with the following three assumptions: [58]

1. The World Wide Web is the new target platform.
2. The Web Browser is the new operating system.
3. JavaScript is the de facto programming language of the Web.

The goal of this work is to enable software development on Lively Kernel on mobile devices. We focus on how to transform the desktop centric paradigm to a kind of interaction that feels natural to users of mobile devices. For example we replace its extensive use of mouse & keyboard interaction, screen clattering tools with direct object manipulation through touch gestures.

This bachelor's thesis focuses on a workflow oriented approach. First, we identify workflows in the system. On a explorative basis we implement the workflows for mobile devices. We compare the system to other native authoring applications and evaluate its usability.

To enable basic touch interaction a lot of effort was spent on integrating Lively's event system into that of mobile web browsers. Sebastian Stamm describes this in detail in his bachelor's thesis "Handling Touch Events on Mobile Devices for Lively Kernel" (see section 2).

Furthermore mobile devices have the additional problem of syncing and collaboration. Conrad Calmez has worked on that in his bachelor's thesis "Design and Implementation of Shared Workspaces in a Mobile and Desktop Environment" (see section 4) and describes how we enable users to interact and cooperate with each other on mobile devices.

The remainder of this thesis is organized as follows. Section 2 defines workflows which cover essential parts of Lively's toolset. Section 3 elaborates on related work. Section 4 describes differences in touch screen, keyboard and mouse interaction. Section 5 applies known techniques for user interface design for mobile devices. Furthermore it discusses the given alternatives. In Section 6 we discuss the lessons that were learned during the porting efforts and provide ideas for future work. The last section concludes this thesis.

1.2 Related Work

We enabled software development on Lively Kernel on mobile devices. However there were previous efforts to port Lively Kernel to mobile devices. In addition, touch devices are objects of scientific observations for some time. With the rise of the consumer tablet, such as the iPad or the much cheaper Android tablet Nexus 7 as mobile equipment the software market already developed a series of software, covering aspects of Lively Kernel.

In 2009, Mikkonen and Taivalsaari ported an earlier version of the Lively Kernel onto a Nokia N810 [48]. They used two different approaches: regular web browser running in the mobile device and a custom-built native execution environment. Of particular interest were their findings to user interface and user input related issues. In the desktop version of Lively Kernel its common to have many applications and widgets open. They found that a small screen could not easily provide space for multiple parts and applications at the same time. Eventually they used the entire screen for a single application or displayed much less widgets simultaneously.

Much earlier in 1972, Alan Kay foresaw the concept of the *Dynabook* that mobile devices would tremendously alter the way computer are used [33]. Kay's vision inspired many as well as McDirmid [47] who proposed a do-it-yourself programming language called *YingYang* in 2011. It is based on tile and behavior constructs similar to other visual programming approaches, but specialized for touch-based devices. More work has been done for visual programming approaches similar to LabVIEW, Prograph and AgentSheets in general.

Further in 2011 Tillmann et. al. proposed a system called *TouchDevelop* [60]. It supports the user to programm with a structured programming language, which is build using only a touchscreen as input device.

Furthermore, commercial services ermerged. *Codea*[1] allows content creation for the iPad on a iPad, especially prototypes for games and simulations. Codea has one advantage against web-based solutions such as Lively Kernel: full hardware access, e.g. native multi-touch and access to the accelerometer and other sensors[2]. Further there is currently no way for web browsers to access camera, compass, and microphone without external help. More HTML5 related standards emerge, e.g. WebRTC[3] that allows web browsers to use real-time communication such as audio and video chat, etc.

For that purpose where web framework such as *Sencha Touch*[4] and *jquery Mobile*[5] were created. They allow users to program web applications that look and feel like native mobile applications without the hassle to adapt the web application to every platform and browser. *PhoneGap*[6] is a mobile development framework that allows users to use the mentioned missing features.

---

[1] initially released in 2011

[2] Actually there is W3C draft *DeviceOrientation API* `http://dev.w3.org/geo/api/spec-source-orientation.html` (visited 29.06.2012) which gives access to the accelerometer and is supported in iOS since Safari 4.2

[3] Web Real-Time Communication: `http://www.webrtc.org/` (visited 29.06.2012)

[4] Sencha Touch: `http://www.sencha.com/products/touch` (visited 29.06.2012)

[5] jQuer Mobile: `http://jquerymobile.com/` (visited 29.06.2012)

[6] PhoneGap: `http://www.phonegap.com/` (visited 29.06.2012)

The results of these contributions can not be underestimated for our work.

1.3 Approach - Identify Workflows in Lively Kernel

In this section we generalize an approach introduced by Lincke et. al. [43] for creating applications using the PartsBin and existing parts. We illustrate a common workflow for creating and augmenting applications with Lively Kernel. We now describe the overall high-level features of Lively Kernel. Later sections will discuss specific interaction techniques and how they relate to the mentioned high-level application features.

Create active content with Lively Kernel The first step of nearly every applications creation with Lively Kernel involves dragging objects from the PartsBin [43] to the working environment. Working environments are referenced as *worlds* in Lively Kernel. The universe of worlds is called *Webwerkstatt* [37]. Lively implements a system called Morphic[7] where all these metaphors origin from. The PartsBin for example is a web repository for parts in Lively Kernel.

Objects in Morphic, the so-called parts[8] can differ in complexity. A part may have a low complexity and consist of one morph, e.g. a button that changes its background if fired. A part may also have a high complexity such as the PartsBin which is a part itself and consist of 55 submorphs[9]. After dragging a part from the PartsBin, the user can modify and combine it with other parts. The resulting part can be published in the PartsBin again.

Interaction techniques From an input perspective, the interaction techniques used in Lively Kernel can be distinguished into three categories: mouse input techniques, keyboard input techniques and combination of both. Lively Kernel has more than 40 partly undocumented keyboard shortcuts[10].

Create an application using parts As an example we want extent a color chooser by a slider with the blue color component. We shortly elaborate on the current keyboard and mouse based interaction. In Section 1.5 we exemplary show how a similar workflow works on mobile devices.

In general creating and modifying applications can be broadly divided into four parts:

1. *Write* code,
2. *compose* parts,
3. *inspect* the current state of an object and
4. *version control*.

As seen in Figure 2 the user may repeat these steps in any order until the part has the desired behavior and the corresponding tests run successfully.

---

[7] originated from the programming language SELF [61]
[8] visual objects, composition of multiple morphs, active web content
[9] retrieved 27.06.2012 from the category "Basic"
[10] an uncompleted list located at `http://lively-kernel.org/repository/webwerkstatt/documentation/ShortcutList.xhtml` (visited 28.06.2012)

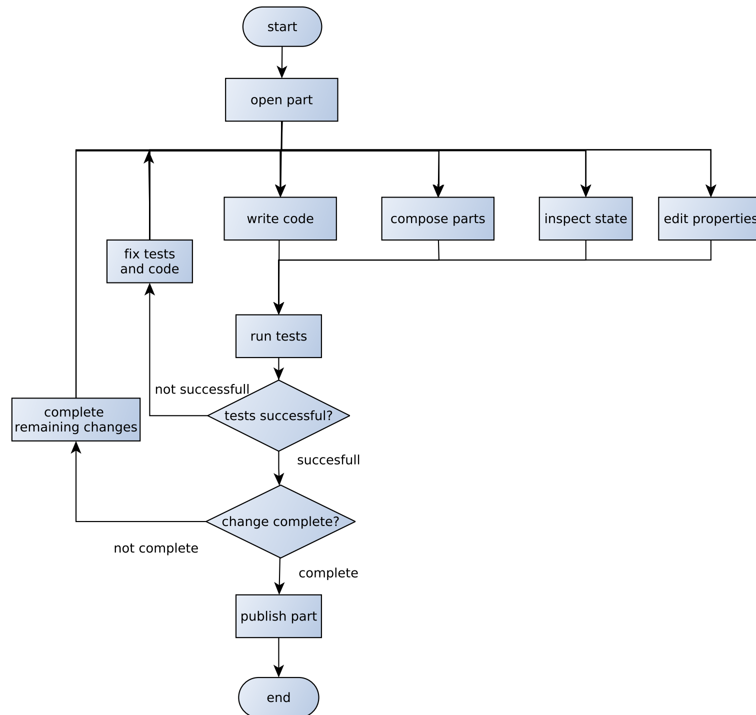


Figure 2: Current workflow for changing and creating an application in Lively Kernel

The first step is to *open a part*. Lincke et al. [43, p. 697] stated that "*creating applications with the Lively Kernel involves combining and augmenting existing objects from the PartsBin.*" However it is also possible to open frequently used parts through the world menu. Although technically this menu contains a series of links to the PartsBin, it differs from a user point of view. Both methods use the mouse as input device. Note that opening a world will deserialize parts in the saved world and make them available as vivid objects again, but as this user interaction – opening a website – is outside the system. We do not consider it further.

As depicted in Figure 3 our user wants to open the part *ColorChooser*. Therefore they open the world menu through a right-click on the world. The PartsBin is the first menu item. In the PartsBin the user enters "ColorChooser" via a keyboard interaction in the search pane. The result list contains the wanted color chooser. While having the left mouse button pressed, the user then drags the PartsBin item onto the world. This interaction typically requires 15 key strokes and four mouse clicks.[11]

[11] measured on the 26.07.2012

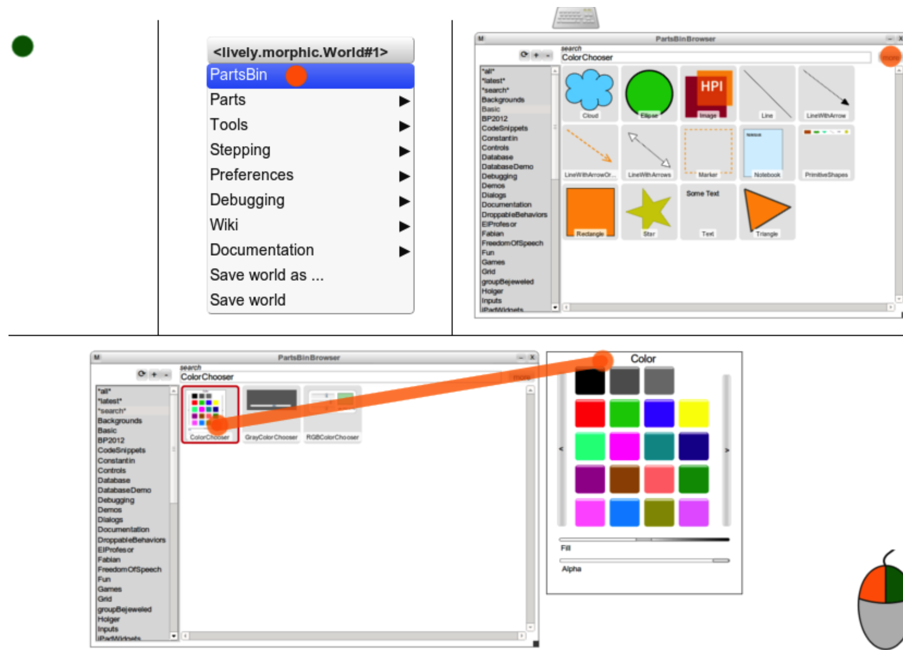


Figure 3: Open the part *ColorChooser* through the PartsBin on a desktop computer

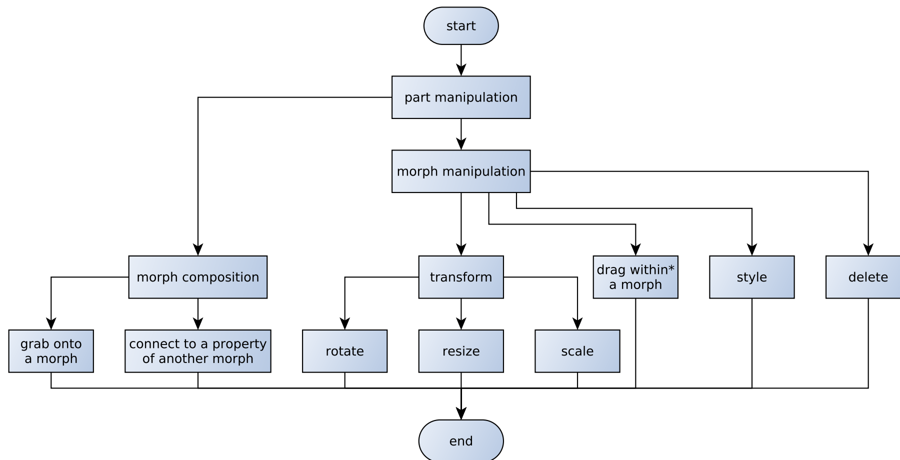


Figure 4: *Compose a part* in detail

The user can use the opened part for four further steps: *Write* new code and extend existing code, *inspect* the parts state, *change* its properties, *combine* the part with another part, e.g. through a concept called connections. Connections link a property change to a script of an arbitrary object which uses the new property value as input[12].

[12] Its also possible to convert the given input and to link properties to other properties directly.

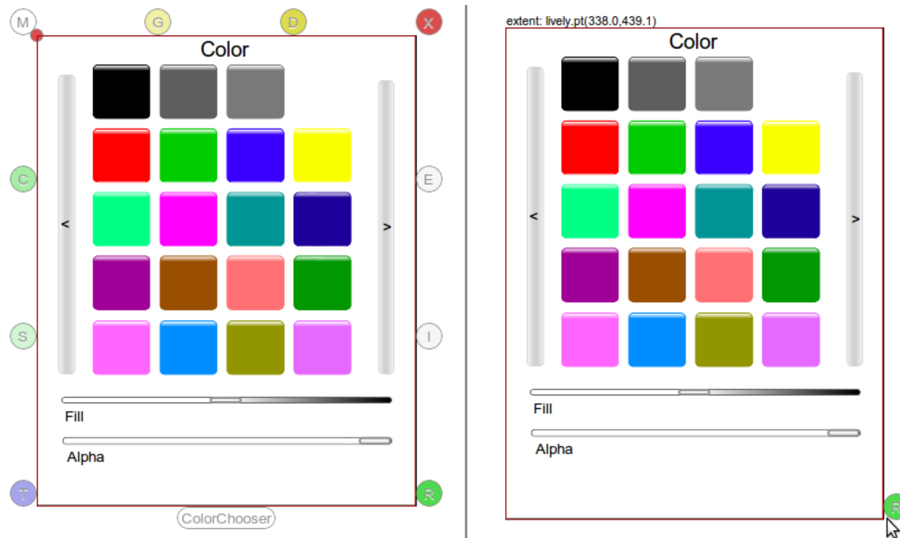


Figure 5: left: Morph composition and manipulation with halos originated from Etoys [3]. right: Resize halo activated

As seen in Figure 4 we categorize part composition in two main aspects: morph *composition* and morph *manipulation*. Both are available through so-called halos a non-standard user interface inherited from Etoys [42] which opens via *cmd + click* or *right-click*.

Morph composition again consists of two possible actions: connect two objects via a connection or drag (G) one morph on another which creates a child-parent so-called submorph relation. Whereas morph manipulation give a range of actions that change the visual representation. The reference to the corresponding Halo in figure 5 is given brackets:

- rotate (T), resize (R), scale (shift + T) change to the visual representation with direct object manipulation through mouse and keyboard interaction
- drag (D) within a morph changes the position without changing the submorph relation. A (negative) offset over the edge of the parent morph is possible. This is the *default behavior* if the left mouse button is pressed over a morph and the mouse is then moved while the button is pressed. Some morph such as *buttons* are not dragable without Halos.
- edit style (S) changes the visual representation of the object as well, but indirect over a style editor. A style editor usually has one static target per instance.
- delete (X) removes the visual representation of the morph

More actions are available over the morph menu (M)

To begin with the ColorChooser the user resizes (R) the morph. Then the user needs to add a new slider. He could retrieve a new one from the PartsBin. Conveniently, its sufficient to copy (C) one of the two sliders, e.g. the color slider. The copy functionality is also available via *shift + drag*

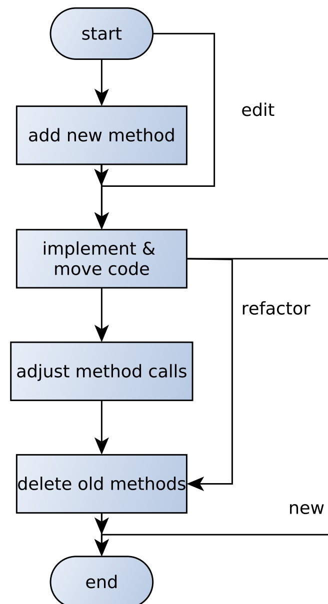


Figure 6: step *code transformation* in detail

Figure 6 shows the process of write and edit in detail. There are two levels of content editing in Lively Kernel: [58]

- *editing objects scripts* with an object editor
- *editing classes* with a system code browser that indirectly change the behavior of the object

The general workflow concept show in Figure 6 equally applies to both, but different tools are used.

In our example case ColorChooser the user has to adjust the color calculation which takes place in the slider.

Furthermore, a reference to the new added slider inside the ColorChooser object has to be established. Remarkably this can be done via the *object inspector*. Its original purpose was to display, change and at best add object properties. However, in the current organic grown implementation it contains a text field to execute arbitrary code[13]. The user first has to select to statement to be executed and then press *cmd* + *d*
Finally as of *version control* the finished part can be published to the PartsBin. This is done via the MorphMenu-Halo (M).

In this section we have seen Lively Kernel's core functionality, which we need to port to a mobile device.

[13] To avoid hidden references Lively Kernel's code convention is to use soft links in the form of `morph.get("anotherMorphName")`

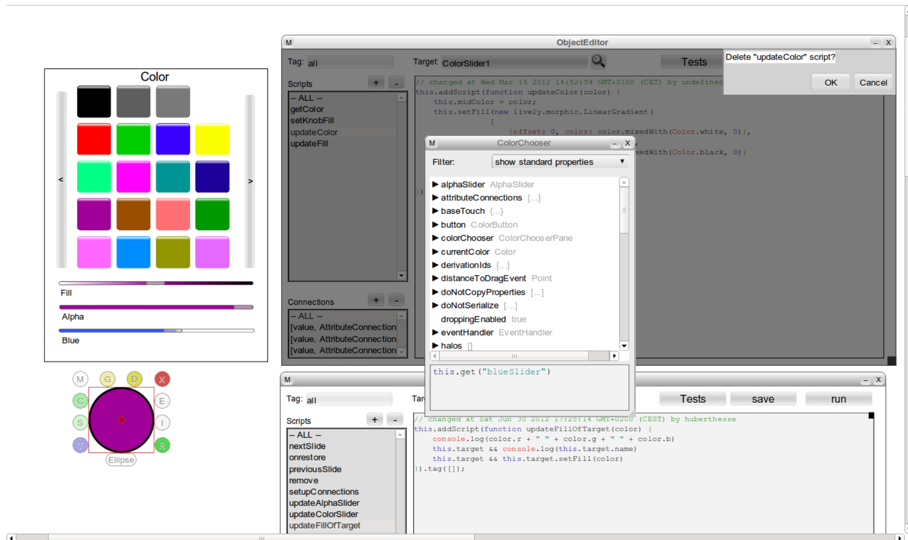


Figure 7: Two instances of the object editor. The one for the ColorChooser is blocked due to an unsaved change. Additionally one object inspector in the middle

1.4 Touchscreen interaction vs. mouse and keyboard interaction

In this section we give an overview of the differences between Lively's current input devices such as the keyboard and mouse, and mouse interaction. Further we classify Lively Kernels user input according to scientific literature.

The literature classifies pointing devices as follows:

– *absolute* positional information, e.g. on a touch screen and *relative* positional information, e.g. with a mouse
– *direct* input, e.g. a touch on a touchscreen and *indirect* input, e.g. move a pointer though a mouse
– *discrete* input, e.g. key presses or *continuous* input, e.g. a mouse movement

Touch screens have been praised in scientific literature: "*Interaction on touch sensitive screens is literally the most "direct" form of HCI, where information display and control are but one surface.*" [2]

Lively's Kernel WIMP (windows, icons, menus and pointing) user interfaces require frequent selection of very small targets. [12] For example, the window resize handles are often to small. Furthermore, fingertips can occlude small targets

1.5 Implementation - Bring Lively's Workflows to Mobile Devices

In Section 1.3 we identified and analyzed existing workflows. In this Section we implement them. After the initial porting of the event system Lively Kernel was somewhat useable on mobile devices. Nevertheless the system was far from intuitive. We had to replace the right mouse click, modifier keys and screen cluttering tools in an straightforward way.

Preconditions Prior to our work we were able to load a Lively page on an iPad with iOS 5 Safari, but no functionality was possible due to the lack of mouse click support of web browser on mobile devices. Lively Kernel a is self-supporting system. Our goal was to the enable the Lively's self-supporting capabilities on mobile devices such as the iPad.

Therefore several issues have been addressed and influenced the following bachelor theses.

- The browser event system and its form on mobile devices. Sebstian Stamm elaborates on that fundamental work in his bachelor thesis *Handling Touch Events on Mobile Devices for Lively Kernel* (see section 2). Without understanding for the event system in different browsers no touch interaction would be possible.
- Mobile devices are suitable for Collaboration. We put a lot of work into collaboration tools. Conrad Calmez concludes in his thesis "the system [we build] is suitable platform for collaboration research" (see section 4).
- Many collaboration features require diffing and merging objects. Astrid Thomschke describes how we did that with Lively Kernel parts (see section 3).
- Performance is critical on mobile devices, because of the reduced computing power and the more direct way of interaction. Benjamin Siegmund compares different platforms in his work *Benchmarking Lively Kernel on Various Systems* (see section 5).

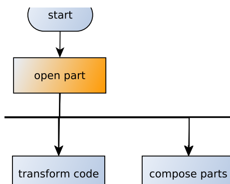


Figure 8: Step *open part*

As seen in section 1.3 Lively Kernel makes heavy use of the right mouse click. We currently use the tap-to-select model with the option to active a pie menu via touch-and-hold. We have experimented with a few other models and decided against them, S. Stamm elaborates in detail on that in his bachelor thesis (see section 2).

Mobile Tools Technically, the step *Open a part* requires the user to choose an object. This object is then deserialized and made available as lively object again. In the simplest case the user chooses an object to open from a textual list. Since all morphs have a graphical representation its helpful to display a small preview of the corresponding morph, because pictures can be more efficient than text[14]. Two main factors constrain our implementation.

First we want to keep a *drag out* of PartsBin metaphor, because users quickly grasp how to use the part then[15]. Therefore we can't use a full-screen application, because we still need space to drag the part to.

[14] according to the Android Design Principles, `https://developer.android.com/design/get-started/principles.html` (visited 29.06.2012)

[15] Metaphors are strongly advised in both Apples iOS [9, Section: Human Interface Principles] and Androids design guidelines [23, Section: Iconography]

Second, it is not trivial to retrieve an image of a part. The current implementation of the PartsBin retrieves an HTML representation of the part and directly inserts that into the visual representation of the part item. This results in a very large HTML document, which takes long to render on the desktop. On mobile devices this takes even longer. Therefore this solution is not acceptable.

To summarize, we want a screen view that consists of one flexible pane for the PartsBin and another pane for the world. The iOS Guidelines suggest a use of the user interface element *Split View* [9, Section: iOS UI Element Usage Guidelines] for that, but here both panes are fixed and cannot be resized. Morphic provides a similar, but resizeable widget for that purpose its called *flap*. Other implementations of morphic, such as Squeak [1] also contain multiple flaps on the edge of every world. Finally, we realized a multi-purpose flap and then adapted a PartsBin for that.

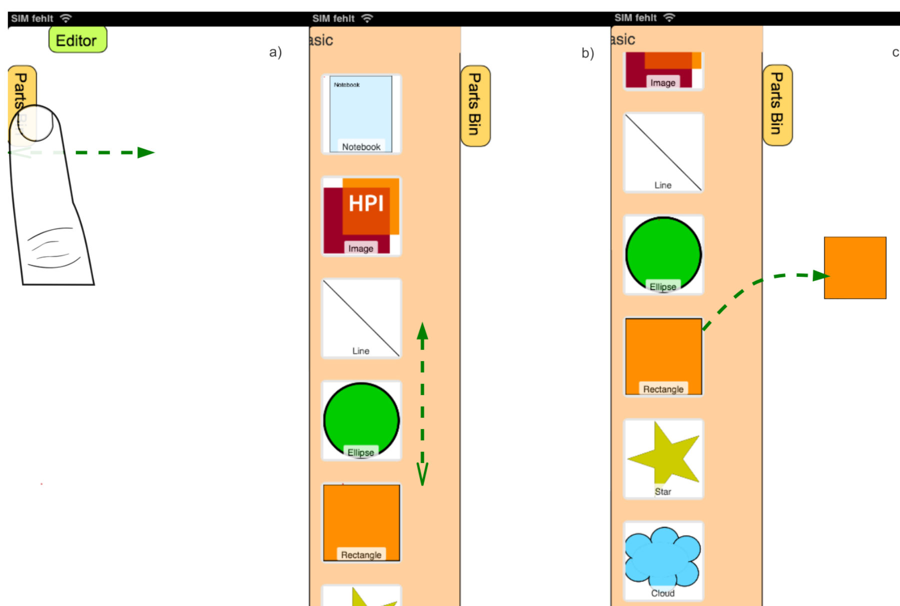


Figure 9: A half-extended flap as alternative user interface are consistent with Lively's background in Morphic and Etoys. Preselected category *Basic*.

In Figure 9 we see two closed flaps. There is a flap handle which can be dragged to the right to open the flap (*a)*). What cannot be seen in the Figure is the selection of the category *Basic* from a single-column list of multiple rows, a so-called *table view*.

Section *b)* shows a partly open flap. Its possible to scroll with a vertical swipe gesture. The third section *c)* shows the PartsBin category *Basic* here its possible to drag a part on the world with a gesture beginning on an item, resulting in an open item. Parts are always the same size if a users drops them out. The PartsBin flap is zoom independent (see Section 1.5)

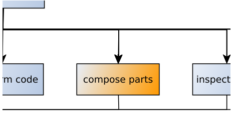


Figure 10: *compose part*

Finally we replaced the HTML based thumbnail through a picture. In order to do this we combined a server-side tool that can take a snapshot of a world [32] with another world which is capable of displaying one specific part. We exposed its functionality through a RESTful[16] API.

Compose parts As seen in Figure 4 there are various ways to combine and interact with morphs on the desktop. A level which was not considered is that we need to distinguish between trigger a default action (on the desktop that's a right-click, a tap on the mobile device) and open halos. In addition, we want to avoid multiple instances of the same tool, e.g. the style editor, opened simultaneously.

Selection That is why we have introduced a selection state that identifies a target for various types of tools. Furthermore, halos and their mobile equivalents are opened only on selected objects. Currently, only one object can be selected at a time. We use a quick tap gesture to set the selection. This was confirmed to a suitable default for application creation by a non-scientific survey in the Lively community and through user testing. In addition, selected objects can be resized with a handle at all four corners as seen in Figure 11. It's also possible to rename a morph through a mobile device friendly rename halo. This allows a much more direct and faster way of manipulation as proposed in the iOS human interfaced guidelines [9, Section: Human Interface Principles].

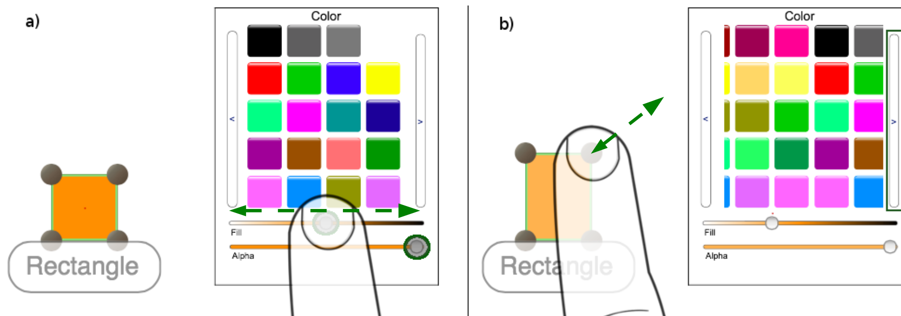


Figure 11: The tool *ColorChooser*: a) selected morph. highlighted magnified slider knob b) selected morph changed color and resized. changed color palette after clicking the right button

[16] REpresentational State Transfer: style of software architecture

Pie Menu Pie menus or gesture menus have the menu items evenly distributed on a circular arc. Pie menus have advantages over normal menus. The big advantage is that an advanced user can learn directions instead of memorizing menu item positions. Instead of thinking of a certain icon, the user learns to swipe in a certain direction, after selecting the morph. Pie menus reduce the search time and lower the error rate by reducing the distance between the menu items and increasing the target size according to Fitt's Law.[17] Pie menus can be significant faster than linear menus after a certain learning phase. [14]

Our implementation is shown in Figure 13. It offers all the actions identified in Section 1.3 (see Figure 4) starting from 12 o'clock clockwise: Morph Menu, delete, drag within a morph, scale, rotate, connect, drag a morph, copy[18]. Additionally, we introduced a new and fast way to connect a morph to a property of another morph. We decided to use pictograms instead of letter to characterize the pie item. Again in order to enforce a metaphor which facilitates the user to use the system. The color is kept simple in order not to distract the eye from the content.

The implementation currently uses the tap-to-select model to select. Then it is possible to activate the pie menu via touch-and-hold. Before a menu item is activated it lights up slightly, so that a beginner can try without actually trigger an action and can abort if necessary.

To sum up:

1. Select a morph to enable Pie Menu Gestures
2. Begin Touch on a morph
3. Swipe gestures invokes actions
4. Pie Menu is shown, when user does not perform any action

Tools Tools do all operations that cannot be accomplished by direct object manipulation. That is, for example, the style editor, or the object editor. We have implemented a number of tools, e.g. the ColorChooser (Figure 11) and the object editor flap (Figure 14). Evidently tools can perform a variety of tasks For the ColorChooser we reduced the color palette to lower reasonable number according to Hick's Law[19] and made the color choices at least fingertip-size, for devices such as the iPad the means at least about 44 x 44 points [9, Section: iOS UI Element Usage Guidelines].

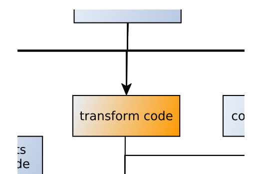


Figure 12: Step *write code*

Write Code In this subsection we describe the work done on the object editor flap as depicted in Figure 14. In Section 1.3 we have seen how the user currently activates a halo to start one object editor for one morph. In Figure 14 our proposed mobile alternative object editor in a flap can be seen. The object editor flap can be instantiated through the world menu (**Tools->Object Editor**).

[17] Fitts' Law [21] predicts that the time required to select an object depends on its size and the distance to the target.

[18] Note that the step *style* is available through three new tools. *resize* is available through the selection handles

[19] Hick's Law [26] describes the time it takes for a person to make a choice is proportional on how many choices there are.

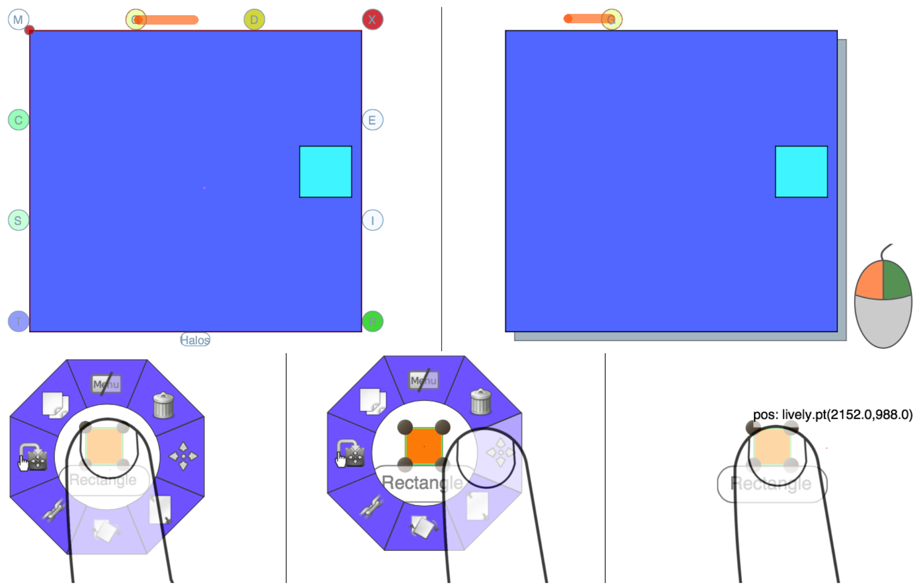


Figure 13: Grab Halo and Pie menu grab in comparison

The functionality of the editor flap covers 3 out of 4 in figure 6 mentioned functions (add new method, implement and move code, adjust method call, delete old method) , but one can't remove existing methods. As we developed the object editor flap as the last tool it is stil incomplete. The object editor flap has *currently limited* capacities to handle connections. Editing classes in order to change core functionality of Lively Kernel is not possible on mobile devices.

To be a really useful object editor the user needs to be disable auto correction and auto-capitalization for text input. A split keyboard i as shown in figure 14 gives better context of the working environment. Unfortunately there is currently no way to do this for a single web page or out of a web page – it must be set system-wide.

While our work the desktop version of the object editor was independently further developed. It is now possible to the change goal of its actions with a mouse click. However programmatically it was always possible to change the target.

Connections Another good example for how workflows differ on mobile devices with touch input and desktop with mouse oriented input. Let's first look at the desktop implementation:

1. Open Halos on Source Morph
2. Open Morph Menu
3. Hover over Menu Entry *connect*
4. Select Source Property
5. Get a Droppable with the Source Property Name

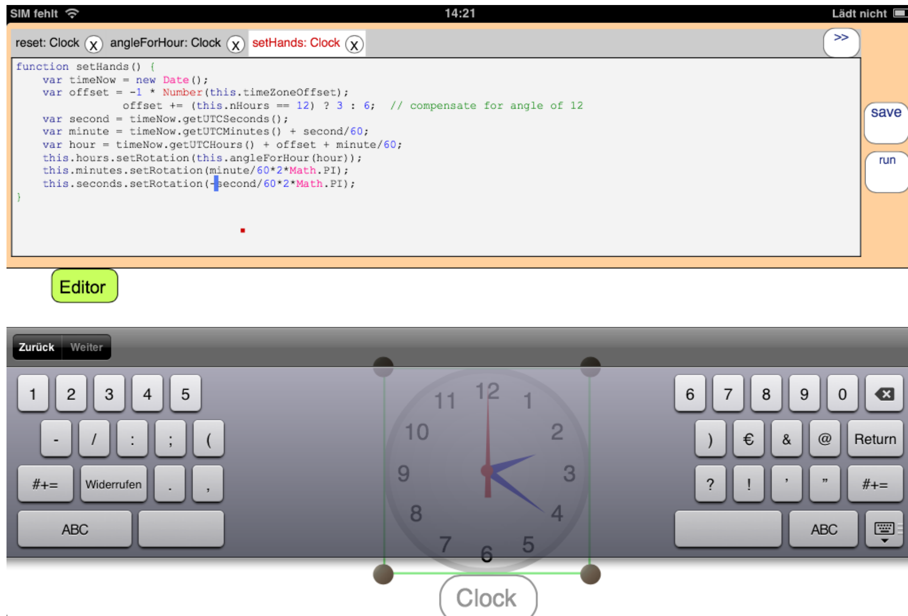


Figure 14: Alternative Object Editor in a flap: use tabs to open multiple scripts. *setHands* is modified so that the clock runs backwards. Available screen space is completely occupied.

6. Drop on Target that accepts Droppable
7. Select Target Property
8. Connection established

The user needs to know what his goal is. Errors while choosing sources and targets lead to a restart. The users must know the properties they want to connect in advance, but sometimes one need to add a script before that. Errors while choosing the property lead to a restart. In addition, the workflow was not possible on mobile devices.

Our idea was to create an extra pie menu item for that purpose. As illustrated in Figure 13 the bottom left pie menu item, connections are represented through a chain. The user first selects the source morph. The target is highlighted while aiming. However it is still possible to change the target, the source and the corresponding properties. The users can refine their choices in a top bar menu through two drop-down lists. If they are ready they hit the *OK* button otherwise they cancel.

1. Open Pie Menu on Source
2. Select Connect Item
3. Drag arrow to Target
4. Top Bar Menu opens
5. Select Properties
6. Press *OK*
7. Connection established

Further Improvements In this section we present minor by-products, which have emerged from our work and have proved to be useful.

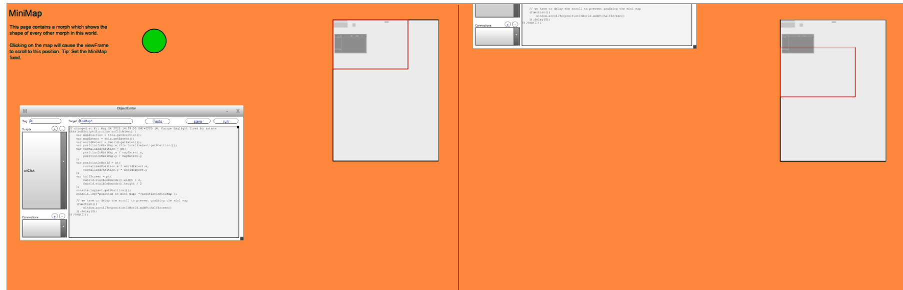


Figure 15: A fixed mini map (for purposes of illustration increased size of factor 1.5)

Fixed morphs respond to the browsers' gesture events, especially zoom and scroll events, because the desktop way a CSS position: absolute it is not advised on mobile browsers. See S. Stamm work for details (see section 2).

The *mini map* is a morph which shows the shape of every other morph in this world. The view frame is highlighted through a red border. Clicking on the mini map will cause the view frame to scroll to the clicked position. The reason we created the mini map for mobile devices are: primarily mobile devices have small screen space and Lively worlds are too large compared to the available screen size. Therefore users have to scroll and zoom through the world. That means it's very likely that the view frame changes its position. But the user has the need for an overview of the world. Finally, the mini map provides a convenient quick jump to a given point functionality.

The motivation for the *Native Lists* imitations was that HTML list boxes don't look good. We want lists that look like the ones in the iOS system settings. As alternative we implemented a list with velocity (snap back on overscrolling) and integrated API[20], so that we can replace the HTML list with specialized lists for Lively. We then replaced the World menu with a nice looking lists as seen in Figure 1. We even improved the lists further by adding CSS transitions to the transitions of the menus. As a subproduct, it is possible to move morphs using CSS transition. Especially with many movements involved that makes a difference in performance on mobile devices, because CSS transitions have hardware-supported graphic acceleration.

Limitations This section is about limitations in the implementation of current system. Any further thoughts will be discussed in section 1.6.

[20] that means here: enable modifications via JavaScript interface, prior lists where read only

As a central component of system on mobile platforms flaps have to be more stable. This includes, among other things: the PartsBin flap has currently no search functionality. The margins between the objects in the PartsBin flap are too large. And if one drags parts from the PartsBin it can happen that parts are not dropped on the world of the world correctly, but remain in a hover state. This can be resolved by another manual click, e.g. through touch and hold.

The object editor flap opens an empty overview, when a morph was selected, but no script opened. The default should rather be an overview of all script of the selected part. Finally it would be an enhancement to allow flaps to snap in different predefined position, e.g. open or half-open.

Finally we can imagine projects were the users need 2 or 3 categories very frequently, but it is not possible to have multiple PartsBin flaps at the same time.

Backports Even we have not made any backports ourself, we were pleasantly surprised when a member of the lively community integrated submenu indicators – small black arrows. We originally developed these indicator for the adjusted world menu and were pleased that idea also fits into the desktop menu morph. That shows we don't need to port fully touch based parts such as the object editor flap to the desktop to improve the overall user experience – small contributions could be just as good.

1.6 Results and Future Work

The Lively Kernel web-based programming environment runs on iOS 5 Safari. Our implementation consists of about 6800 additional lines of JavaScript code[21]. Theoretically, the system should work on other platforms as well, when the event system is adapted accordingly and standards are followed. In practice, porting to other platforms and browsers however, requires some work. (see Section 1.2)

The native look-alike user interface we build allow Lively Kernel programmers to alter and create their Lively application with a mobile device. Yet there is room for improvement, but after this foundation we build, an expansion should be sufficiently easy.

On the other hand, the biggest problems that we have identified with our current solution are as follows:

- Text input was very hard as the font size was very small. In addition there were non-reproduceable bugs, for example it is impossible to write the first character of first the line of an input field on mobile Safari – sometimes. Furthermore Lively's code auto-completion is slow and hardly working on mobile devices.
- Searching items in the PartsBin flap is slow too, because there is no textual search across multiple categories.
- The workflow step *inspect state* is not available on mobile devices. The user can use the normal object inspector, which is difficult to use.

[21] Measures with *cloc* on the 10.07.2012. This includes *core/lively/Touch.js* and our project folder *projects/BP2012*, but excludes parts we worked on

– Editing Lively classes is not possible on mobile devices. On the desktop this is done via the *source code browser* (SCB)
– Fat fingers are a problem for the morphic architecture and touch input. We implemented a specialized solution for slider as seen in Figure11. Other parts also need an specialized or generalized solution.
– Legacy applications with windows are not available as screen space saving on a mobile device version.
– Hard to write a native-look alike application, because very few prefabricated standard user interface for mobile device layouts are present in the PartsBin as they are described in the operating system guidelines
– Editing source code involves opening an object editor flap and following its menus. That is slow at the moment.

We envisioned our user to be someone with software development background and casual experience on mobile device, coming to Lively Kernel. So we made the creation of applications possible. But what if Lively Kernel applications are more used than altered by users on an overall time scale. When the focus of most users of an application shift from creating to using, we clearly have to adapt the application, not to make modifications hard, but to make simply using the application faster and easier. This also applies for some the tools we developed.

For example its fairly easy to select an object using a tap and then alter its code through an *Object Editor Flap* or its style through a *ColorChooser*. However for an user it might be appropriate to use these fast gestures for others actions such as a click and move the selection to a slower gesture such as the long-tap similar to how the Android uses its selections [24] Other measures are also possible: For example PieMenus with less and context-aware options.

Our current prototype is progressing towards an efficient and usable programming environment. Nevertheless, it is still unclear how far the Lively Kernel can be improved on a mobile device.

Below we give some ideas for improvement and further questions:

– Create a complete layer of abstraction from source code such as Etoys hybrid solution or a system tile-based input such as *YinYang* [47] proposed by McDirmid.
– Alternatively adapt and optimize textual input for source code. For example a context-aware code auto-completion is needed.
– At the moment, a user can not create selections with multiple morphs. Some of the alternative mobile tools, we implemented, use the selection as target for their operations. One would have to rethink how tools operate in the future.
– Build a killer application, e.g. a game. Although this not a scientific problem. Lively Kernel lives through its user. This is enforced through one single repository the PartsBin and a single code base. So what is the best way to enhance the system and to attract a creative and passionated users, who will use Lively as their end-user programming environment.
– Event System unification. To attract more users an equal support for Android based devices is necessary. From a software architecture point of view it is interesting here how sophisticated system such as Sencha solve this problem.

– Full multi touch gestures. In the meantime we had one multi touch gesture for resizing a morph. However that was replaced by the resize selection handle. Which multi-touch gestures can be applied into the morphic system easily while maintaining proper user interaction metaphors?

Conclusion The goal of this work was to enable software development on Lively Kernel on mobile devices. We focused on how to transform the desktop centric paradigm to a kind of interaction that feels natural to users of mobile devices. For example we replaced its extensive use of mouse and keyboard interaction, screen clattering tools with direct object manipulation through various touch gestures. For example: the object editor flap, the PartsBin flap and the Colorchooser.

With the increasing use of the World Wide Web as an application platform – the Web Browser as the new operating system – and more web-enabled mobile devices in all price ranges, we added a new quality to code collaboration on mobile devices in Lively Kernel.

As this bachelor's thesis focused on a workflow oriented approach. It first identified workflows in the system. It then implemented the workflows for mobile devices. This thesis compared the system to other authoring applications and evaluated its usability.

In summary, we are looking forward to the development of mobile devices as casual and professional content authoring devices. We think that with the further spread of the Internet mobile devices will step out to be at least as powerful as desktop computer in regard to the web. Differences between mobile web application and desktop web application will vanish, which lead to an universal accessable web.

2 Handling Touch Events on Mobile Devices for Lively Kernel

2.1 Introduction

The Web is an important platform for modern applications. Most of these applications use keyboard and mouse as input devices. These work well on desktop computers, but mobile computing is becoming more popular. Smartphones and tablet computers are outnumbering regular desktop PCs. As mobile devices often have neither keyboard nor mouse, the way users interact with applications is changing. Rather than having two input devices[22], there is only one touch enabled display. This display provides the possibility to recognize and trace multiple fingers simultaneously, which offers new possibilities for user interaction. Applications need to adapt to this paradigm shift.

On mobile devices, we distinguish between native applications (apps), which are developed solely for these medium and Web applications (Web apps), which behave like native apps, but are executed in a Web browser. Native apps are not supposed to work on desktop computers and can take advantage of software development kits (SDK) optimized for multi touch input. Web apps however should work in both environments. Therefore they must implement keyboard and mouse as well as touch optimized methods of user interaction. Most applications deliver distinct content for desktop and mobile devices to solve this problem.

Web apps are executed in a Web browser, which is an event based system. Web developers implement the behaviour of their website based on the events the browser provides. On a stationary desktop PC without a touchscreen interface, these are mainly mouse and keyboard events. On a mobile device, these are touch events. When a website implements handlers for touch events, users can interact with the site on mobile devices. But many websites are not optimized for touch yet, so the mobile browser generates mouse events based on the touch events to enable the user to interact with these sites anyway. Tapping an element will therefore trigger both, a touch and a mouse event. However, it is advisable not to rely on these generated mouse events, as they do not take advantage of multitouch.

Mobile Web browsers also implement multitouch gestures to navigate the site, like pinch to zoom or swipe to scroll. The browser provides events for these interactions, so that developers can react to scroll and zoom events. It is also possible to prevent the page from zooming and scrolling. However, developers must be careful when doing so as users expect a website to behave in a certain way.

[22] keyboard and mouse

Lively Kernel [29] is an online application in which users can create content directly on a webpage. It uses the Morphic user interface to allow direct manipulation of objects (morphs). [45] To do so, it has an event dispatcher to react to user input. During our bachelor's project, we extended the Lively Kernel's event dispatcher to include touch events. We implemented event handler functions to support mobile devices. Beside adding support for native events like `touchStart`, `touchMove`, `touchEnd` and `touchCancel`, we introduced the non standard events tap, doubleTap and hold. These events are not part of the original set of touch events so they are not generated by the browser. To implement these new events we observe the touch interaction based on the standard events and call the handler functions when needed. This adds a more convenient way to react to user input. We also have overwritten some native behaviour like scrolling and zooming to implement features like zoom-independent tools.

The first part of this thesis will cover events on mobile devices. It will explain the internal structure of touch events as well as iPad specific issues. In section 3 we highlight Lively Kernel's event dispatcher and the event flow in Lively Kernel. The main part of this thesis will cover our work on this event dispatcher and our process of implementing touch events. We will examine various approaches to react on user input, different methods to interact with morphs as well as the implementation of our non standard events and zoom independent morphs.

2.2 Events on Mobile Devices

Touch events are essential when working on mobile devices. There are three important touch events: touchstart, touchmove and touchend. Touch events consist of lists of Javascript touch objects. [19] To make sure websites which are not optimized for touch input work on mobile devices, browsers also emit mouse events based on the touch events. This section we will highlight the internals of touch events as well as iPad specific issues when working with them.

Mobile browsing is becoming more and more popular. As more and more people have a smartphone or a tablet PC with an internet browser, a different way to interact with websites comes into focus. Instead of keyboard and mouse interaction, mobile devices offer a touchscreen for user input. Websites need to adapt their user input handling of these new devices.

Lively Kernel is an online framework to build prototypes and other content directly on a webpage. It was developed for desktop usage and uses keyboard shortcuts and mouse interaction such as right click, which are not available on mobile devices. To enable users to use Lively Kernel on mobile devices, we needed to change and enhance the user interaction possibilities to include touch input.

User input on websites is handled with events [39]. For almost every action a users performs on a website, an event is fired. Javascript functions can register for these events and react to the user input. On desktop PCs these events are mostly mouse and keyboard events. On mobile devices, these are touch events.

Internal Structure of Touch Events The touch event interface was originally specified by Apple in their iOS 2.0 release. [54] They defined the different kinds of events for single touch input[23], multi touch input[24] [5, 8], device movement[25] [6] and other device sensors[26] [7].

There is no official standard for touch events at the time of this writing. The specification of the World Wide Web consortium W3C [63] is currently in the candidate recommendation phase. [13] Most mobile devices implement the API Apple has specified. However, this specification does not include events like tap, doubleTap or hold, while there are events like click and dblclick for mouse events. [39]

We think that convenience events like tap or hold would enhance the productivity of Web developers. Based on the given touchstart, touchmove and touchend events we implemented tap, doubletap and hold for Lively Kernel[27].

The basic structure of a single touch event is shown in figure 16.

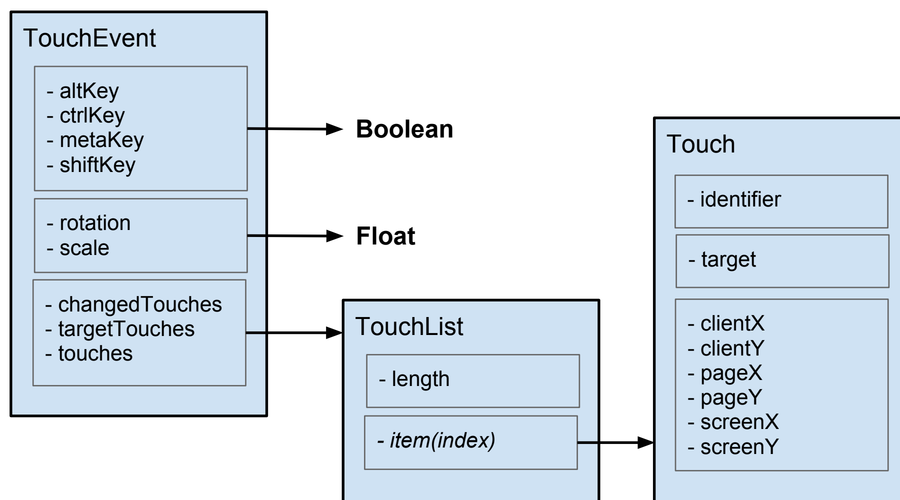


Figure 16: A touch event consists of three TouchLists for all touches, changed touches and target touches. A TouchList is a Javascript object, which behaves like an array. An entry in a TouchList is a Touch object, which contains its target as well as the touch coordinates in various systems

[23] scrolling and swiping
[24] zooming
[25] shaking the device or changing the orientation
[26] compass
[27] see section 4.3

A touch event object consists of three parts: The first part contains information about pressed keys. These keypress information are required by the event standard of the W3C. While the TouchEvent itself is not standardized by the W3C yet, it must still implement the DOM event interface to be counted as an event. That is why the TouchEvent object has these properties even if they are not as usable on mobile devices, as these usually have no keyboard attached. On the iPad with no external keyboard attached these fields will always contain the value false, even if the shift key is pressed or held on the virtual keyboard.

The second part contains information about rotation and scale. They are always 0 respectively 1 when there is only one touch on the screen as a single touch can not perform a zoom or pinch gesture. If there are two or more touches on the screen at the same time, they will contain values describing the pinch/zoom gesture the user made. These values are not associated with any DOM node but describe the global transformation the gesture would apply. There are also gesture events which contain the same information, but are only fired when two or more fingers touch the screen. Since a touch event contains the same information as the gesture event, as well as information about single touches, we suggest using the touch event rather than the gesture event.

The third part is the most valuable part for Web developers. It contains three arrays each containing a TouchList. The TouchLists contain single touch objects. The *touches* list contains all touches on the screen regardless of the position and the target of the touch. The *targetTouches* list contains all touches which started on the element which handles the current event. This is useful for identifying gestures on objects. The *changedTouches* list contains all touches which values have changed since the last touch event.

The touch objects themselves contain an identifier, a target and the touch coordinates in different coordinate systems. The identifier is unique for each touch and can be used for identifying a touch between different touch events. The target of the touch object is the DOM node where the touch started. This target never changes, so that if a user starts a touch on object A and moves to object B, the target of the touch will still be object A.

The touch object itself is persistent between touch events. Developers can save references to a touch object within their application and use them instead of the touchLists of the touchEvents. They can also add custom properties to the touch object itself. We used this to add functions to the touch object which calculate the distance to the start point of a touch[28].

iPad Specific Issues with Touch Events Even though mobile Web browsing is becoming more popular, many webpages do not have a specific handling for touch events. Nevertheless these websites should be accessible and usable with mobile devices. Therefore most mobile browsers emulate mouse events based on the touch input. The iPad triggers mouse events after the corresponding touch events occured, but only if the browser did not do any default action like scrolling or zooming. A touch event sequence is followed by a mousemove, mousedown and mouseup, click sequence. All mouse actions have the coordinates of the touchend event. Figure 17 shows the action of events triggered by a simple touch gesture given that no default behaviour occured.

[28] see section 4.5

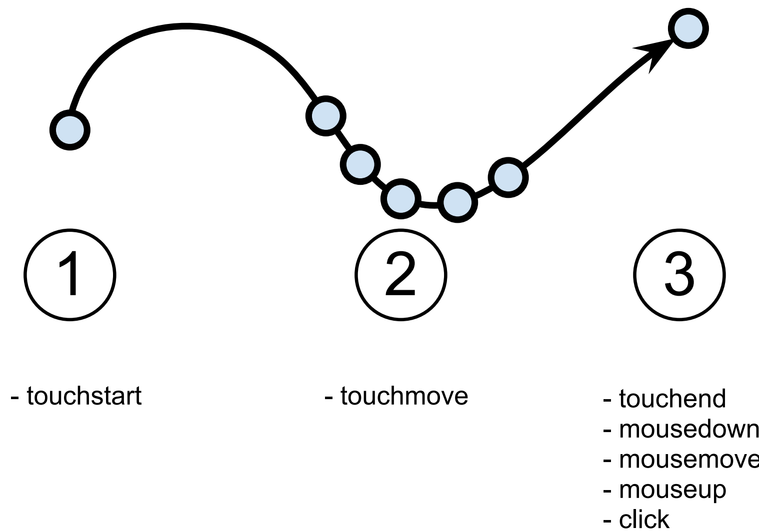


Figure 17: A simple touch gesture produces a series of events: (1) single touchstart event, (2) multiple touchmove events, (3) single touchend event, single mousedown event, single mousemove event, single mouseup event, single click event. The mouse events all have the coordinates of the touchend event.

Since mouse events are only fired if the browser did no default behaviour, their occurence is hard to predict. One way to solve this is to prevent the browser from doing default behaviour. This is done by calling `evt.preventDefault()` on the *touchstart* event. *preventDefault* also disables scrolling and zooming. When zooming and scrolling is not disabled, it is hard to predict when mouse events will be fired. This can lead to a situation where buttons can get triggered twice[29]. For this reason it is not advisable to rely on the simulated mouse events when zooming and scrolling is not disabled. In our work, we disabled the native scrolling to implement scroll independend morphs[30]. The default should not be prevented on editable text fields like input elements as *preventDefault* also prevents the virtual keyboard from opening.

2.3 Handling Events in Lively Kernel

Lively Kernel uses a Morphic user interface. [45] In such interfaces, every object a user can interact with is a morph. Morphs can have scripts which define their behaviour. Scripts with a special name like onClick are treated as event handlers. Figure 18 shows how an event can be registered for a morph in the Lively Kernel.

---

[29] by the touch event handler and by the mouse event handler
[30] see section 4.5

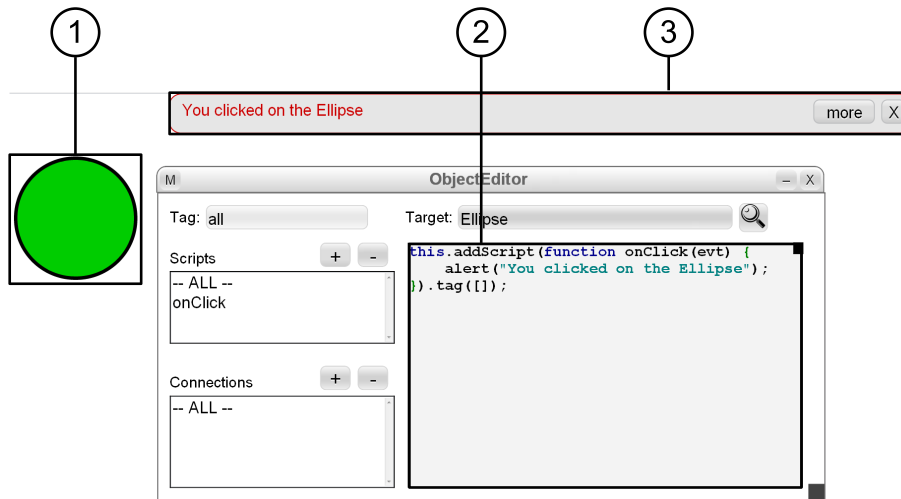


Figure 18: 1: Morph the event is registered on, 2: Javascript event handler function, 3: perceived output when the event occurs

Note that despite the similarities in the syntax of the source code, registering an event on a morph is significantly different from registering an event on a DOM node. For registering an event on a morph, users have to create a script with a predefined name. This name is based on the event names. The name for registering an event on click for example is *onClick*. For touch events, users can simply use *onTouchStart*, *onTouchMove* or *onTouchEnd*. Additionally to the touch and gesture events specified by the Apple Developer Guidelines, we introduced *onTap*, *onDoubleTap* and *onHold*. Like every other event handler in Lively Kernel, these handlers are just scripts on a morph with a special name. We instrumentalized Lively Kernel's event dispatcher to call these functions when needed[31].

Lively Kernel uses its own event dispatcher so that basic morph interaction like drag and drop is still possible when the morph itself overwrites the event listeners responsible for drag and drop behaviour. Figure 19 shows the process of registering an event for a morph in Lively Kernel. We will have a look at each function and analyze its behaviour.

Morph – initialize *initialize* is called by the constructor of the morph. It gets executed whenever a new morph is created. This function implements different cases for object initialization. Initialize itself sets up some basic variables for the morph like arrays for submorphs and scripts. Beside that, it calls *prepareForNewRenderContext* with the default render context (HTML) as argument.

[31] see section 4.3

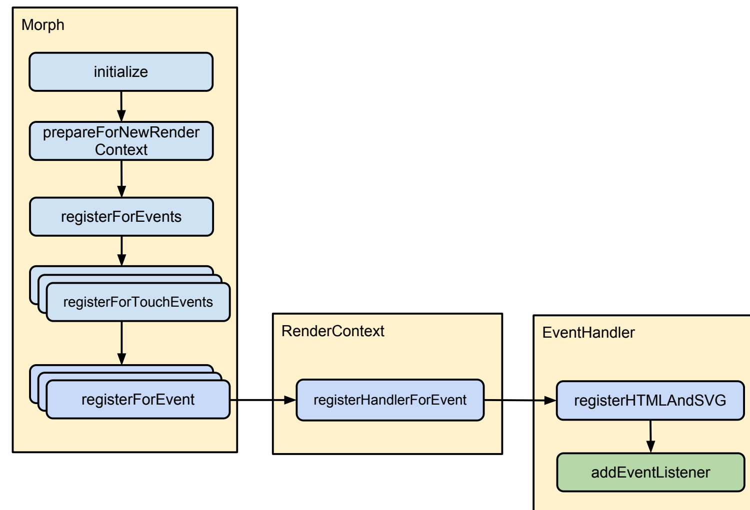


Figure 19: method calls for registering an event

Morph – prepareForNewRenderContext Lively Kernel was supposed to work with render contexts apart from HTML. *prepareForNewRenderContext* calls functions to initialize the renderer and create a graphical representation[32] for the morph. It also calls itself for the submorphs of the morph.

Morph – registerForEvents This method takes the argument *handleOnCapture* as parameter which is defined in the global *Config* object. *handleOnCapture* indicates whether the eventListener should be called in the capturing phase[33] of the event. Per default Lively Kernel uses the capturing phase. *registerForEvents* dispatches the call to multiple registerForEvent methods like *registerForMouseEvents* or *registerForTouchEvents*.

Morph – registerForTouchEvents *registerForTouchEvents* contains calls to *registerForEvent* with DOM event names as argument.

Morph – registerForEvent Every morph has an event handler which contains a dispatch table where all callback functions are stored. This dispatch table is an associative array where the key is the name of the DOM event[34] and the value is an event specification. This event specification contains the type of the event, the target morph, the target method name and the *handleOnCapture* flag. This function creates the event specification object and calls *registerHandlerForEvent* on the render context of the morph.

[32] for the HTML renderer this is a DOM node
[33] rather than the bubbling phase
[34] for example touchstart or mousemove

RenderContext – registerHandlerForEvent This method dispatches the call to different functions of the event handler regarding the render context. In the default case the render context is HTML, so *registerHTMLAndSVG* is called.

EventHandler – registerHTMLAndSVG This methods adds the event listener to the DOM node of the morph. The registered method is not the one the user implemented like *onClick* or *onTouchStart*, but an anonymous function which just calls *handleEvent* on the event handler. Since the event handler keeps its own dispatch table[35] it can call the associated function when the event occurs.

If the user interacts with the site, certain DOM events are fired. These are dispatched by the browser and the registered callback function is executed. In Lively Kernel, this callback function is the same for every event.

```
registerHTMLAndSVG: function (eventSpec) {
    var handler = this;
    eventSpec.handlerFunc = function(evt) {
      handler.handleEvent(evt);
    };
    eventSpec.node.addEventListener(
        eventSpec.type,
        eventSpec.handlerFunc,
        eventSpec.handleOnCapture
    );
}

handleEvent: function (evt) {
    var eventSpec = this.dispatchTable[evt.type];
    if (!eventSpec) {
       return false;
    }
    var target = eventSpec.target;
    if (target.eventsAreDisabled) {
      return false;
    }
    target[eventSpec.targetMethodName](evt);
    return true;
}
```

Code Example 1: process of registering an event handler to a DOM node and calling the user defined method

[35] see Morph - registerForEvent

The event handler of the morph is scoped into the callback function. The *handleEvent* function checks if the event should be dispatched. Morphs have an *eventsAreDisabled* flag. If this flag is set, the actual callback function will not be executed. The *eventsAreDisabled* flag is not to be confused with *eventsAreIgnored*, which is used by Lively Kernel's default actions for events. The *eventsAreDisabled* flag tells the dispatcher, that events for this morph should be discarded. If events should not be ignored, the actual callback function the user wrote will be called.

Lively Kernel uses the capturing phase, consequently events are dispatched top down from the root node to the children. Figure 20 shows an example scene with the associated DOM tree.

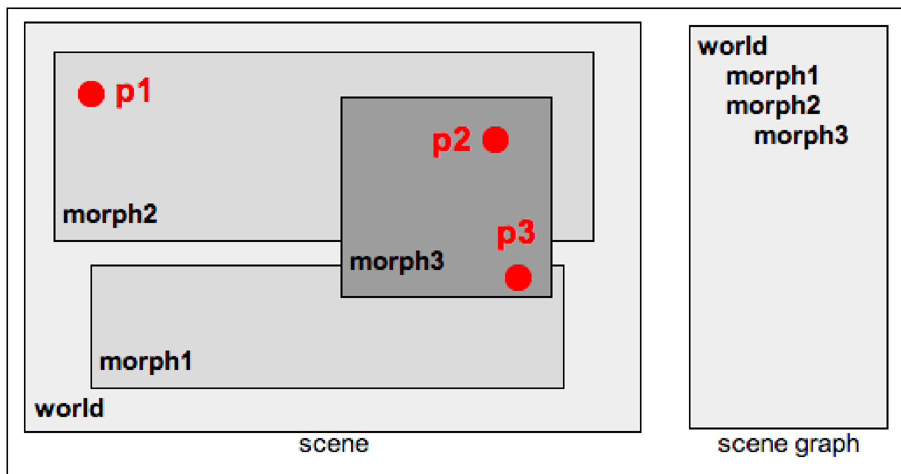


Figure 20: Example scene [36]

morph1 and *morph2* are direct submorphs of the world. *morph3* is a submorph of *morph2*, but its shape overlaps with *morph1*. If an event occurs on position *p1*, the event handlers of the world and the event handlers of *morph2* are called. An event on position *p2* will trigger the handlers of the world, *morph2* and *morph3*. Note that an event on position *p3* will also trigger the handlers of the world, *morph2* and *morph3*, but not *morph1*. *morph1* is only visually below *morph3* on position *p3*, but in a different branch of the scene graph. The path through the scene graph from the root node to *morph3* does not include *morph1*.

If a script needs the morphs under a certain position rather than the scene graph hierarchy, developers can use the function *morphsContainingPoint*. This function does not use the DOM but internal data structures to determine the position of morphs. It will return an array of morphs which include the given point.

2.4 Implementing Touch Events in Lively Kernel

Mapping Touch Input to Mouse Events Before our work, the Lively Kernel loaded on mobile Web browsers, but interactions were limited. We could press buttons, edit text and navigate through the world. These interactions were only possible because of the generated mouse events of Safari as Lively Kernel had no support for touch events back then. The generation of the mouse events was very unpredictable[36]. The best case scenario would be if every morph in Lively Kernel had event handlers for touch events. Most morphs have event listeners for mouse interaction, so we decided to build upon these handlers.

Concept The touch events are similar to mouse events, as there is a touchstart, touchmove and touchend event which are similar to mousedown, mousemove and mouseup. If we fire a mousedown whenever a touchstart occurs, a mousemove on touchmove and mouseup on touchend, we could simulate normal mouse interaction on mobile devices. All these mappings are only performed if and only if there is one touch on the screen. This way users can still zoom and scroll using two fingers. We also prevent the browser from firing mouse events itself so that buttons do not get triggered twice.

Implementation To generate events, Javascript provides the build-in functions *createEvent*, *initMouseEvent* and *dispatchEvent*. This way, we can generate mouse events on touch devices. We wrapped these functions in a *fireMouseEvent* method.

```
1  fireMouseEvent: function(evtType, touchObj, target) {
2      var buttonFlag = touchObj.buttonFlag }| 0;
3      if(buttonFlag === 0 }|
4         buttonFlag === 1 }|
5         buttonFlag === 2) {
6          var mouseEvent = document.createEvent('MouseEvents');
7          mouseEvent.initMouseEvent( /*a lot of arguments*/ );
8          mouseEvent.fromTouch = true;
9          target.dispatchEvent(mouseEvent);
10     }
11 }
```

Code Example 2: the fire mouse event method which creates a mouse event on mobile devices based on the parameters. The button flag is set by the caller and determines whether the event should be a left or a right mouse button event. The variable fromTouch indicates that we generated this event.

[36] see iPad Specific Issues with Touch Events

This method is called if we want to fire a mouse event. A mouse event is always bound to one mouse button (left, middle or right). Only if this button flag is set correctly, we fire the event. We also add the *fromTouch* property. This property is used to distinguish between generated mouse events and mouse events of the browser. We only want our mouse events to be handled, so we discard every mouse event without this flag in Lively Kernel dispatcher.
The right click metaphor for touch events is touch and hold. Unfortunatly we can not predict on touch start if the user is going to do a normal tap[37] or touch and hold[38]. Consequently we can not fire a mouse event on touch start. Instead, we start a timeout whose callback function will trigger a right mouse button down event, if no other mouse event has been fired at this time.

```
1  onTouchStart: function(evt) {
2      var touch = evt.touches[0];
3      touch.buttonFlag = "unknown";
4      var that = this;
5      var touchAndHoldFct = function() {
6          if(touch.buttonFlag === "unknown") {
7              touch.buttonFlag = 2; // right click
8              that.fireMouseEvent('mousedown', touch, evt.target);
9          }
10     };
11     // check for right click after 750 milliseconds
12     window.setTimeout(touchAndHoldFct, 750);
13 }
```

Code Example 3: setting a timeout for right click mapping

Note that we are adding the *buttonFlag* directly to the touch object. Unlike the properties of other DOM events, the touch object is persistent between events. So the corresponding touchend event will include the same touch object as the touchstart event. The buttonFlag property tells the script whether this touch should become a right or a left click. On touch start this is still unknown since we do not know how long the touch will last.
We define a function, which will trigger a mousedown for the right mouse button after 750 milliseconds. We found this timespan by testing the system ourselves. To make sure that we do not fire both: left and right mouse button click for a single touch, we only do this if after the timeout it is still not clear if this touch is equivalent to a right or left click. If we get a touchmove or touchend within the 750 milliseconds timespan, we know it should be a left mouse button click and set the button flag accordingly.

[37] equivalent to left mouse button
[38] equivalent to right mouse button

```
onTouchMove: function(evt) {
    if(touch.buttonFlag === "unknown") {
        touch.buttonFlag = 0; // left mouse button
        this.fireMouseEvent('mousedown', touch, evt.target);
    }
    this.fireMouseEvent('mousemove', touch, evt.target);
}
```

Code Example 4: firing mouse events on touch move

```
onTouchEnd: function(evt) {
    if(touch.buttonFlag === "unknown") {
        touch.buttonFlag = 0; // left click
        this.fireMouseEvent('mousedown', touch, evt.target);
        this.fireMouseEvent('mouseup', touch, evt.target);
        this.fireMouseEvent('click', touch, evt.target);
    } else {
        this.fireMouseEvent('mouseup', touch, evt.target);
    }
}
```

Code Example 5: firing mouse event sequence on touch end

The browser emits mouse events itself based on the touch input, for example if the user presses a button. In our case this would trigger buttons twice[39]. To prevent that, we had to patch Lively Kernel's event handler. We added a layer which discarded the event if the user agent matches a touch enabled device and the event is a mouse event which was not generated by us. We implemented the latter by adding the flag *fromTouch* to the mouse event object we generated.

Evalutation With this implementation, users are able to invoke and use halos and the world menu. They could open the PartsBin, a workspace and other tools. Typing text is also possible. However, using the touch and hold metaphor for right click is not efficient as users have to wait at least 750 milliseconds until a right click is performed. This is especially unsatisfactory as the right click is extensively used by lively kernel for invoking halos. It is not possible to move the mouse without firing a mousedown event[40]. Users could not open the Parts submenu in the world menu. Scrolling through the world with only one finger on the screen is also not possible as this invokes the selection box.

Interaction Techniques for Morphs The implementation of the touch to mouse mapping taught us that there has to be a faster way to invoke halos on the iPad. Touch and hold is the traditional right click metaphor on mobile devices, but since Lively Kernel uses right click a lot, we decided to break this metaphor. Beside touch and hold we implemented and tested two other methods to invoke the halo.

[39] once by our implementation of *touch to mouse* mapping and by the events the browser emitted
[40] hovering

```
handleEvent: function(evt) {
    if(UserAgent.isTouch &&
        !evt.fromTouch &&
        (evt.type == "mousedown" ||
         evt.type == "mousemove" ||
         evt.type == "mouseup" ||
         evt.type == "click")) {
            evt.stop();
            return false;
    }
    cop.proceed(evt);
}
```

Code Example 6: discarding mouse events on mobile devices which were not generated by us

The first implementation was to simply tap the morph to open the halo. This has the advantage, that the interaction is very fast and allows quick manipulation of the morph. The drawback of this method is, that a left click on a morph is often used to perform actions like triggering a button. When a tap invokes halos, users can not trigger buttons anymore without toggling the halo. Tap also opens the virtual keyboard for text editing on the iPad, so we had to disable all text editing in order to make the halo work on text. This was not acceptable.

To fix these issues, we implemented double tap to open halos. Users need to tap the morph twice in order to interact with it. This is similar to performing a double click on a desktop PC. With this implementation buttons can still be triggered by a tap and text can still be edited. A double tap is not significantly slower than a single tap and much faster than touch and hold. However, we needed to set a threshold for the delay between the first and the second tap. This delay describes the amount of time which is allowed to pass between two taps until it is not interpreted as double tap anymore. In our current implementation this is set to 250 milliseconds. We found this timespan by testing the system ourselves.

In our first implementation we took care of the timekeeping ourselves. We took the time at each touchstart event handler and compared them in order to find out if the two events occured within 250 milliseconds. However, the touchstart event handler script could take more than 250 milliseconds itself to do expensive operations. Since Javascript is noninterruptive the next event can only be handled after these expensive operations are finished. [62] This leads to the case where we could not interact with the morphs anymore, because our double taps were not recognized as double taps when the site was under heavy load. We fixed this issue by using the timestamp attribute of the touch events.

Pie Menus as an Alternative Way of Morph Interaction In this section we will show how pie menus can replace the halo items for morph interaction on the iPad. A pie menu is a two-dimensional, circular menu. [28] With a swipe in one direction one menu entry is selected. We will show that halo items are not suitable for mobile devices and that pie menus offer a faster and more direct way to interact with morphs.

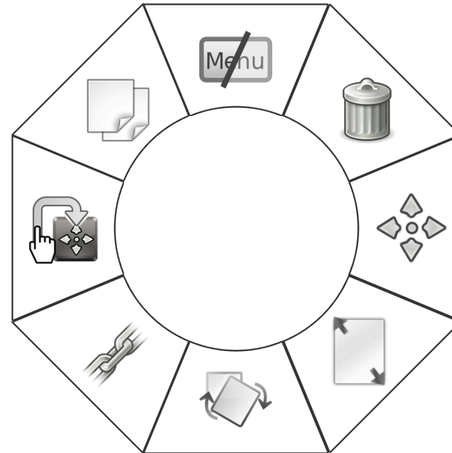


Figure 21: A pie menu with eight menu entries, moving towards an entry will highlight it, when the touch passes the defined section for the entry the associated method will be called.

Using tap events, we can easily implement methods to interact with morphs. On desktop PCs users interact with morphs using the halo items. They use right click to open the halo and left click to activate the halo items. On mobile devices only touch interaction is possible. In the previous sections we described three ways to select a morph. For the implementation of pie menus we use a single tap to select all morphs except text and buttons. These two kinds of morphs are selected via double tap. Buttons are triggered on single tap and texts can be edited when tapped. This has technical reasons. We can not trigger the virtual keyboard of the iPad with Javascript. Instead it opens automatically when an editable morph is tapped. Consequently we can not use a single tap to select text.

The problem of the halo items is that they are too small for touch interaction. The Apple user interface design guideline states, that "The comfortable minimum size of tappable UI elements is 44 x 44 points." [10] On a desktop PC the size of UI elements does not matter so much as users can aim precisely with a mouse. On mobile devices, users interact with their fingers, which are not as precise as a mouse pointer. They also can not see what is below their finger. To solve these problems we increased the size of the halo items to make them work on the iPad. This worked, but through the bigger items, the halo itself became bigger and more distractive. Furthermore it was problematic to arrange all halo items around a small morph.

At that point, our project partner Dan Ingalls suggested the use of pie menus. They are a completely different approach to user interaction. Instead of clicking a visible element on the page, users perform a swipe in a specified direction to select the entry. This form of interaction is much more suitable for the iPad as the user does not need to click elements. Another advantage is, that the pie menu does not have to be displayed in order to interact with it. Users can perform the gesture without the interface by remembering the arrangement of the menu items. If users do not know how to interact with morphs on mobile devices, they will naturally touch the morph. If they do not perform a gesture within a certain timeframe, the pie menu is shown. This way they can see in which direction they have to move to trigger the desired behaviour.

Experienced users however can perform the gesture without the menu because they already know which direction triggers which action. In contrast to halo items, we don't have to create and render the pie menu, what makes the whole application more responsive.

Implementation of Custom Events In Lively Kernel, event handlers are registered by adding scripts with special names like *onClick*, *onMouseDown* or *onMouseUp* to a morph. These scripts are registered by the event handler. We wanted to do the same for the touch interface, so we extended the event handler to include *registerForTouchEvents* and *registerForGestureEvents*. These methods work like the other *registerForEvents* methods[41].

The standard event interface for touches includes only touchstart, touchmove, touchend and touchcancel. We want to provide convenience methods like *onTap*, *onDoubleTap* and *onHold*, which are not part of the original touch interface, but very useful for efficient programming. To implement these methods we could not use *registerForEvents* as it just calls the standard addEventListener method, which only recognizes the official event interface.

Instead, we assign default event handlers to every morph, which call the *onTap*, *onDoubleTap* and *onHold* methods when needed. To do so we have to manage the touch events. This is done by event handlers, which use the standard DOM event interface (touchstart, touchmove, touchend) and decide if the series of standard events fulfill the requirements for a custom event. The following code example shows the standard handler for the touchstart event:

The default event handler for the touchstart event sets some variables on the morph to make sure that the tap events are triggered correctly. Note that the handler is called *onTouchStartAction* instead of *onTouchStart*. This way users can not overwrite our default function when they implement their own onTouchStart event listeners. This corresponds to the implementation strategy which is used for morph interaction with the mouse.

We assign a *tapTouch* property, which holds a reference to our touch object. This touch object is persistent between touch events[42]. We need a reference to this touch to check when the touch started and if the user moved the finger. A tap event should not be triggered if the user made a fast flicking gesture. The moveTouch property is used for our implementation of scrolling[43].

[41] see Handling events in the Lively Kernel
[42] see Events on Mobile Devices
[43] see Implementation of Zoom Independent morphs

```
onTouchStartAction: function (evt) {
    if (evt.targetTouches.length === 1){
        this.tapTouch = evt.targetTouches[0];
        this.moveTouch = evt.targetTouches[0];
    }
    if (evt.touches.length === 1) {
        $world.scheduleHoldIndicatorFor(this);
    }
    if (typeof this.onTouchStart === "function") {
        return this.onTouchStart(evt);
    }
}
```

Code Example 7: touchstart handler for tap events

If there is only one finger on the screen we also want to schedule a hold indicator for the touched morph. The hold indicator is a morph, which indicates the necessary timespan to trigger a hold event. The hold indicator is loaded from the PartsBin when the world is loaded and is then used as a prototype for every hold interaction. There can only be one hold indicator at any given time, so it is loaded only once. The hold indicator itself takes care of calling the onHold function on the morph it is assigned to.

Lastly we also want to call the user defined function onTouchStart if it exists. We make sure that we return the value the user defined script returns. This is important because this return value is used within Lively Kernel's event dispatcher.

The following code shows the default handler for the touchend event:

```
onTouchEndAction: function (evt) {
    var out = false;
    if (typeof this.onTouchEnd === "function") {
        out = this.onTouchEnd(evt);
    }
    if (this.tapTouch &&
        evt.changedTouches.include(this.tapTouch)) {
        this.checkForTap(evt);
        this.cancelHold();
    }
    return out;
}
```

Code Example 8: touchend handler for tap events

When the touchend event occurs, we first call the user defined *onTouchEnd* script of the morph if it exists and save its return value. We do the *onTouchEnd* call before we check if we must call any *onTap* event handler to meet the mouse metaphor. The standard mouse event sequence is *mousedown*, *mousemove*, *mouseup*, *click*. Similar to this structure we want our event sequence to be *touchstart*, *touchmove*, *touchend*, *tap*. That is why we call *onTouchEnd* before the tap function. Nevertheless we want to return the user defined return value so we have to save it locally.

Afterwards we check if the morph has a tapTouch assigned and if this touch has changed in the event. We do this to ensure that a tap can only be performed with exactly one finger on the morph. If this is not the case, the user performed a gesture and we do not trigger tap events. If the touch qualifies as tap touch, we cancel the hold indicator and call a function which checks if the touch fulfills the tap requirements:

```
1 checkForTap: function (evt) {
2     var delta = this.tapTouch.timeFromStartToLastUpdate();
3     if (delta <= 200 &&
4         this.tapTouch.getScreenDeltaToStart().r() <= 25) {
5         this.tapped(evt);
6     }
7 }
```

Code Example 9: checkForTap function which checks whether the touch input fulfill the tap requirements

This function checks if the tap was fast enough to be interpreted as a tap. To do so we use the *timeFromStartToLastUpdate* method, which is added to each touch event by the *patchTouchEvent* method. This method uses the timestamp attribute of the event itself. This way it is independend from the actual load of the page. If the touch lasted no longer than 200 milliseconds and the finger moved no more than 25 pixel, we call the tapped method of the morph.

We now know that a tap has occured, but it could be the second one in a row. In this case we want to call *onDoubleTap* rather than *onTap*. We decided that in the event of a double tap, only the *onDoubleTap* method is called and not both[44]. Users can still get the other behaviour by calling *onTap* in the *onDoubleTap* method themselves.

We defined a maximum amount of time which is allowed to pass between two taps. If we had a tap on this morph before and less than 250 milliseconds passed since then, *onDoubleTap* is called if it exists. Otherwise we call *onTap* and set the *lastTap* property.

[44] onDoubleTap and onTap

```
tapped: function (evt) {
    var doubleTapTimeout = 250;
    if (this.lastTap &&
        new Date() - this.lastTap <= doubleTapTimeout) {
        if (typeof this.onDoubleTap === "function") {
            this.lastTap = false;
            this.onDoubleTap(evt);
        }
    } else {
        if (typeof this.onTap === "function") {
            this.onTap(evt);
        }
        this.lastTap = new Date();
    }
}
```

Code Example 10: tapped function which calls onTap respectively onDoubleTap

Implementation of Zoom Independent Morphs This section describes how we implemented zoom and scroll independent morphs. With this functionality, we can build tool morphs, which modify the currently selected morph. This way we only need one tool for each purpose instead of one tool for each purpose and morph. Having less tools on one page saves screen space on mobile devices.

Motivation The iPad has a native implementation for zooming and scrolling. To do this fast, it stops the rendering of the page. This means that while the users scrolls the page, positions of morphs can not get updated. It is not possible to set the position of a scroll independent morph when the user scrolled or zoomed. Normally this behaviour can be achieved by using the CSS attribute position fixed, but this does not work on mobile devices. [35]
Furthermore the browser does not create an event for scroll interaction. All events we get are touchstart, touchmove, touchend as well as gesture events. So we decided to implement scrolling ourselves and not use the native iPad implementation. That has the advantage, that we can control every phase of the scrolling and react to the user interaction by changing the position of fixed morphs appropriately. The drawback is, that if we do this with Javascript, it is slower than the native implementation by the iPad. With scrolling the performance was still okay, but we could not implement zooming ourselves because the permanent redraw of the whole page made smooth zooming impossible. There are no explicit events for zooming. We utilized the gesture events, which are only fired when two or more touches occur at the same time. This is exactly the gesture which invokes the zooming. Consequently we can just use the gesture events to react on the zooming.

Implementation of scrolling To implement scrolling we added a default touchmove handler to every morph.

```
onTouchMoveAction: function (evt) {
    if (evt.touches.length === 1 &&
        this.tapTouch &&
        evt.touches[0] === this.tapTouch) {
          var delta = this.tapTouch.getScreenDeltaToStart();
          if (delta.r() > 25) {    // not hold
              this.cancelHold();
          }
    }

    if (evt.touches.length === 1 &&
        this.moveTouch &&
        evt.touches[0] === this.moveTouch) {
          this.moveToTouchPosition(evt);
          evt.stop();
    }

    if (typeof this.onTouchMove === "function") {
        return this.onTouchMove(evt);
    }
}
```

Code Example 11: default touchmove handler for every morph

We have to be careful with our implementation of tap events, especially the hold event. If the user scrolls the page, the touch often lasts longer than the timeframe necessary to trigger the onHold function. So we have to cancel the hold if a tapTouch exists[45] and the touch has moved more than 25 pixels. This threshold is necessary to make sure that users do not not accidentally cancel the hold by moving their finger. To determine how far the touch has moved, we use the *getScreenDeltaToStart* function of the touch object. This function is patched to every touch in the emphpatchTouchStartEvent method.
If the morph has a *moveTouch* assigned, *moveToTouchPosition* is called. The *moveTouch* is set in the *onTouchStartAction* handler[46]. As always, if the user defined an ontouchMove function, we call it and return its return value.
The following method is called when the touch qualifies as scroll touch. It checks if it reaches the threshold to initialize respectively emulate the scrolling.

[45] that means that the hold is scheduled, see Implementation of Custom Events
[46] see code example 7

```
moveToTouchPosition: function(evt) {
    var delta = this.moveTouch.getScreenDeltaToStart();
    if (this.scrolled || delta.r() > 25) {
       // scroll the world
       if(!this.scrolled) {
           $world.initializeBrowserScrollForTouchEvents(
               this.moveTouch.startTouch);
       }
       this.scrolled = true;
       $world.emulateBrowserScrollForTouchEvents(this.moveTouch);
    }
}
```

Code Example 12: Function to check for the scrolling threshold. It triggers emulated scrolling if this threshold is reached.

In this function we check if the touch has moved more than the 25 pixel threshold. If this is the case, we initialize the emulated browser scrolling algorithm and call the emulate method, which does the actual scrolling. We also set a scrolled field on the morph. We need this because *getScreenDeltaToStart* only returns the absolute distance from the start of the touch to the current position. If the touch passes the 25 pixel barrier and then goes back near its starting position, `delta.r()` will return something smaller than 25, but the scrolling must not be canceled. Therefore the information, that the touch has passed the 25 pixel theshold once is stored in the scrolled field of the morph.
The following methods implement the calculation of the scroll position as well as the Javascript call to set the viewframe.
initializeBrowserScrollForTouchEvents sets some fields on the world which are required to calculate the scroll position later. It also sets the *emulatedScrolling* flag of the world to true. This field is never read, but scroll independent morphs connect to this field so they are notified when it changes. It is reset to false in the *onTouchEndAction* of the morph.
emulateBrowserScrollForTouchEvents calculates the target scroll position based on the current touch position and the fields we set in the initialize method. It uses `window.scrollTo` to set the viewframe of the page. This call produces an window scroll event, which we use to update the *scrollOffset* variable of the world.
For zoom events we utilize the gesture events of the browser to set the property *zoomingInProgress* in the same manner as *emulatedScrolling*. When the zooming is done, the zoomLevel of the world is recalculated and stored in the *zoomLevel* property of the world.

Implementation of fixed Position and Scale Now that we have a notification when the world scrolled and zoomed, morphs can connect to the property and implement behaviour so that they are always displayed at the same position and in the same size. To do so we added the method *setFixed* to each morph. A fixed morph does not change its position or scale when the world is scrolled or zoomed.

```
initializeBrowserScrollForTouchEvents: function(touch) {
    this.emulatedScrolling = true;
    this.scrollStart = pt(document.body.scrollLeft,
                                          document.body.scrollTop);
    this.scrollTouchStart = pt(touch.clientX, touch.clientY);
}

emulateBrowserScrollForTouchEvents: function(touch) {
    var touchDelta = pt(touch.clientX, touch.clientY).
                    subPt(this.scrollTouchStart);
    var scrollTarget = this.scrollStart.subPt(touchDelta);
    window.scrollTo(scrollTarget.x, scrollTarget.y);
}

onWindowScroll: function(evt) {
    $world.scrollOffset = pt(window.pageXOffset,
                                            window.pageYOffset);
}
```

Code Example 13: calculating and setting the scroll position of the viewport

```
onGestureStart:function (evt) {
    this.zoomingInProgress = true;
}
onGestureEnd: function(evt) {
    $world.zoomLevel = document.documentElement.clientWidth /
                       window.innerWidth;
    $world.zoomingInProgress = false;
}
```

Code Example 14: Gesture event handlers which set the zoom level of the world. Morphs can connect to the zoomLevel and zoomingInProgress field to get notified when changes occur.

This method calculates a normalized scale and position for the morph. These properties depend on the current zoom level of the world and the scroll offset of the browsers viewport. We store the calculated normalized position and scale on two properties on the morph and set connections to all properties we change in our zoom and scroll implementation. This way the morph can react to any change that occurs.
As mentioned earlier, the iPad browser stops the rendering of the page when it is zoomed. One possibility is to update the position and scale after the zooming. This would mean that parts of the fixed morphs which should be invisible[47] become visible when zooming. Alternatively, we could hide the morphs when zooming started and display them with the correct scale and position after the gesture. This way fixed morphs would not be displayed at all when the user zooms or scrolls. We implemented the second alternative[48].
To make sure fixed morphs always are rendered before other morphs, we use *remove* and *addMorph* to hide them. The update functions set the position respectively the scale of the morph.
With this functionality we were able to implement morphs which always have the same screen position and size. We used it to implement tools and widgets like a color chooser, a minimap or flaps.

[47] because they are fixed outside the viewframe
[48] see code example 16

```
setFixed: function(fixed) {
  this.fixedScale = this.getScale() * $world.getZoomLevel();
  this.fixedPosition = this.getPosition().
      subPt(pt(document.body.scrollLeft,
               document.body.scrollTop)).
      scaleBy($world.getZoomLevel());
  connect($world, "zoomLevel", this, "updateZoomScale");
  connect($world, "emulatedScrolling", this, "toggleScrolling");
  connect($world, "zoomingInProgress", this, "toggleScrolling");
  connect($world, "scrollOffset", this, "updateScrollPosition");
}
```

Code Example 15: setting connections to react to scroll and zoom events

```
toggleScrolling: function(isScrolling) {
    if(isScrolling) {
        this.remove();
    } else {
        $world.addMorph(this);
    }
}

updateScrollPosition: function(newPosition) {
    this.setPosition(this.fixedPosition.
        scaleBy(1/$world.zoomLevel).
        addPt(newPosition));
}

updateZoomScale: function(newZoom) {
    this.setScale(this.fixedScale/newZoom);
}
```

Code Example 16: updating morph properties on scrolling and zooming

2.5 Related work

Sencha Touch Framework The Sencha Touch Framework[49] is a popular framework for developing mobile webpages. It is not limited to the iPad but works for most mobile devices including Android and Windows tablets. To support the different touch APIs, it provides a standardized interface offering tap, double tap, long press[50], swipe, pinch and rotate gestures. Like our solution it wraps the DOM events.

---

[49] `http://docs.sencha.com/touch/2-0/` (visited 29.06.2012)

[50] equivalent to our hold

Applications done with the Sencha framework are supposed to look like native applications. The framework offers widgets for touch interaction like menus, lists or icons. This can be compared to Lively Kernel's PartsBin, which also offers user build widgets for everyone to use.

PhoneGap PhoneGap[51] is a framework for developing native apps with HTML5, Javascript and CSS. Unlike other HTML5 frameworks it embeds the Javascript source code in a custom virtual machine. This way the application can take advantage of features which are not accessible via plain Javascript, like vibration, notifications or contacts. Using Javascript as programming language it avoids device specific APIs. The same source code can be used for various platforms.

The drawback of this method is, that users can not simply go to a webpage and get started. They have to install the app first. Sharing of creations is also limited, because other users also have to install the app.

Morphic.js Morphic.js[52] is an alternative implementation of the Morphic user interface. It supports touch interactions, which act similar to our touch to mouse mapping. Touch and hold opens the right click menu on a morph and all mouse click actions are triggered on tap. It does not support scrolling the page. Instead users can drag and drop morphs by simply dragging them on the screen. There is no dedicated selection mode.

2.6 Conclusion and Future Work

The way users interact with mobile devices is different from the way they interact with desktop PCs. Instead of a mouse and a keyboard, mobile devices only have one single touch enabled display. This offers the possibility to recognize and trace multiple fingers simultanously. Web pages are required to adapt to this new situation.

In this bachelor's thesis we have shown how we added support for touch interaction in the prototyping framework Lively Kernel. We examined the internal structure of the touch events Apple specified and had a look at the event registration process of Lively Kernel. We showed how event handlers can be added in Lively Kernel to react to standard and non standard events. We introduced three non standard events tap, double tap and hold, which developers can use to make their Lively application touch compatible. More gestures like swipe, pinch or rotate can be added in the future.

To enhance the usability on mobile devices we implemented features like pie menus or zoom independend morphs based on the work we did on touch events. We also implemented widgets like scrollable lists, hold indicators or flaps. These widgets were successfully tested in applications like an iPad PartsBin with a scrollable category list, an object editor which always edits the currently selected morph and is located in a flap in order to be easily made available. More widgets like formulars can be added in the future.

---

[51] `http://phonegap.com/` (visited 29.06.2012)

[52] `http://chirp.scratchr.org/dl/experimental/JsMorphic/morphic.html` (visited 29.06.2012)

Our work was focused on the Apple iPad 2 with the default Safari Web browser. Future work has to test the implementations on other mobile devices like tablet computers and phones. Different browsers might provide other events. The Firefox mobile browser in the version 6.0 for example has touchenter and touchleave events. Research has to find out how to utilize these possibilities. The touch event API itself is a subject of change as the W3C has not released an official standard for touch events yet.

There are still a lot of possibilities to explore regarding user interaction especially on small devices like phones. Our work has laid the foundation for further support of touch interaction on mobile devices in Lively Kernel.

3 Diffing and Merging of Lively Kernel Parts

3.1 Introduction

The Lively Kernel is a self-supporting, web-based prototyping environment that allows designing active web content.
The content management concept of Lively Kernel is very similar to a wiki: each user is capable of developing websites individually. Apart from designing static content, it allows designing object interaction based on a morphic environment.
The morphic concept offers direct object manipulation and interaction with visual objects that form a scene graph (morph).
The Lively Kernel is built in JavaScript, which causes morphs to encapsulate behavior and state in objects, represented by functions and properties.

Unlike most platforms that allow interactive prototyping, the Lively Kernel runs in a web browser. Hence it can be transported from desktop to mobile devices without switching between platforms.

Sharing concepts in Lively Kernel Because users of the Lively Kernel create web content, they can very easily share their results. With worlds and parts, the Lively Kernel provides two major sharing features.
Each page a user designs is accessible as a *world* and can be shared via a unique URL. The visual content of a world is created by *morphs*.
Morphs that form proper applications or basic structural constructs are shared as *parts* that are stored in a central repository, the *PartsBin*. The PartsBin can probably be best described as an app store that provides parts instead applications. Parts are accessible through copying them from the PartsBin.

Version control in the PartsBin As a central sharing mechanism of the Lively Kernel, the PartsBin provides access to collaborative work and as well demands it. By giving every user access to every part, improving a part is not restricted to its originator. The PartsBin concept rather stimulates users to refine parts individually, due to the self-sustainability of the Lively Kernel.
The immense power the PartsBin obtains in the Lively Kernel is based on the possibility of re-using parts, combining them to complex objects and creating whole applications. [44]
Already during the implementation of part sharing, the PartsBin was characterized with such attributes similar to github [53]. But also the need for proper version control, like github provides it with git [34], can be observed, e.g. when "users want changes to take effect in many places" [44] or two parallel development branches are supposed to be re-joined.

Version control systems (VCSs) are most frequently used in software development. When writing code, a VCS allows saving the state of program code and restoring it when required. In addition to those two abilities VCSs provide mechanisms for version difference detection (*diffing*), *updating* old versions, splitting development paths (*branching*) and joining such branches (*merging*).
By Design, the PartsBin already provides version control mechanisms. It is capable of storing part versions, loading old versions, displaying revision comments and, by saving a part with a different name, branching. Yet some features are missing: There is no possibility to view differences between part versions, or to propagate major improvements to the community by allowing them to update a local copy of a part. Also, multiple branches of parts already exist that cannot be reunited automatically due to missing merge mechanisms.

Aim of this thesis Our implementation of PartsBin version control mechanisms is supposed to add those features. First, it should allow the user to find out which changes were done between part versions. Second, it should allow updating a local copy by applying changes made on graph structures, visual representation, scripts and properties. Third, it should provide the possibility to update local copies of parts that are outdated. Ideally it should also inform users that their copies are outdated, whenever they are trying to accidentally overwrite a newer part version.. Finally, branches of parts are supposed to be merged.

Yet another version control system? Technically, a classical text based version control system that is used to manage a codebase could handle those tasks. Version control in the PartsBin is based on Apache Subversion (SVN) [34]. Each part version is stored as a serialized JSON object, a text based JavaScript Object Notation.

There are two major reasons to not rely on version control mechanisms that come with SVN in the Lively Kernel.
First, understanding changes made to morph properties based on textual diffs is impossible due to the complexity of a morphs structure, resp. its serialized JSON representation.
Second, the LivelyKernel community is not exclusively formed by software developers, whereas the use of revision control is still mainly restricted to software development. We wanted to keep the mechanisms as simple and intuitive as possible: When a user publishes a part that is outdated, he should be recommended to update the part. The update then should be pulled, merged and be published. We do not want to force the user to use a command line tool or a domain-specific equivalent.

3.2 Identifying the problem areas: An example setting

This chapter describes a challenge that evolved in the Lively Kernel between February and June 2012. No diff or merge mechanisms were included in the PartsBin at this time. Figure 27 shows the original version of the ObjectEditor, a part that is used as a tool for manipulating morphs in the world. Four groups were improving the ObjectEditor. It is used for writing scripts for morphs that define their special behavior, but it also allows extending and establishing connections between morphs and evaluating code (*doIt*).

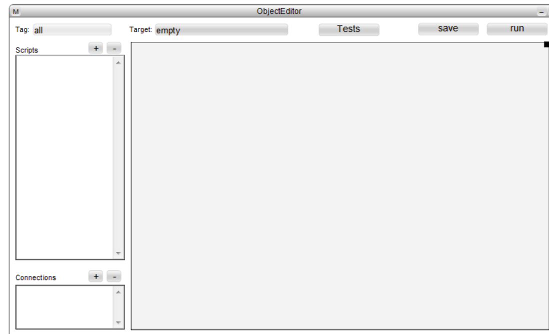


Figure 22: Fabian/originalVersion

During the month, six different ObjectEdiors in four different categories of the PartsBin were created, all deriving from the same stable original version. Those branches had very different aims:

- Fabian was the first to implement additional features. According to his commit messages, his main aspect was to implement tagging of scripts and thus sorting them by those tags. This resulted in the Part *Fabian/ObjectEditorII* (Figure 23).
  In a second branch, he refactored several scripts that he stored in *Fabian/ObjectEditor*.

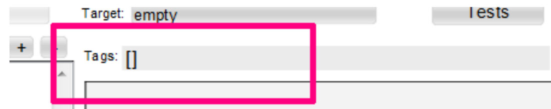


Figure 23: Fabian/ObjectEditorII

- Willy was the second to improve the ObjectEditor. He added a button that when clicked opens a ClassBrowser. The resulting branch is *PartsClasses/ObjectEditor*. Their changes can be seen in Figure 24.

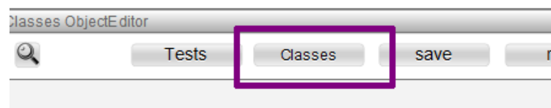


Figure 24: PartsClasses/ObjectEditor

- Third, Lauritz, Tim and Philipp collaboratively worked on *Tools/ObjectGroupEditor*. According to their commit messages, they implemented "saving and retrieving of groups, using tags on the morphs" (2012-06-14, 13:09, timfelgentreff, revision 171073). They, too, refactored bits of the scripts. Figure 25 shows their changes.

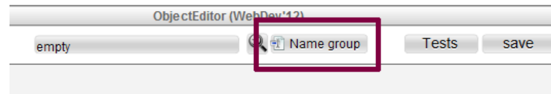


Figure 25: PartsClasses/ObjectGroupEditor

– The last version was created by Jan who implemented a history view for scripts (see Figure 26), so a user could revert changes at any time. He created the *groupBejeweled/LimeMachineObjectEditor*.

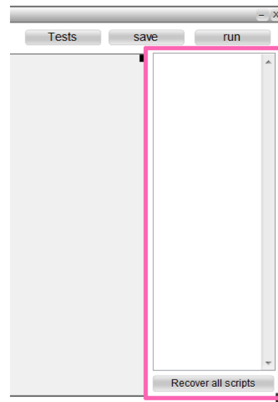


Figure 26: PartsClasses/LimeMachineObjectEditor

– Finally, the main branch *Tools/ObjectEditor* was improved and refactored, too. Figure 27 shows, that e.g. a search button was added.

Figure 27: PartsClasses/ObjectEditor

As the PartsBin is based on an underlying SVN repository, all those revisions were given increasing revision numbers. Figure 28 naively guesses the possible history flow of the example branches. This guessing is based on the naive assumtion that the revision numbers imply not only the order of part creation, but also the instanciation order of morphs that were later saved as parts. Therefore, if there are three part versions

– version *A1* with revision number 1 ([A1,1]),
– version *A1* with revision number 2 ([A1,2]),
– version *A2* with revision number 3 ([A2,3]),

and we know that A2 was created out of A1, we assume the following order of actions:

1. a morph was created
2. the morph was published as [A1,1]
3. a local copy 1 of [A1,1] was created and modified
4. the local copy 1 was published as [A1,2]
5. a local copy 2 of [A1,2] was created and modified
6. the local copy 2 was published with a different name (branching) as [A2,3]

With that, we assume that [A2,3] derived from [A1,2]. The order could also have been 1-2-5-3-4-6, meaning that [A2,3] could also be derived from [A1,1].

Our task was to merge those many different branches back into the main part *Tools/ObjectEditor*. We had to find out the correct branching history, find the best merge plan and, one after another, merge the branches.
The resulting part should then include tag visualization, grouping, script history access and opening the ClassBrowser. Also, the user interface providing access to these features should include all required access points at the end.
Figure 29 gives an example of what the merge could look like.

Without an automatic merge strategy, this task may be very time-consuming. Apart from the optical differences like resized, repositioned and added submorphs, many changes were made in scripts and non-visual properties. Going through all those possible changes manually, performing a manual merge process in the end, turned out to be very annoying due to several reasons: It requires a correct remembering of the changes an author made and by that the real-time availability of the author for a version to merge. And yet, a merge could probably be incomplete.
Merging two part version automatically requires

1. to avoid accidental overwriting of parts, warning mechanisms for outdated morphs
2. computing the correct history of a version to find its origin version
3. computing differences between part revisions
4. extracting updates that are not fast-forward (conflicts)
5. updating the part revision

We propose a merge workflow that assists the user during step 1 and 2 without doing them automatically. With then three given versions, a common parent and two differing, we want to be able to perform an automatic update, with respect to possible conflicts that can occur if two users e.g. refine the same script.

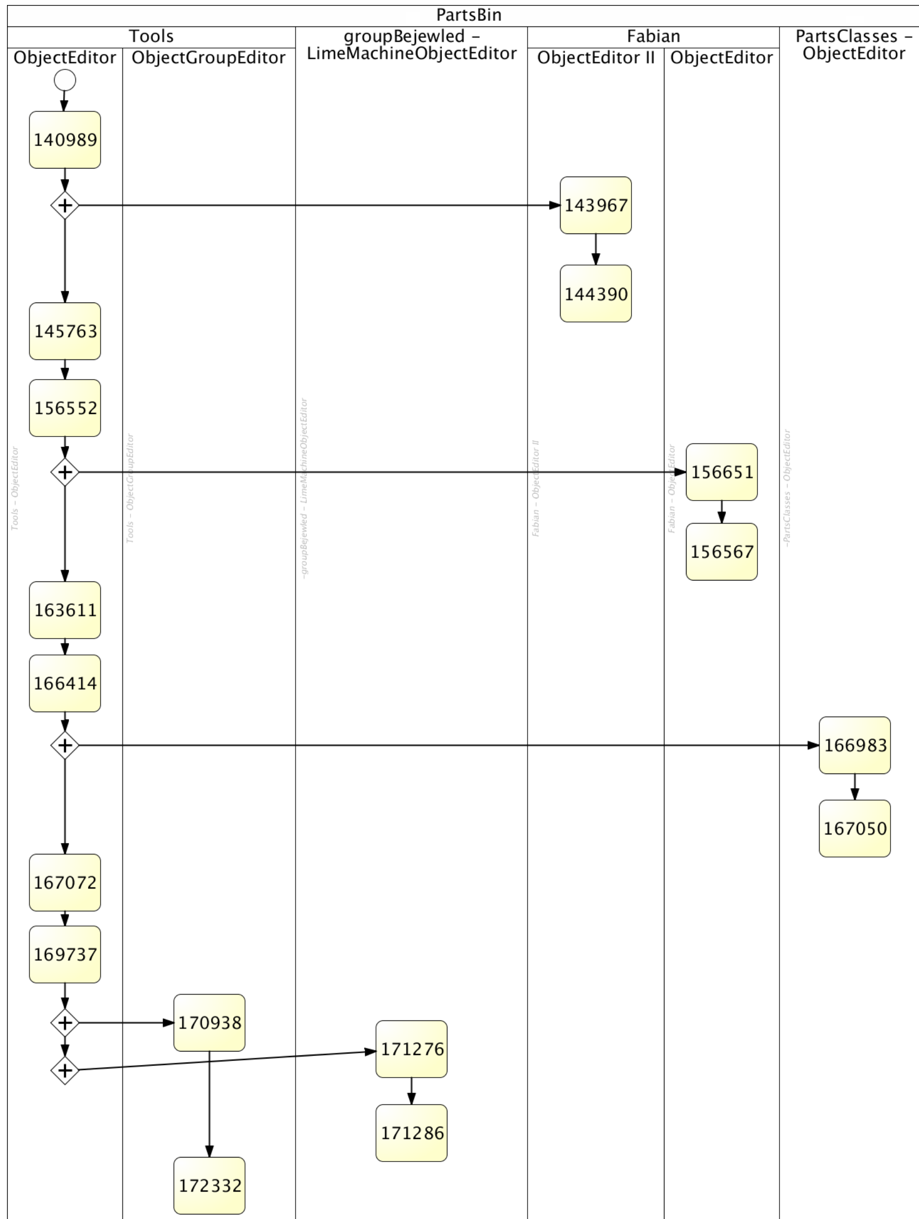


Figure 28: The naive assumption of the merge history, only based on revision numbers and part names.

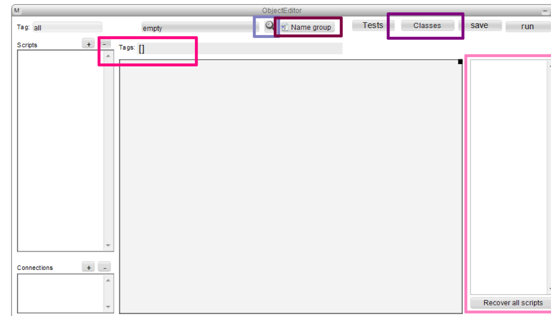


Figure 29: The most probable merge result - The changes of differend part branches are highlighted.

For the last step, we want to implement features that support the awareness of collaborators. In many cases, a user of LivelyKernel copies a part to its world, refines it and then publishes the new version. If another user, too, refined the part in the meantime, we want to warn the overwriting user, present possibilities to merge and solve conflicts and then publish a merged revision – so awareness of a collaborating user should be created automatically by the system. Informing users that copied the old part version to their worlds about the updates in the part could also be an aspect worth being implemented. Such continuous integration features are supposed to avoid creating huge branches like in the example in Figure 29.

We distinguish between continuous integration and branch merging. For continuous integration, steps 1–5 are supposed to be done automatically, for branch merging we only want to support the automatic execution of steps 2 and 3.

3.3 Approaches and decisions

Version control systems that support all the mechanisms shown in the example can be based on several very different concepts.

This chapter is structured to first give the academic motivation for the need for merge strategies and the reason for implementing a state-based comparison approach. It introduces the copy event as trigger for a new version creation and a system to associate those versions. With this background it describes more specific approaches to version diffing and merging, in particular analyzing the two-way and three-way merge strategies and proposes a morph representation that we used for comparison.
Those steps will be explained with the use of the example setting introduced in the previous chapter.

An optimistic source-management model Before thinking about branching and merging, it is necessary to know about the motivation for those techniques.

Beginning at the very base of a version control system, its collaboration model can be either pessimistic or optimistic. In pessimistic approaches collaboration conflicts are actively avoided by applying blocking behavior to make sure that only one collaborator can access a resource at one time.

By design this avoids scenarios that require merging two or more versions, but with the huge disadvantage that collaboration can only happen sequentially instead of parallel. The Lively Kernel PartsBin as well as world sharing mechanisms follows the optimistic approach. It explicitly distinguishes between parts as commonly available objects and morphs as local copies, ensuring that changes made to a morph do not propagate to the whole system. But, as stated in [44], this must at one point result in different part versions and even whole development branches that need to be merged. Our ObjectEditor challenge is a branching example as many versions existed that were further improved. But modifications on a morph that are then published to the same part can bring up merging issues, whenever there are two collaborators improving the same part at the same time.

On the contrary, the Lively Kernel code base follows this global concept, ensuring the existence of only one single development branch of the core system.

Comparison models With the challenge to merge branches that may have existed for a long time, we need to be able to compare two existing states of a part.
A state-based diffing model provides a general approach to object comparison. Computing a state-based diff, especially for complex objects like morphs, tends to be rather time-consuming, a fact that we will evaluate in chapter 3.5.
A second diffing approach is to record each manipulation done to a morph in a single iteration step. To update a revision, those iteration steps are put in the right order to one big change set and are finally applied to the base version for both.
In the Lively Kernel, there is a huge number of possibilities to manipulate morphs which were evaluated by Dannert during his implementation of WebCards in LivelyKernel ([16]). Dannert extracts the possibilities of exclusively using tools for morph manipulation or utilizing convenient getter functions to hook into convenient events. He aims to create command objects to propagate changes during a template modification. He decides to utilize getter methods, finally being able to produce commands that propagate visual changes on morphs.

Our merging approach requires possibilities to trace all kinds of property changes, which would result in either creating a very large set of convenient access methods or tracing general access events, like mouse and keyboard events. As both possibilities include too many different options in the Lively Kernel, we decided for the state-based diffing approach.

In connection with the state-based approach, we chose to consider a copy event as indicator that a new version of a morph is being created.

Computing branching histories based on unique ids Due to this optimistic source-management model, we will have to deal with different part versions. There can be several reasons for creating new morph versions by copying them. A *local copy* of a part in a world created a new, non-permanent morph version. Local copies can be meant to

- improve a part by copying, refining and publishing it,
- be stacked to a composition of morphs, forming new parts,
- allow editing other morphs, the basic use case for lively tools and widgets.

Typically, if a local copy is supposed to be published to form a new part version, it only exists a relatively short time (hours or days).
A *branch* is a copy of a part that was published in the PartsBin with another name. One reason for branching is the development or completely new parts, but often, like in our example, two similar branches represent two development directions that are supposed be joined at one point. A branch that is aimed to refine a part is often supposed to exists only over a time of days or weeks.

When a local copy of a part is created, its overall identity is stored as its parts bin meta info directly on the morph. When publihsingin this morph a s part under another name an with that creating a branch, this means changing its parts bin meta info and with that changing a morphs identity. The part that results from the publishing has no relation to the part it originated from anymore. When it comes to merging branches, an approach must be found to identify their relation. Simply assuming their common history based on the order of their revision numbers they get from the underlying SVN is a way we tried in our example (see Figure 28). But without a part name and good guesses, the commit order gives no hints at all as it is increased with every commit to any part.
The other very relevant issue is about finding related morphs within two morph versions. When scene graphs are supposed to be compared, it is an essential task to find out correlating submorphs in both versions (see chapter 3.3).

There are several approaches to identify branch histories. SVN [34] tags branches with meta data and git references a head revision for each copy. Due to the fact that parts consist of morphs, part branches differ from branches in text-based approaches, because submorphs need to remember their own identity. Only with that they can later be updated as submorphs, too.

As parts and morphs are strongly connected concepts, branch history extraction and submorph identification can be considered to be overlapping steps. We decided to implement just one general strategy that suits both problems, by giving morphs (and thereby parts) unique identifiers.
In the Lively Kernel, every morph is given an identifier (ID) when it is copied (from a part or a morph). An ID is a four digit number that must neither be unique in one world nor in the PartsBin. As we need unique identifiers, we reused an existing implementation in the Lively Kernel by Czuchra [15] in our approach and integrated the unique identifiers in the live system.
To restore a parts history later on, we have to be able to keep track of the copy actions a morph has gone through. We decided to not only keep track of publishing actions (which require a copy, too) that create nuew part versions, but also local copying of morphs, to preserve the possibility of local diffing and merging, too. To remember each copy action, we simply stored the unique identifyers a morph was given through time, thus keeping track of part version creations through the action of copying a morph to the PartsBin. We implemented these mechanisms at a very early point of our project with the benefit that the branches of the ObjectEditor were referencing complete derivationIds arrays.

This very general access to a morphs history allows us to compute branch histories as well as finding related morphs in two different scene graphs by comparing sets of derivation ids.
Based on this knowledge, we are able to correct our original assumption of the ObjectEditor branch history and manually restore the actual one, which is shown in Figure 30.

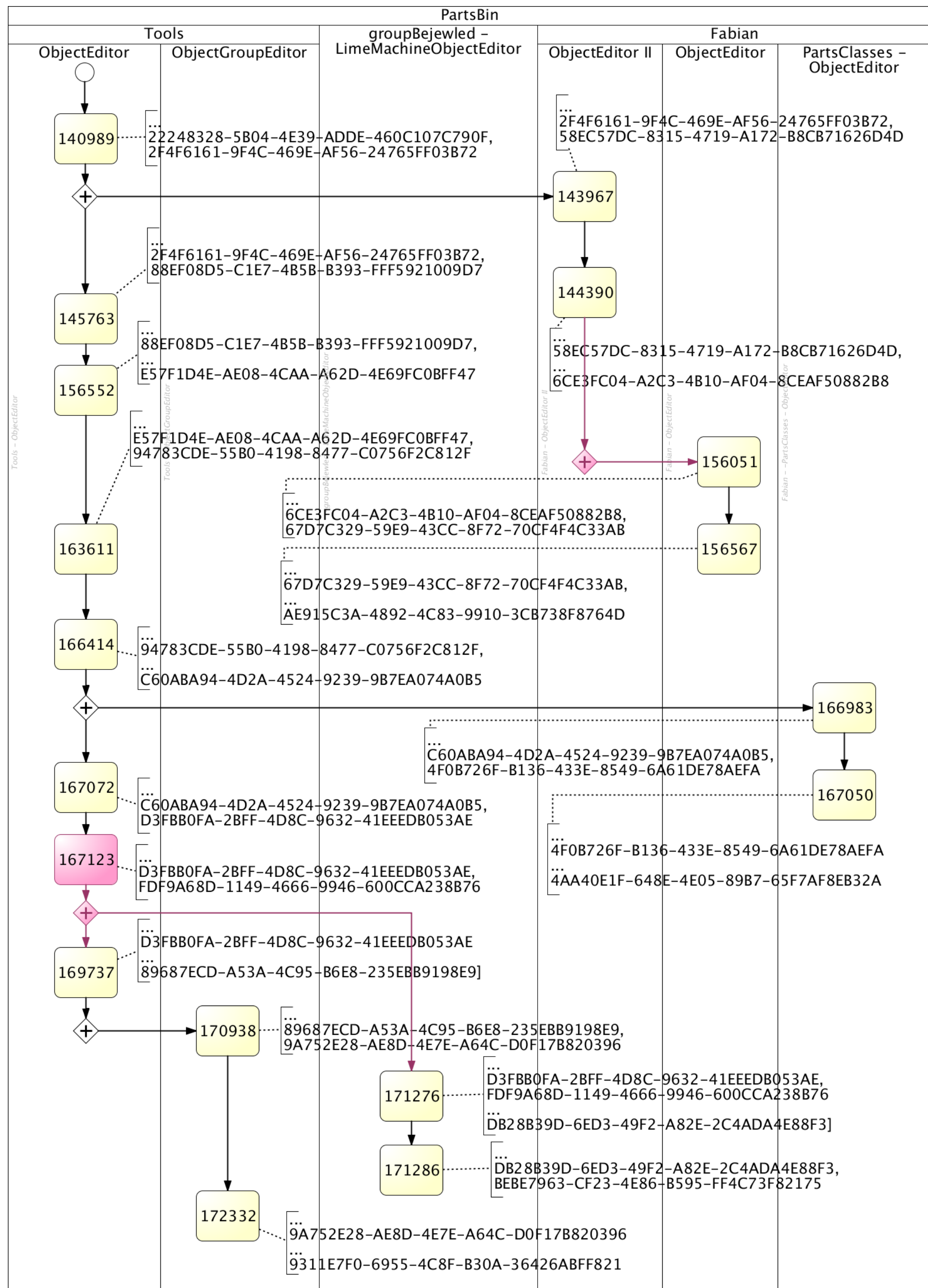


Figure 30: The correct branching history, restored based on derivation ids. The most interesting derivationids are displayed as annotations to the revisions.

The second benefit of derivationIds, morph relation computation, will be explained in the next subsection.

Relationships and derivation Based on derivation ids it is now possible to find out the relation between two morphs by analyzing their common history. Common derivation ids indicate a relation. The less derivation ids two morphs have in common, the less they are related. There are three types of relations that can be identified here: unrelated, parent-child relation and sibling relation. They are visualized in Figure 31.
We can determine that

- non-related morphs have no common derivation ids at all,
- the current id of a parent morph can be found within the derivation ids of its children (a child was copied from its parent),
- siblings have a common history (i.e. non-empty intersections of derivationIds), but the id of one sibling is never included in the derivationIds of the other one.

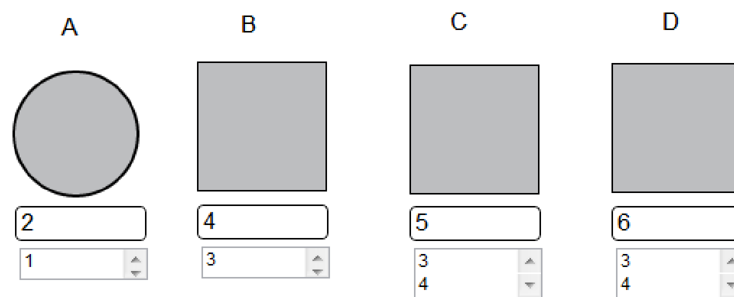


Figure 31: Relation types in Lively Kernel: The single numbers are current ids, the number lists are derivation ids. Morph A is related to neither B, C or D. Morph B is a parent morph of C and D. Hence, C and D are siblings.

Based on this idea, two Buttons in a scene graph are always related due to their identity as buttons. This becomes quite a problem when scene graphs are compared, as this makes it hard to determine if two Buttons are siblings due to their classification (both are Buttons) or if they are related even closer (one Button was copied from the other when the part was copied).

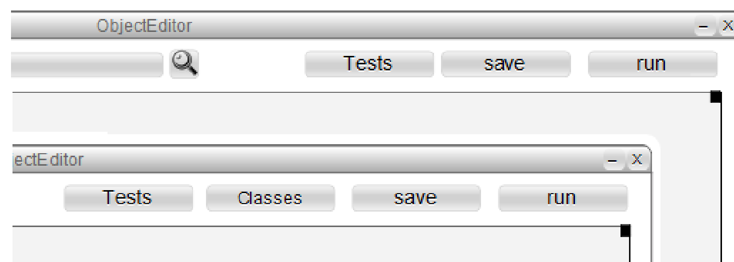


Figure 32: Finding pairs of related Buttons in ObjectEditor revisions.

Figure 32 explains this problem by example. The *PartsClasses/ObjectEditor* and its parent revision from the *Tools/ObjectEditor* branch are being compared. In the upper area, we want to mark the relations of the Buttons. We can see that the *Classes* button must have been added and want to find that out based on comparing derivationids.

Table 1 displays a simplified overview of the derivation ids for this setting, comparing a Button that was just copied from the PartsBin with the *save*, the *Classes* and *Test* buttons from the ObjectEditor versions. For readability reasons, we shortened the ids and exchanged equal lists of ids with the capital letters A to E. The four digit numbers in row 1 are leftovers from the time when identifiers in the Lively Kernel weren't unique.

		Tools/ObjectEditor		PartsClasses/ObjectEditor		
	Button	Tests	save	Tests	Classes	save
1	7251	7251	7251	7251	7251	7251
2	5EE2BB6065A6	279956A34FC2	413BD2FFA0D1	279956A34FC2	279956A34FC2	413BD2FFA0D1
3		A	B	A	A	B
4		8B5630CF2EDE	090B9817E099	8B5630CF2EDE	8B5630CF2EDE	090B9817E099
5			C	D	D	C
6			2537402A6382	77DA5B862FA6	83FFD3D2AE15	2537402A6382
7					85AD9789CED9	93FD076B9089
8						E
9						475FD541703B

Table 1: Simplified listing of the derivation ids for Figure 32

Analyzing the steps we can state the following:

1. From row 1, we can see that all are descendants from the original Button part, assuming that the four digit id (a leftover from the old implementation od ids) is unique within the PartsBin.
2. *Tools/ObjectEditor - Tests* is the parent of *PartsClasses/ObjectEditor - Tests*, as its derivation ids are completely contained in those of his child.
3. For the same reason, *Tools/ObjectEditor - save* is the parent of *PartsClasses/ObjectEditor - save*, and *Tools/ObjectEditor - Tests* is the parent of *PartsClasses/ObjectEditor - Classes*.
4. From row 2, we can see that in the *Tools/ObjectEditor* version the *Tests* and the *save* Button are siblings, so both were originally copied from the PartsBin.
5. Also from row 2, it becomes clear in *PartsClasses/ObjectEditor* the *Classes* button is stronger related to the *Tests* Button than to the *save* Button, as their common history is longer.
6. *Tools/ObjectEditor - Tests* and *PartsClasses/ObjectEditor - Classes* are siblings, but not by their identity as Buttons, as their common history is longer than their common history with a Button.

What we actually want to find out is that *PartsClasses/ObjectEditor - Classes* has no parent match in *Tools/ObjectEditor*, but we found *Tools/ObjectEditor - Tests* in point 3 due to transitive parent relations.
We therefore need to find out if a submorph was individually copied or copied with the whole scene graph. Our solution is to compute the amount of versions that lay between the two parts or morph versions, whose scene graphs are compared. If a scene graph was copied n times, a child's derivationIds array cannot be more that n times longer that its parent's.

Based on these approaches, we are now able to manually recreate branching histories of parts and matching submorph relations between two morph versions.

Granularities Until this point we assumed that there is a need to find matching submorphs withing two morphs under comparison. This chapter gives the reason for this assumption.

Transferring the morphic version control approach to classical text-based diffing, morphs and submorphs can be handled like folders, properties like files and property values like file content. As functions in object oriented approaches like JavaScript are object properties, too, they can be handled like properties.
This model is a very generic one with quite a balanced granularity.
However, when moving away from this folder-file-content metaphor, the method of comparing scene graphs as single units comes into view. This approach would include handling a morphs submorphs as a property without distinguishing them.
Due to the intuitive approach of handling morphs as such we decided for the fine-granular approach. Apart from being as intuitive and clear as possible, we wanted to provide possibilities to notice if a morph was removed in one version or simply changed its owner within the scene graph. When morphs are handled as properties, this is impossible.
Sow those submorphs are compared will be explained in the next chapter

Object structure In the Lively Kernel there are at least four representation types for morphs.
First, there is the visual representation. For human eyes, it is very easy to see visual modifications here, but impossible to find non-visual changes. Therefore, the visual representation is not suitable for comparing objects algorithmically.
Second, there is the basic JavaScript Object. It consists of native types that are *string*, *number*, *object*, *function* and *unefined* and references to such properties. JavaScript Objects can contain recursive structures withing their properties.
Third, there is a linearized representation of JavaScript objects. During a linearization process, recursive structures are removed from the JavaSctipt object and complex objects of no native type are split.
The fourth representation is a serialized JavaScript Object Notation (seriailzed JSON). This is a text-based representation generated right from the linearized object.

Differences in two morph versions can be detected in each of these representations, as it can be seen in Figure 33, but not all representations are suitable for a computational morph comparison.

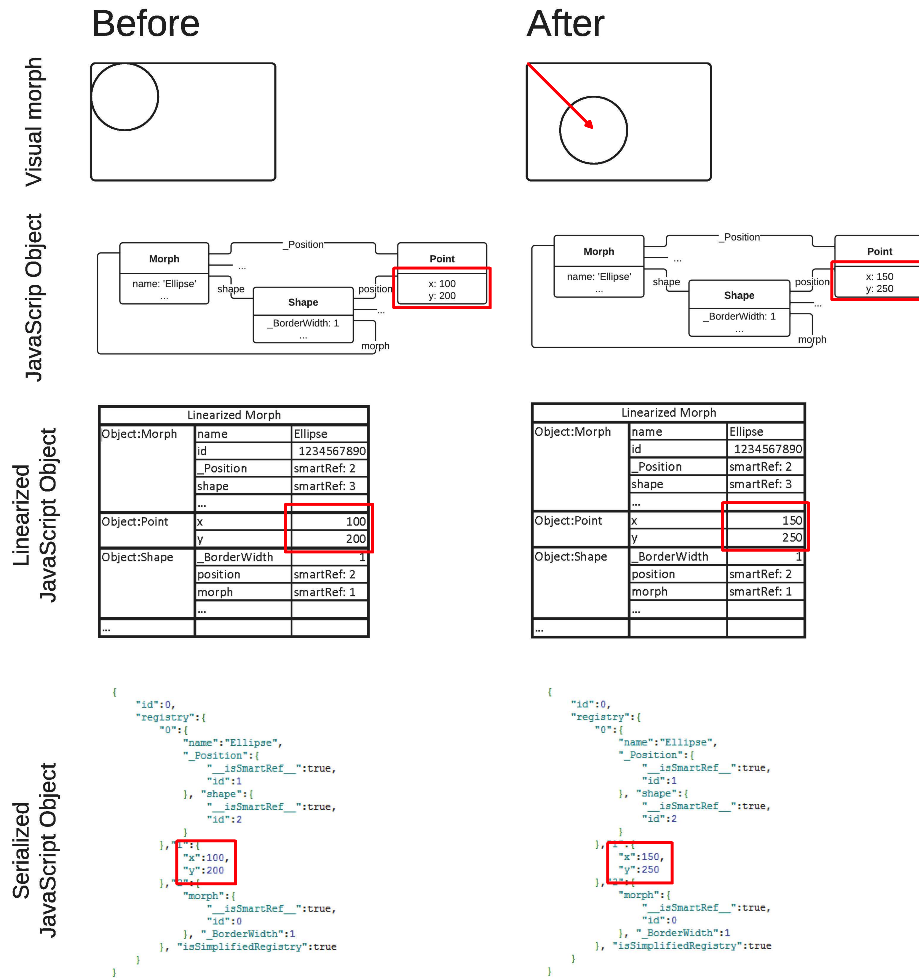


Figure 33: Two morph versions in different representations: (top-down) visual, JavaScript Object, Linearized JavaScript Object, Serialized JavaScript Object Notation. In the visual representation, we see that the circle changed its position which is marked in the on-visual representations.

It is impossible to directly compare JavaScript objects that contain recursive structures, as we cannot ensure that such an algorithm terminates. The other extreme, a serialized representation, is very suitable for a diffing approach, as it is not only in a text formate but also without any recursive structures. Cosidering the usage of a third-party application to compare the textual object representation, the result could be a basis to recalculate the changes done to the actual objects.
The diff that a text-based diffing algorithm produces basically contains a line number and the textual change. We consider it as ineffective to recalculate the correct modification to a property out of these data and decided to implement our diffing algorithms based on a linearized object structure, that represents an interstage where changes are computable and still meaningful enough to recalculate the actual modification made to the JavaScript object.

Diffing bases Two linearized JavaScript objects can basically be considered as equal if they consist of the same subset of properties and all those properties are equal. There are several reasons why this definition is not suitable in our implementation of version control:

- The properties *id* and *derivationIds* cannot be equal in two versions of a morph due to their design presumptions.
- There are properties that are simply way too complex to be compared recursively, like the data object model (DOM) representation of a morph.
- We have rapidly changing properties that are not relevant for the object itself, like references to a halo or pie menu, or connections that are established when a morph is transformed
- There are use cases when certain properties should not be compared. In our example application, the *Add Script* button keeps connections to the list that displays the scripts. Those connections should be excluded from the comparison as they keep hard links to target objects that exist in one version but are replaced by a copy in another.

There are two approaches to ensure that a certain subset of properties is supposed to be diffed: The first is to have a predefined subset (whitelist) of properties that will be considered for the comparison, and the second is to define properties that will be excluded from diffing (blacklist). A whitelist algorithm is very static, as the list of comparable properties has to be updated each time a property is added to the object. For a flexible framework like Lively Kernel, the blacklist algorithm is much more appropriate, although much more complex.

To compare two linearized objects, we traverse the linerarization array in both versions, compare them step by step and store differences in a Diff object. The next chapter will explain where in the update workflow those two-way diffs are being considered.

The three-way comparison There are two ways of comparing objects, a two-way diff and a three-way diff.
The intuitive way is probably the two-way diffing strategy, that takes two objects and compares their properties. Without any statement about possible relations between two morphs, with a two-way diff we can find out that a submorph is present in one version and not in the other version, that colors differ or that the set of scripts that defines morph behavior differs.
We can, however, tell which version derived from the other. The statement "This button exists in only one version" can with that be clarified to "This button was added in version X" or "The color was changed from yellow to blue". So we can very well use a two-way diff to compare parent morphs with children.

When merging branches, we are comparing sibling morphs where a two-way diff looses this precision. Again, we could simply state that e.g. "One sibling is blue, the other yellow.", even if only the author of one version changed the color from yellow to blue and the other did not modify this property.
By determining a parent version both siblings derive from and computing the modifications each author made to its own version, it is possible to make more precise statements. Such a comparison between three morph versions is called a three-way diff.

sibling version	focused version			
	added	removed	modified	untouched
added	∅ (morph) diff (property)	∅	∅	∅
removed	∅	OK	conflict	update
modified	∅	conflict	diff	update
untouched	∅	OK	OK	OK

Table 2: A three-way merge matrix

According to two-way diffing a morph to its parent, we isolated a set of different change types that can be done to a morph. Considering a focused morph and a version it derives from, this means it or its properties can be

- modified
- not modified
- added
- removed

Three-way diffing approaches take three morph versions: A source or parent version, and two versions that represent derives of the sources. Lets call those the focused version and its sibling.

A three way diffing algorithm computes changes made in both versions, the focused and the sibling, and compares them. Whenever a morph or one of its properties was modified in both versions, this is a possible conflict. The merge matrix in Table 2 is based on the merge matrix proposed in [31] and is a basis for a three-way merge algorithm to decide whether a property or morph modification is conflicted or not.

That merge matrix can be read as following example shows: If I modified a morph or property that was removed in the sibling version (usually the current PartsBin version), this is a conflict.

The matrix applies to modifications on morphs as well as modifications on properties. We assume that the user wants to update the focus version with changes made to its sibling respecting their common parent version. *OK* states that no update is required, *update* states a non-conflicting modification to a property, *conflict* states a conflict that has to be solved and *diff* means that both versions modified a morph or property and that it may be conflicting. By comparing both modifications, it is possible to find out if they conflict or were equal. If a meeting of both changes is impossible, this is marked with ∅.

Based on this matrix, an applicable patch can be created that updates a morph. The next section will explain ways of solving conflicts that cannot be included in such a patch. After a three-way comparison has been done and a patch was created, this patch can be used to merge the two versions.

Merge policies A merge policy can be automatic or manual. Manual does not mean the fully manual merge of version without programmatic support, but a conflict solving strategy.
Whenever two collaborators are developing a software or object version, their modifications can be conflicting when they are made to the same atomic unit. For text-based software, a conflict could occur if the same file is modified in two concurring versions. For morphs this can be transferred to properties, so whenever both versions manipulate the same property of a morph, this can be conflicting if the results are not equal.

The manual merge policy is known from version control systems like git or SVN [34]: Non-conflicting updates are pulled automatically, whereas conflicting updates are presented to the user, who then decides about the correct merge.
Automatic merge policies support time-critical merges best by finding indicators (e.g. time of the modification or its author) that specify the modification that is more relevant [31].

We implemented the manual approach for update mechanisms in the PartsBin, as currently there is no indicator for which version is the best. Still, as a merge can possibly pull updates that are either dysfunctional per se because a version contains errors or because they conflict too badly with the own implementation, an update must in every case be revertible. We also want to allow to revert decisions made during the conflict solving process, to test different combinations of merge possibilities.
The automatic merge policy in the online collaboration of the SyncMorph (see section 4) uses the time of a change as an indicator for a 'last commit wins' policy. When there are only very small iterations that result in a new version, an automatic merge policy is suitable, as resulting mistakes can instantly be noticed.

With respect to the long-term iterations described in the example setting (see chapter 3.2), we want to reduce the number of manual-merge requiring conflicts as much as possible. A neatly isolated subset of merge relevant properties massively affects the number of conflicts.

3.4 Implementation

When publishing a morph in the PartsBin, the user opens the morphs context menu and uses the *publish* entry. This opens the publish window (Figure 35) that allows setting the part name, choosing a category in the PartsBin, adding a comment for the part and adding a revision comment.

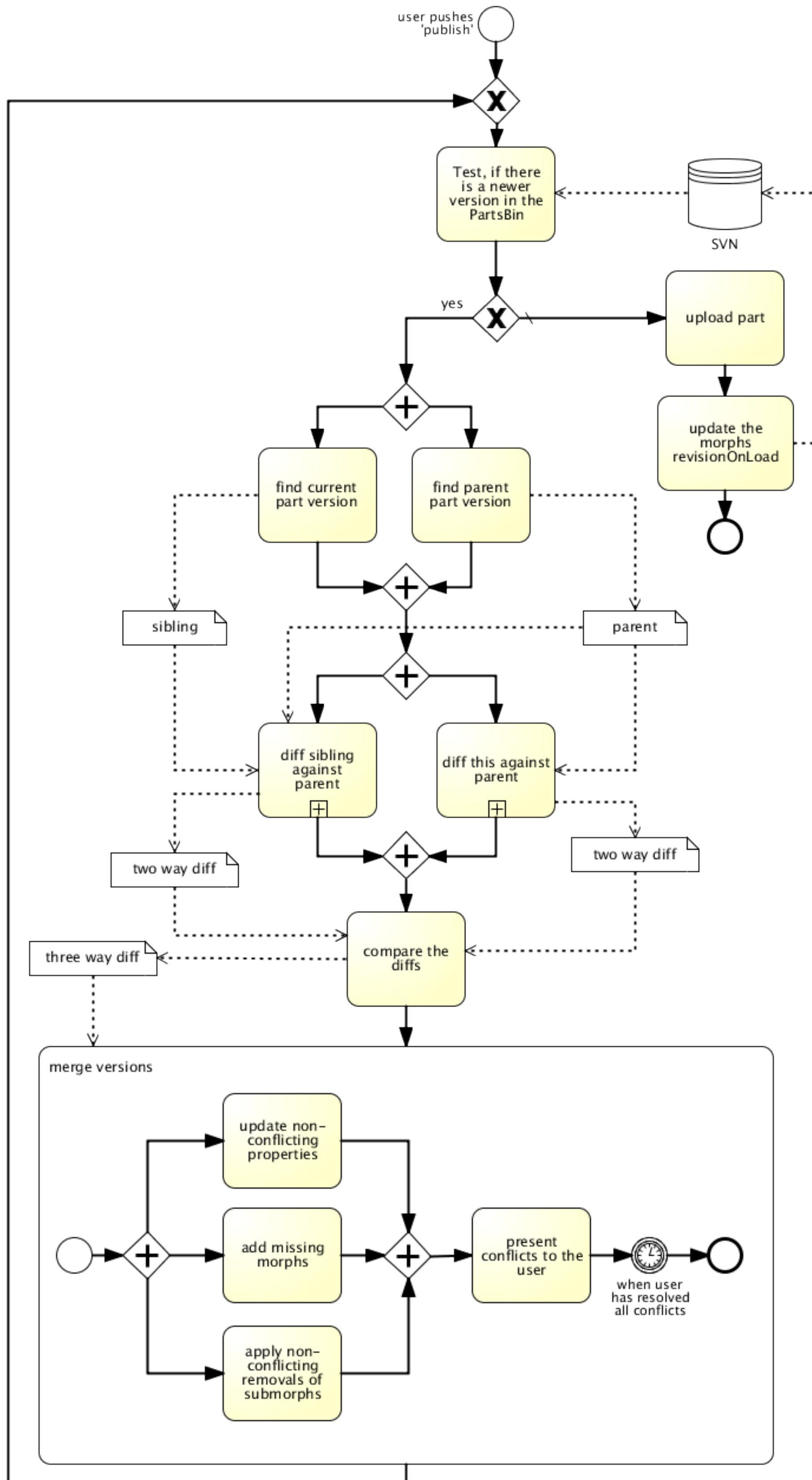


Figure 34: The update workflow.

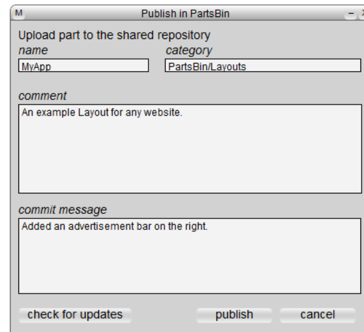


Figure 35: The publish dialog.

There is a publish button that uploads the part, also the process can be canceled. We added a button *check for updates* that, when pressed, requests the web resource for the part that is being published and, if there is a newer version, allows triggering a merge process.
In most cases a user will simply press publish as he will not await that other users updated the same part. To prevent collaborators from accidentally overwriting useful changes, we extended the publish function by the same request that is done when checking for updates, so the user can be warned if a newer version exists and gets a possibility to update his morph. The following paragraphs will explain what happens in those cases.

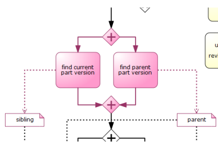


Figure 36: Version finding during update.

Version extraction of morphs and parts During the update process a three way diff is performed which requires three revisions of the morph:

- *focus*: The local copy in the workspace (this)
- *sibling*: the current revision in PartsBin
- *parent*: The revision both sibling and focus derive from, i.e. the revisionOnLoad of the focus.

Versions are stored in a backend SVN and can be accessed via a URL that is computed out of the part name, part space (category in PartsBin) and its revision. The function getParentPartVersion computes the original version by accessing the morphs revisionOnLoad:

```
getParentPartVersion: function () {
  /* returns the original PartsBin version of the morph */
  var partItem = this.getPartItem(),
    revision = this.getPartsBinMetaInfo().revisionOnLoad;
  if (new WebResource(partItem.getFileURL()).exists()) {
    return partItem.loadPart(false, null, revision).part;
  }
}
```

Formerly, the loadPart function used the morph name, which caused problems whenever morph name and part name differed. For the computation of the sibling version, the function getCurrentPartVersion utilizes loadPart without adding a special revision number.

Based on the derivationIds of all three parts, we can find out how often each one was copied from the parent version. As loadPart performs a copy action, all of the three versions were copied at least once:

- The parent was copied exactly once
- The sibling was copied at least twice: loadPart, copyToPartsBin (which does not perform a copy action), and again load loadPart
- The focused version was at least copied once, as well as the parent

```
1 timesCopied: function(parent) {
2   /* returns the number of copy actions
3   regarding preceding parent version */
4     return this.derivationIds.length -
5       this.derivationIds.intersect(parent.derivationIds).length
6 },
```

As stated in chapter 3.3, we use those findings as a hint to find out if submorphs were added in a newer version.

Detecting changes The main part of the update workflow deals with change detection. As we want to compare submorphs based on a three-way diff, we first have to produce a change set for both versions, sibling and focused, for each submorph pairs. Only after that we can compute the actual three-way diff. Figure 39 vizualizes this workflow.

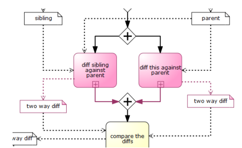


Figure 37: Diffing steps during the update.

Part and morph relations: About UUIDs and derivation During the two-way diff of a morph against an older version of it, it is necessary to find pairs of submorphs to compare them. Those relations are computed based on the derivationIds of submorphs.

We begin with two different morph revisions (we call them focused and relative) and, as shown in Figure 39, start searching pairs of corresponding submorphs. We begin with the first submorph and hand it over to the function findAncestorIn, that compares it with each submorph in the relative. Also, we hand over the number of times the focused (m) and the relative (n) version were copied from their common parent (see page 63 for computation of m and n).

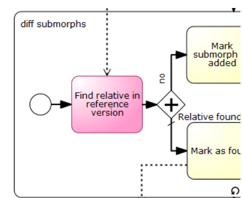


Figure 38: Finding related submorphs during a two-way diff.

prepareDerivationIds removes all those Ids from the derivationIds that were added during the various copy events required to access the versions for our comparison. The last identifier on the stack then must be the id of the youngest part revision both versions directly derive from. If those equal, we found a sibling. If there cannot be any sibling found, the morph was just added.

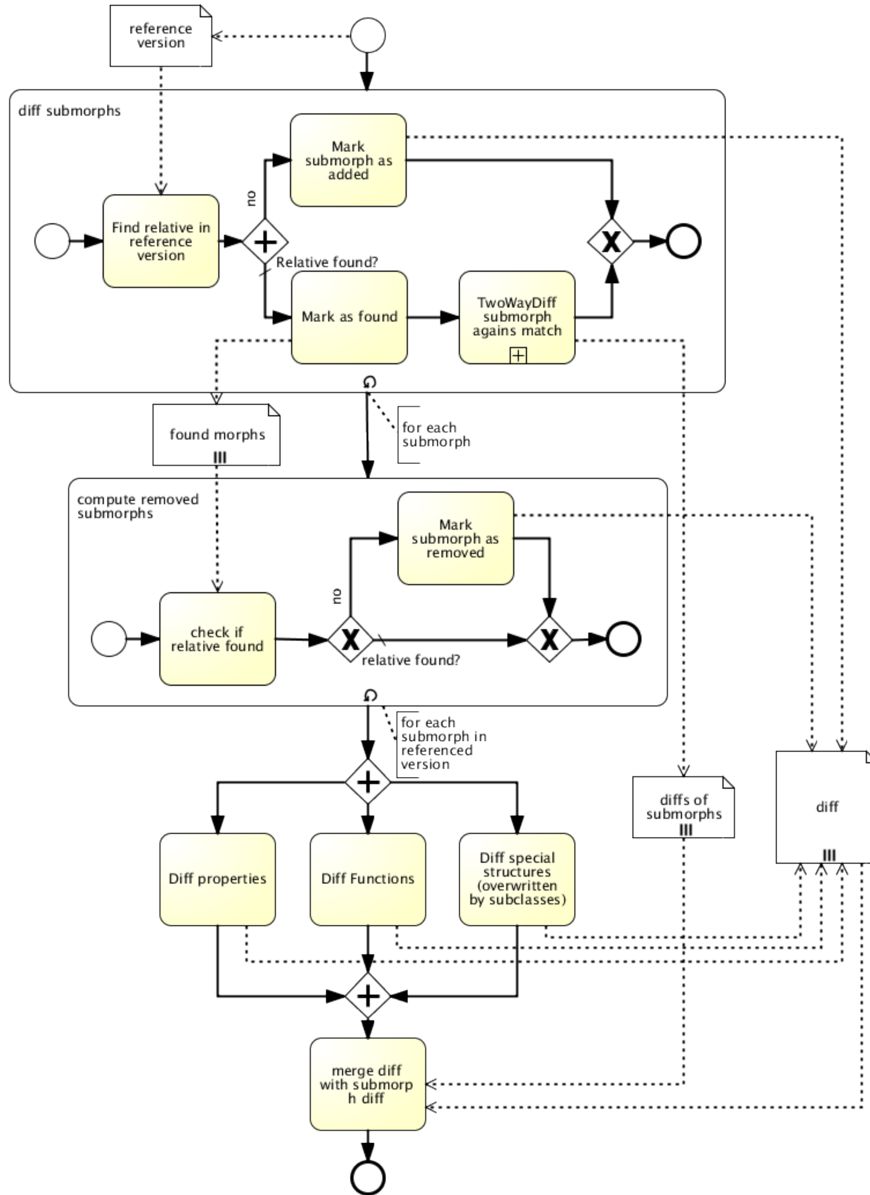


Figure 39: Two-way diff

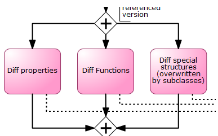


Figure 40: Two-way diff during diffing process.

Two-way diffing The two related submorphs found will then be compared. Functions can simply be accessed via *Functions.own(morph)* and compared after converting them to a string native. Also, it is theoretically possible to compare special structures, so that very complicated morph classes like Text can be compared, too. Those become complicated as they consist of RichText TextChunks, which is hard to read for a user that may have to solve conflicts. Simply comparing two TextStrings is much easier and considered as a sufficiently exact modification indicator.
The most interesting part of the two-way diff are a morphs properties, as those can be recursive or very complex structures. For a property diff, we first linearize the two morphs by sorting all *object* type properties that may keep recursive structures into a list and accessing them by reference numbers, thus breaking recursive structures (see Figure 33 for a comparison of possible object representations). We can then traverse the linearization until we find atomic properties. We define an atomic property to be an object that

```
findAncestorIn: function (scope, m, n) {
  /* this morph has been copied m times from the common
  parent, the scope morph has been copied n times. */
  var myIds = this.prepareDerivationIds(m),
    self = this;
  if ((this.derivationIds.length - this.getIdOnLoad() < m)
    return;
  if (myIds) {
    return scope.withAllSubmorphsDetect(function (ea) {
      var otherIds = ea.prepareDerivationIds(n),
        otherIsNew =
          (ea.derivationIds.length - ea.getIdOnLoad()) < n;
      if (otherIds && (!(iAmNew && otherIsNew)))
        return otherIds.last() === myIds.last();
    })
  }
}
```

- is a JavaScript native type, i.e. *number*, *string*, *undefined* (*function* objects are not found as property),
- is a null object,
- is an empty object, or
- defines an *equals* function that, with the corresponding property from the sibling morph, does not return undefined.

As stated in chapter 3.3, we allow handing over a blacklist of strings representing properties that are not going to be compared.

This dispatch is done by the *areEqual* function. Because, like Czuchra already described during his implementation of WebCards ([15]), *undefined*, *null* and empty objects are hard to compare, this is a very central function in our implementation of a two-way diff. We also added a possibility to compare *NaN*, which is a number type but does not equal NaN by default.

Whenever two properties are comparable, *areEqual* returns either true or false. Based on this result, we store both values in a Diff object that will be reused for the three-way diff.
E.g., if the color of a morph was changed, the Diff object looks like displayed in Figure 41.

```
areEqual: function (a, b) {
  /* compares two objects. Returns
  true, if both are equal
  false, if they are not equal
  undefined, if they are incomparable */
  if (typeof(a) === 'object' || typeof(b) === 'object') {
    if (a == null) {
      if (typeof(b) == 'number' && isNaN(b))
        return true
    }
    else if (Properties.all(a).length === 0
        || Properties.all(b).length === 0) {
      return
        Properties.all(a).length === Properties.all(b).length
    }
    else if (a.equals)
      return a.equals(b)
    else
      return undefined
  }
  else if (typeof(a) === 'number') {
    if (isNaN(a) && (isNaN(b) || b == undefined))
      return true
  }
  return a === b
},
```

```
▼shape {...}
  ▼_Fill AtomicDiff
    ▼newValue Color
        a 1
        b 0.921
        g 0.921
        r 0.921
    ▼oldValue Color
        a 1
        b 1
        g 1
        r 1
      type property
```

Figure 41: The Diff object

To make it easier for users to read a three-way diff and with that resulting conflicts, we flatten these nested properties in the Diff object.
Also, properties are removed or added rarely, for a simple reason: The Lively Kernel allows connecting to properties of an object. Those connections dysfunction whenever their target or source property is deleted. For that reason we do not distinguish between property additions, removals or modifications, but record such changes in Diff objects with the oldValue or newValue set to *undefined*.

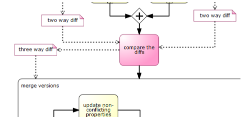


Figure 42: Three-way diff during update process.

Comparing changes: The three way diff The three-way diff is a comparison of two-way diffs. We want to merge properties and morphs based on the merge matrix in Table 2 and therefore first compute which morphs were added, removed or modified in which version.

For reasons of transparency, we combine both Diff lists to one based on following rules:

Definition 1. *Let*

- $A_{\{s,f\}}$ *be the submorphs that were added,*
- $R_{\{s,f\}}$ *be the submorphs that were removed,*
- $M_{\{s,f\}}$ *be lists of AtomicDiffs of the properties that were modified, sorted by morph id in the sibling or the focused version.*

Then the three way diff between them has following properties:

- *addedInSibling* := A_s
- *addedInFocus* := A_f
- *removedInSibling* := $R_s \setminus M_f$
- *removedInFocus* := $R_f \setminus M_s$
- *conflicted* := $\{i \in M_s \cap M_f \mid M_s[i].newValue \neq M_f[i].newValue\} \cup (M_S \cap R_f) \cup (M_f \cap R_s)$
- *modifiedInSibling* := $M_s \setminus conflicted$
- *modifiedInFocus* := $M_f \setminus conflicted$

Preparing for a three-way merge, only those properties that are conflicted according to Table 2 will be presented for the user. Those that were modifiedInSibling, removedInSibling or addedInSibling can simply be added, removed or modified.

Applying change sets As soon as we have computed the three-way diff, we can apply the changes to our local copy of the part. Naturally, this excludes the changes that are conflicting.

Adding morphs that were added in the sibling version simply requires adding them at the correct position, which means finding the correct owner by its identifier that we remembered and adding the morph. Removing morphs works almost the same way: find them by their identifier and remove them.

The interesting part is applying the new values from nested properties. What we computed is an array that holds every layer of the nested property as a string. The change set for the property `'_Extent'` of the `'shape'` of a target morph would then exemplary be

```
{
  property: ['_Extent', 'shape'],
  newValue: pt(0,0),
  oldValue: pt(100,100)
}
```

The function *applyNestedProperty* takes the property array, a starting point as an object (usually the morph) and a value that will be applied (the newValue, in particular):

```
applyNestedProperty: function (prop, pathArray, value) {
  var subProp = pathArray.shift();
  if (pathArray.length > 0)
    this.applyNestedProperty(prop[subProp], pathArray, value)
  else
    prop[subProp] = value
}
```

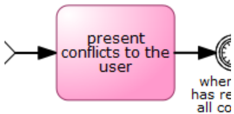


Figure 43: The user has to resolve conflicts.

During the automatic update, all non-conflicting changes are applied in that way.

UI decisions: How to present conflicts Due to our semi-automatic merge strategy, conflicts have to be presented to the user visually. We have two important requirements here:

- each change must be revertible individually
- all changes must be revertible as a whole

Therefore we decided to copy the morph that is updated, apply the changes to it and, if the user is done with merging manually, apply the changes to the original morph. This allows the user to revert single changes, e.g. to test their interoperability. If inherent conflicting changes (such that affect others without directly conflicting) should then make it impossible to merge updates into the local copy of the part, this can simply be reverted by deleting the copy, thus reverting the whole merge action.

For each individual change, we display an indicator for the submorph it belongs to, that, when clicked, shows the diff with the new and old value. The user then can toggle between both possibilities. Figure 44 gives an example of what this looks like.

After the user has resolved all the conflicts, he clicks OK and all the changes he chose are applied to the original morph, and also are those changes that did not conflict. Hence no change is applied to the original morph before this moment, and the user is able to revert the whole update process whenever it is required.

3.5 Evaluation

Time complexity This chapter evaluates our diffing algorithm according to their time complexity.

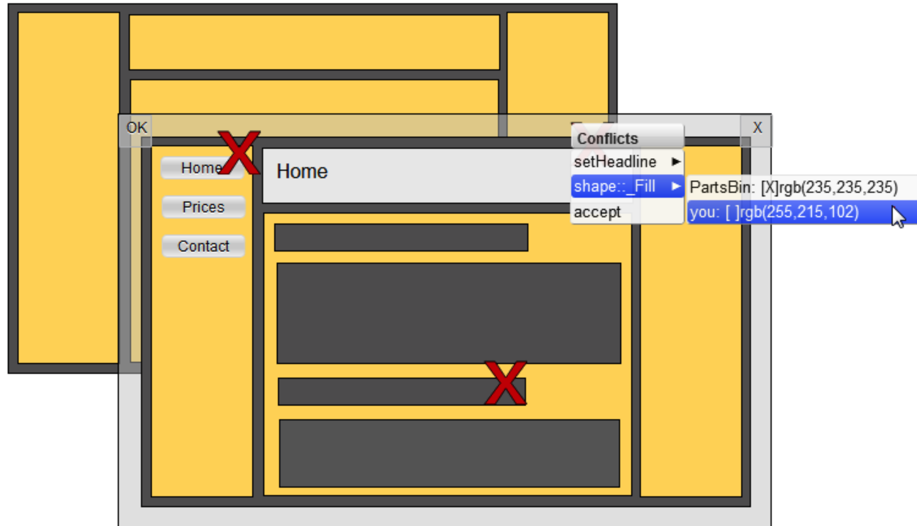


Figure 44: The user interface for solving conflicts. Conflicted morphs are marked with an *X* that, when clicked, displays all the differences.

We evaluate two main parts of the three-way diff: two-way diffing including the relative finding and the actual diff, and the three-way diff itself.

Definition 2. *Let n be the number of submorphs. We define*

- *c to be the constant time to search through all submorphs of a morph. This may differ, as a container morph usually has more submorphs than e.g. a button. To ease calculations, we used submorph structures with an average sumborph count of 2 submorphs per morph.*
- *l be the constant time it takes to linearize a morph without its submorphs*
- *p be the constant time it takes to compare two morphs*
- *t be the constant time it takes to compare two diffs*

Then the time complexity for each part of the diffing process is the following:

- $time(relative(n)) = n * c$
- $time(diff(n)) = n * (l + p)$
- $time(threeWayDiff(n)) = n * t$

This makes the overall computation time for a three way diff $time(relative(n)) + time(diff(n)) + time(threeWayDiff(n)) = n * (c + l + p + t)$ which is in $O(n)$. Figure 45 confirms this calculation showing a linear rising of diffing times.

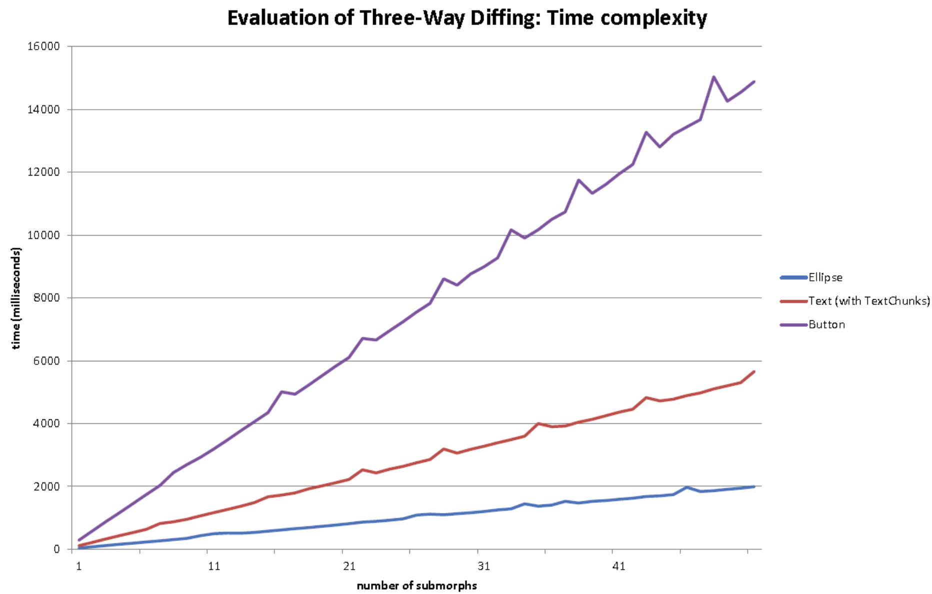


Figure 45: Measurements of the three-way diff

The fact that morph diffing requires an average time of 40 ms up to 120 ms per morph makes clear that, referring to a maximum acceptable response time of 4 s, we would await to be able to compare parts with less than 50 submorphs in an acceptable time. Basic Lively tools like the ObjectEditor or the PartsBin keep an average of 40 submorphs.

A desirable goal should be to reduce the average diffing time for those parts to one second, a quarter of the current results. Trying to find areas that are worth improving, Figure 46 shows that most of the time during a three-way diff is filled by two-way diffing. Figure 47 shows time consumptions within the two-way diff, stating that *Property* and *Function* iterations are very time relevant.

Here we are confronted with the drawbacks of a generous blacklisting algorithm. Via whitelist approaches, such iterations could have been avoided, but the diff quality would have been reduced.

Based on those results we evaluated the efficiency of the update algorithm, measuring the time it takes to call the update command until the conflicts are presented to the user. We tested the ObjectEditor, PartsBin, Explorer and TestRunner by loading the second last version (reseted state) and calling *update* on it. Table 3 shows the results.

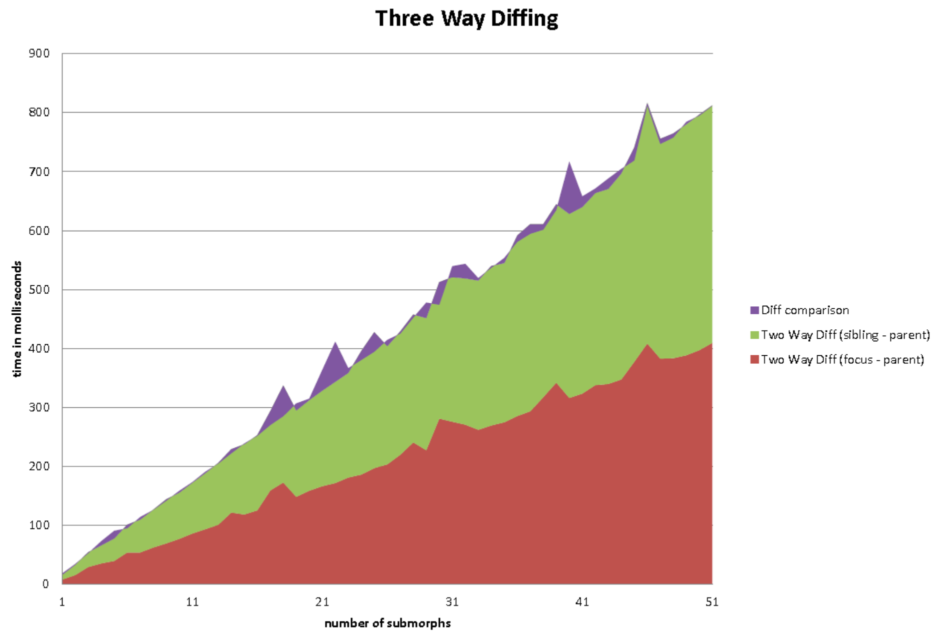


Figure 46: Measurements of the three-way diff

Morph name	number of submorphs	time in milliseconds
Object Editor	40	6643
Explorer	13	1484
PartsBin	40	4931
TestRunner	43	11539

Table 3: Update time of chosen parts.

We can see that complex scene graphs with hard references between their submorphs via properties show an increased average diffing time, but are acceptable with a response time of less that 1.5 sec for the ObjectExplorer. This time already includes the rendering of conflict indicators.

Quality evaluation and subjective durations Apart from the speed of an update its quality is worth being evaluated. We tested our implementation with the challenge described in chapter 3.2.
Like already described in 3.3, we manually reconstructed the correct branch history based on derivation ids. This task took us roughly 15 min, which is an indicator that automatizing the branch history computation is an idea worth thinking through.

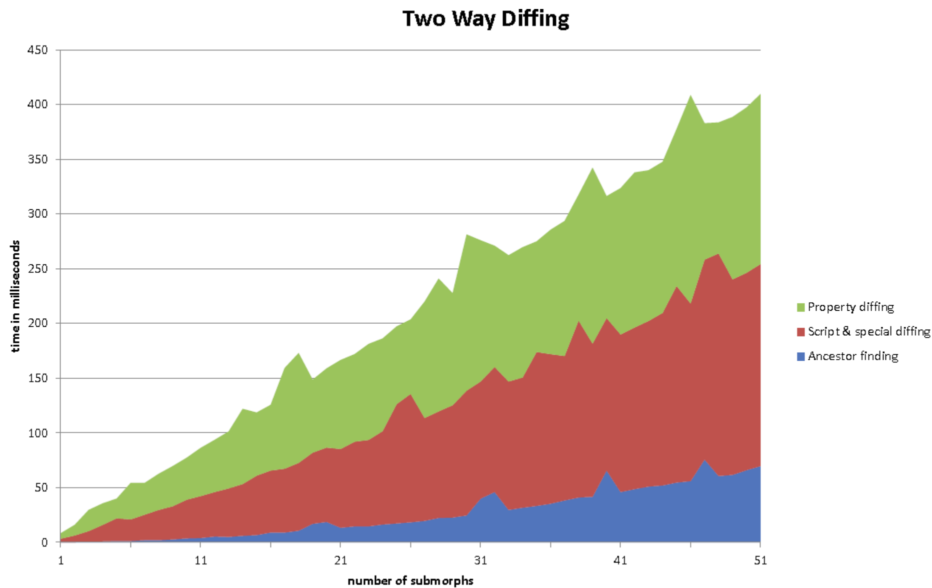


Figure 47: Measurements of the three-way diff

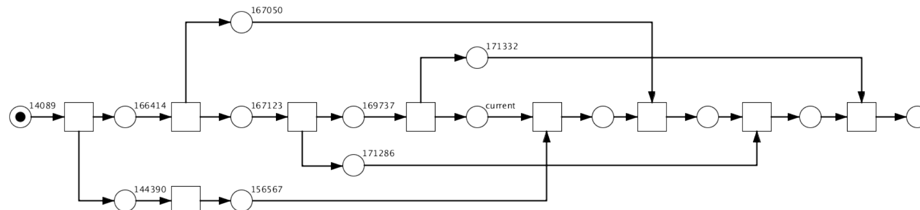


Figure 48: Simplified merge plan for the ObjectEditor challenge

Based on that we created a merge plan that in a lean representation is shown in Figure 48. The merge took us another 15 min, due to several script merge conflicts (Script merges are not included here as we applied them later on with the help of a text diffing example in Lively Kernel). We collected following statements:

- Many conflicts were found that weren't important. Those include invisible extent corrections of Text morphs as well as Position differences of not noticeable pixel amounts.
- Script conflicts were the only really solvable ones. The fact that they are only displayed as strings but not in an additional script environment was a huge drawback.
- Morphs were hard-referencing other morphs, that naturally do not exist in other newer part revisions. This heavily increased the part size of merged versions. More convenient reset functions that refresh those connections could have been a solution.
- A distinction between the different diff types would have been helpful, distinguishing between e.g. morphs, scripts and properties.

Still, the result was a merged ObjectEditor with all the features available.

3.6 Future work

Based on the evaluation, we propose several points of interest in our implementation that are worth being improved. Also, we suggest some features that could make the merge functionality much more powerful and some the merge functionality could support.

Efficiency improvements of part merging First, it is easily possible to extend the algorithm to find out if a morph was moved from one owner to another. This only requires browsing all removed and added morphs and find out if one morph can be found in both arrays. Moving a morph from submorph A to B would then only be a conflicting operation, if in the other version it was also moved or removed.
We also do not consider the order of submorphs. By comparing the submorphs array of a morph with its ancestors submorph array according to the position of the morphs, this can easily be implemented, too.

We did not implement an exact algorithm for text comparison, due to the complexity of RichText TextChunks and the long diffing times (see chapter 3.5). The *equals* functions are already implemented, but they do have to be made faster to be really helpful.
We also did not implement the comparison of connections, as they keep hard links to connection targets. One very general approach to solving that issue could actually be soft links for connections, that find their source and target e.g. by id or name.

We think that a massive improvement to the merging process would be a fully developed script merge. Our algorithm could e.g. use techniques from text-based diffing approaches that are already available in the Lively Kernel. This idea is heavily connected to distinguishing between types of conflicts to support a better understanding of them. By distinguishing between conflicts in morph structure, properties and scripts, this could as well be achieved by distinguishing between the aims of a refinement. E.g., if it was automatically computable if a refinement aims at refactoring or implementing a new feature, a user could choose changes implementing new features over refactoring.
Access to such an understanding could be given by analyzing a list of Diff objects according to the number of added scripts and properties, and those that were modified: A user could await that a new implementation requires more additions and that a refactoring often touches existing properties and scripts.

With derivation ids, object diffing in the Lively Kernel provides a hook to implement automatic merging of global branches, by finding splitting points and ancestors automatically. Also, an implementation of octopus merging [34] that allows merging multiple branches automatically by computing the best merge plan would improve this workflow. The time spent on branch merging could then be reduced by 50% based on our experiences.

World merging It would be a great advantage to extent the part diffing algorithm in a way that it allows world comparison and merging, to support another LivelyKernel sharing concept. As this brings up scaling issues (worlds tend to become larger that parts), the algorithm should be made faster by then. A goal could be to reduce the diffing time for an average morph to 10 ms. As the most time-consuming part of the diffing is the comparison of linearized morphs, this should be speeded up.

Undo/Redo Undo and redo functionality is almost completely missing in Lively Kernel. The classic software pattern is to create undo and redo patches with every command that is processed in a system. Due to the huge variety of commands (see 3.3) this can only be implemented with commands for a relatively small subset of changes, like Dannert stated during his implementation of WebCards [16]. Our diffing approach could then support undo/redo of the remaining property manipulation, as already guessed for OfflineWorlds [15].

3.7 Related Work

Jonathan P. Munson and Prasun Dewan proposed "A Flexible Object Merging Framework" [31] in 1994. They concentrate on the concept of merge matrices and give a pseudo code implementation of their approach.

OfflineWorlds [15] for LivelyKernel is an implementation includes storing changes of morph versions in a world, providing offline support and auto-safe mechanism.

WebCards [16] is an re-implementation of HyperCard, a Macintosh application from the 1990s that can loosely be defined as a powerful, programmable counterpart of presentation design programs. It provides template concepts by stacking cards on a adjustable background. Dannert proposes a command-based merging approach to propagate changes on thebackground to its stacked cards.

Git ([34], git-scm.com) and SVN (subversion.apache.org) are widespread software version control systems. They provide an exclusively file-based approach to version comparison.

Conrad Calmez' implementation of shared workspaces (see section 4) utilizes our provided implementation of two-way diffing to support live collaboration in LivelyKernel.

3.8 Conclusion

Our implementation of part merging fulfilled the requirements for the proposed challenge of merging branches of the ObjectEditor. This almost completes the version control mechanisms of the PartsBin that were missing according to [44].
In particular, we wanted to implement diffing algorithms to compare part versions, updating copies of old revisions and merging branches. We are capable of comparing versions by producing change sets as JavaScript objects (as Figure 41 shows). Our very early decision to implement unique identifiers and storing them as derivation ids turned out to be an excellent method that allows restoring the complete history of morphs and parts. Based on the same general approach, we were able to find ways to determine the kind of interactions with morphs that were added, removed, copied or modified.

Although our implementation aims at asynchronous collaboration, it suited the implementation of synchronous collaboration in shared workspaces (see section 4), so our implementation proved as stable enough to be expanded with automatic merge mechanisms.

4 Design and Implementation of Shared Workspaces in a Mobile and Desktop Environment

4.1 Introduction

Wiki systems [40] offer an easy and accessible way for people to work together. Usually users can create and save pages that can afterwards be edited by other users of the wiki system. With such systems, work can be distributed over long distances. Lively Kernel [30] offers such functionality via two system built-in mechanisms.

First, the system is split into worlds which are part of the Morphic Framework [46] implementation that is a core component of the system. Those worlds can form a so called Webwerkstatt [37] which collects the knowledge produced by the system's users just as other wiki systems do with pages.

Second, the PartsBin [43], as a way to publish written programs to the system, offers an identical benefit on the level of applications.

Working in wikis surely enables users to work together, but the style of collaboration is rather asynchronous since only one user can save the document at the same time. For the ability to work together at the same time[53], there need to be additional mechanisms.

As changes in a wiki can not be seen until someone saves them, duplication of work can happen if the work to be done is not pre-coordinated. But such coordination creates overhead on the process of working and disrupts the workflow of synchronous collaboration. However, the synchronization of content is an important task that has to be done in near real time to create a notion of synchronous work. Consequently, a system for synchronous work should either be on one location[54], so that synchronization does not have to happen, or it has to synchronize the content in a way that creates the least overhead on the actual process.

As Lively Kernel does not offer such functionality, we approached to implement such a system. This bachelor thesis describes how we augmented Lively Kernel's collaboration facilities by creating a new application that enables its users to collaborate at the same time no matter of what place they are. We compared our system with other collaboration systems in the collaboration matrix in figure 49[55]. Since synchronous collaboration is missing in Lively Kernel, most of the focus is on this particular style of collaboration. In general, the intention of our application is to support collaboration for people working on a common goal. That is why the users should not have to take care of the asynchronous collaboration style as well. Therefore, our system has asynchronous aspects as well.

When users come together to work on a common goal, that is called a session. Those sessions can be started if one user opens a new shared workspace within the system. In order not to have to wait for all participants to join a session, the system needs to support asynchronous collaboration styles. Users joining later need the content that has been produced at the time they joined, as well as the updates that happens after joining. As a consequence, our system does not only support synchronous, but also asynchronous collaboration styles.

---

[53] synchronous collaboration

[54] meant is actually one machine / computer

[55] The graphic is based upon the authors personal estimation and is not backed with measured numbers. It only serves for an approximated comparison of systems.

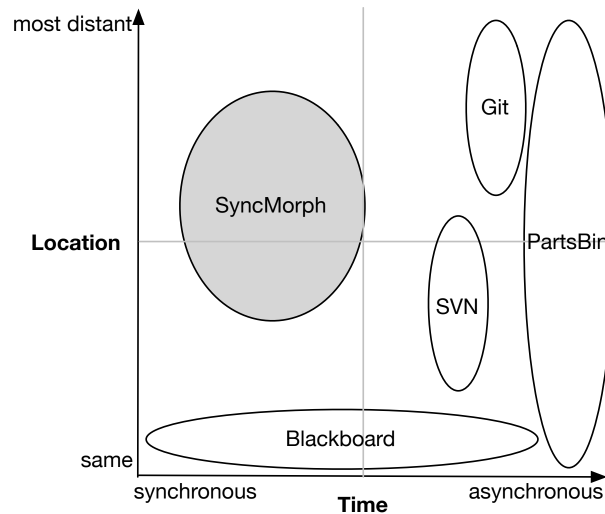


Figure 49: Collaboration Matrix; The SyncMorph is the system of interest in this thesis. The PartsBin is part of Lively Kernel.

Collaboration systems are very diverse in terms of how they use computers to support the work of a team. One natural approach is to physically share one space. This has the side effect that work can not be distributed over a distance. Since Lively Kernel is a Web application that inherently has users all over the world, it should be possible to distribute work all over the world as well. Hence, an important key assumption of our system is that each user is working on his or her own machine. Conclusively, our system embraces distributed as well as co-located work.

As our system is based on the usage of individual computers it needs a synchronization mechanism. To synchronize the content, one needs to identify the content's benefit.

As the Morphic Framework is a core functionality in Lively Kernel it is understood that one wants to distribute morphs via our system that we called SyncMorph. To make the workflow easy, we decided to use interaction that a user of Lively Kernel is used to. In addition, our system is implemented as a morph. This suggests to use morph interaction. Thus, the users can drop a morph onto the SyncMorph and it will take care of the distribution to all other connected clients. Furthermore, if one user removes a morph from the shared workspace the corresponding morph representations on all other clients get removed.

Sharing content is a way to distribute knowledge, but if we want to enable users to work together, they need to be able to alter the shared contents. Consequently a more enhanced version of this SyncMorph also supports the editing of morphs on the shared space. In the interest of making changes to the synchronization state immediate and foreseeable it is also possible to see a morph as another user drags it onto the workspace of the SyncMorph.

Working with Morphic enables users create UI and behavior of their application. Nevertheless, sometimes an idea is not as concrete as it could be directly implemented. Therefore we decided to add drawing support to our collaboration system. The ability to draw enables users to sketch ideas. The drawings get synchronized as well so that ideas can be developed together. In order to create some freedom for the user while drawing, the system allows the customization[56] of pencils.

Having such a system, which synchronizes all contents to all clients, empowers the users to have a common sense of how far the progress of work is. However synchronization is not enough to create an awareness of what each collaborator is currently working on. In addition to the synchronization of contents, further mechanisms are needed for a system that should support synchronous collaboration. Collaboration on a physically shared workspace does not have those problems. Looking at why collaborators are aware of each other's work using such systems, can lead to a solution for a distributed system.

In a physically shared workspace like a large table, each team member is aware of the area in which the others are working by seeing them work there. The current working area in the digital sphere is where a user has its mouse pointer or finger[57]. By synchronizing the position of the mouse pointer or respectively the finger, an equivalent is found for distributed systems. Assigning each user a unique color, that is different enough from the others, enables the users to identify each of their collaborators.

Besides, communication on a table is uncomplicated as one just says something out loud to pass the message to everyone. This kind of instant communication can be achieved via a group chat functionality that we implemented as well. Further our intstant messaging chat has the benefit that the message is only passed to everyone at the same table, but not in the same room.

The following chapter will describe our approach and the problems to solve. The succeeding chapter will explain the implementation of the system. The fourth chapter will evaluate the results by looking at some usage scenarios. Afterwards, the system will be evaluated performance wise. The sixth chapter will present related work. After that, we will give an outlook on how the system could develop. The last chapter will summarize the findings of this work.

4.2 Approach / Problems

When people work together, they naturally gather at one place. Consequently, meetings in the virtual world need to evoke the same advantages. Especially this has to be done in order to escape the need of manual synchronization by the users. The term of the shared workspace in this thesis is meant as an analogy to the physically shared workspace as it is the model for the system that we built. An example of the user interface in action can be seen in figure 50. This chapter describes the problems that occurred while implementing our system. Additionally, it explains our approaches to solve the shown problems.

In pursuance to make a system that supports distributed, synchronous collaboration work well it should be responsive. For responsiveness, it is crutial to synchronize the state or the content in a relatively small time interval.

[56] color, alpha, size, stroke style

[57] on a touch device

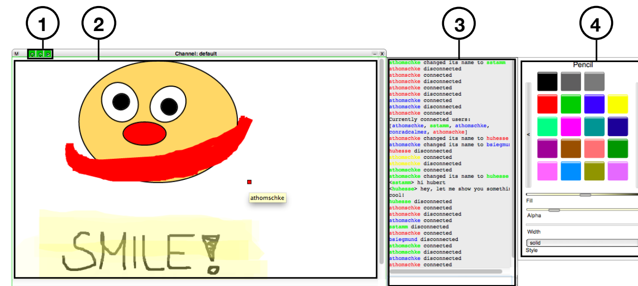


Figure 50: User interface of the SyncMorph with (1) buttons to toggle and indicate connection state, chat pane and pencilstyler (2) synchronization pane, (3) chat pane and (4) pencil styler

A high-level overview of our applications functionality is shown in figure 51. It is based on message exchange to communicate modifications of the content. The figure shows an example of how content can be produced synchronously on two independent worlds.

Client-Server Architecture We decided to implement the application with a client-server architecture. The decision was made for the following reasons:

First of all, the server as a central unit knows all clients. Having such an actor makes it easier to implement the distribution of messages. Althought, this does not say that a distributed networking system might not be equally well. Our solution was just the most suggesting one with the used technology.

Moreover, as the server receives all messages from all clients it is possible to store the input data into a persistent storage. At the moment, the persistence is realized by clients holding a certain state. This concept of persisting state in clients would of course work with a distributed system, but in this case one loses the central server as an additional archiving unit.

Beides, implementing a distributed system without central infrastructure might be convoluted in Javascript.

The client-server architecture also enables the implementation of alternative clients that communicate over the same server. Since the server's purpose is only the message distribution, and possibly persistence, there is no need to replace it with an alternative implementation. Alternative clients could for example implement a different drawing algorithm for the purpose of interoperability. Here again a set of polymorphic clients without a central server unit is possible as well.

Furthermore, the server can act as a centeralized mixer [25] for messages [50]. This can save bandwidth to increase the performance when using slower connections.

Data Exchange Format We decided to define a specific exchange format for message exchange. With this format it should be possible for the user[58] to decide who will get the message that is going to be sent. In favor of applying[59] messages in the correct order it is also possible to augment the format in that way so that each message contains an ID and a timestamp.

[58] in this case a client application
[59] or resending

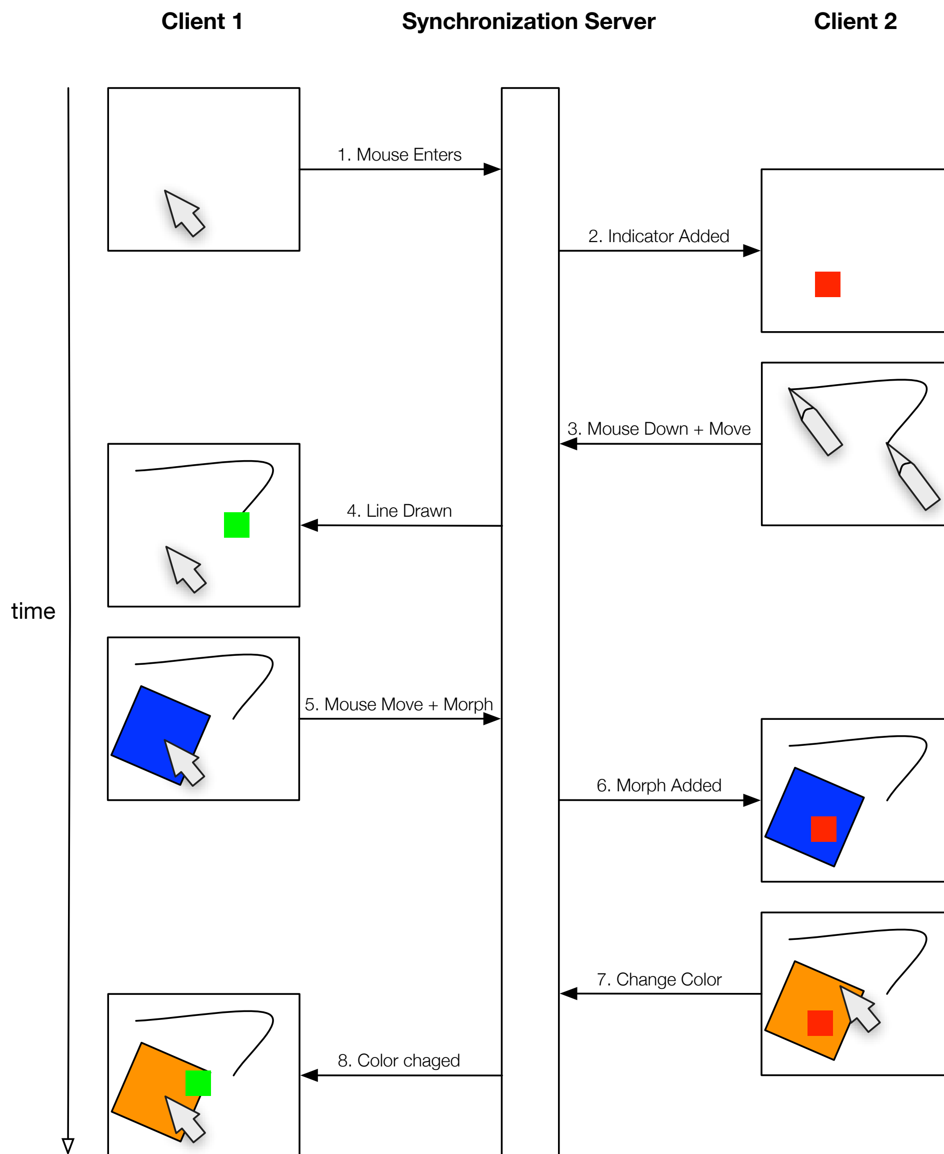


Figure 51: High-level overview of functionality

The message packages itself are serialized JSON objects. The message format looks like the following in code example 17.

```
{
        message: "message content (not necessarily a string)",
        me: false,
        broadcast: true,
}
```

Code Example 17: simple version of the exchange messages

The message content can be string or a data object itself. The properties *me* and *broadcast* specify to whom the server should send the messages. Whereas *me* set to true means that the message will return to its original sender. Moreover *broadcast* means that the message will be sent to every user being connected to the same channel as the original sender of the message but the original sender itself. These options can be extended by a new property *broadcastType* which will determine on which level the broadcast will happen. This option should default to `"channel"` with which it will behave as described above. Setting the option to `"global"` would mean that the message should be sent to every client connected to the server except the client who initially sent the message. An extended version of this exchange format can be seen in the following example.

```
{
        message: "message content (not necessarily a string)",
        id: "unique message id",
        time: 1234567890,
        me: false,
        broadcastType: "global",
        broadcast: true,
}
```

Code Example 18: extended version of the exchange messages; properties id, time and boadcastType are added

Synchronization As our system should support distributed, collaborative work, a synchronization mechanism is needed. This section examines the synchronization process. Figure 52 shows an overview of the event – method mapping for the different kinds of synchronization that are used within the application.

Scope of Synchronization Since work in Lively Kernel happens on so called worlds, it is suggesting to synchronize whole worlds. The disadvantage of this approach is that this would prevent the presence of private workspaces. Private workspaces are important for various reasons. Users might feel a disturbance of their private sphere if all their thoughts and work are synchronized immediately to all other team members [55]. Consequently, a good system has to offer the possibility to decide which content should be synchronized. That is why we decided to implement our system as an application that can be loaded into every world to augment it with the collaboration facilities our system is offering. All data and objects outside of the application then belong to the private workspace of the user. By dragging objects into the application they and all actions on them get synchronized. Therefore, we build a clear boundary between private and shared workspaces. Further, all work within the world remains the same just as the users are used to.

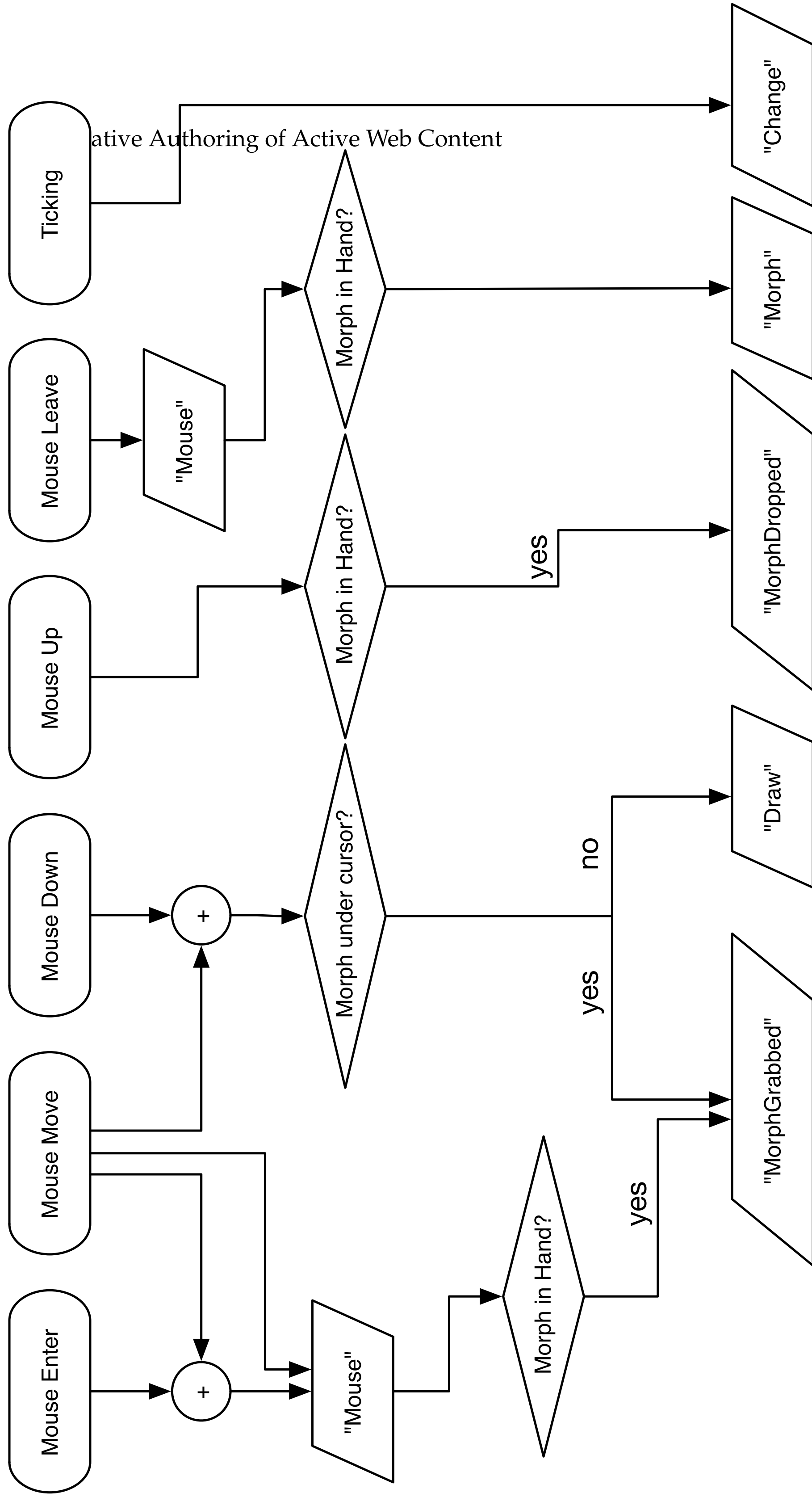


Figure 52: event – synchronization method mapping

However the development of a clear synchronization boundary makes the application logic, and by that the implementation, more complex. Since our system is just another application within Lively Kernel it is part of a world as well, just like every other morph. The contents of the SyncMorph are created in one world. They can have references to other objects in the same world. If a user now decides to synchronize such an object having references to other objects in his or her own world the problem will arise that this reference is actually missing in all other worlds where the object was added by the synchronization algorithm. Naively, it is possible to implement a reference tracer in order to synchronize the references as well. The problem is that the world is a morph as well which can be referenced[60]. As a result, only one reference to the world would cancel out the concept of this synchronization boundary, because the world would have to be synchronized as well.

Synchronizing the world brings further problems. Since there is a world on every client, and there should only be one world, the algorithm would have to merge as many worlds as there are participants in one session. Moreover, the SyncMorph is part of the world as well ergo it would have to be synchronized as well. This would lead to an infinite recursion.

In conclusion, we decided not to synchronize references that point to objects which are not in the space of the shared workspace. The topic of references is part of the future work.

Synchronization of Generic Content For the synchronization of generic objects we needed a serialization algorithm. The first possibility is to implement an algorithm that uses a whitelist to create a serialized object with certain properties (e.g. the visual properties). On all other clients, an object of the same class would be created. The new properties would be applied to this object. Even if this approach needs less network bandwidth, the focus is placed on certain properties which makes the application harder to extend. To support additional properties, the serialization algorithm must be reworked.

Instead, the approach we took for serialization uses Lively Kernel's serialization algorithm to store worlds and morphs. The advantage is that the whole object is serialized, no matter what it looks like. This approach uses more bandwidth since more information are transmitted. Initially an object gets serialized via this algorithm. Afterwards only changes (see section 3) on the object graph get synchronized in order to save network bandwidth.

Generic objects and all changes on them are synchronized in an optimistic way which means that they are first executed on the client and then communicated. This has the benefit that the responsiveness of the system does not change by a considerable amount.

Synchronization of Specific Content For specific content with a limited benefit for the user we took a different approach. Drawings are an example for such content with limited benefit. Communicating whole objects would use more network performance than needed in this case. Restricting the sent information is acceptable, since the possiblities to extend the drawing of lines are endless[61]. We consider the benefit of saving bandwith here to be higher than the need of extensibility. The clients receiving such specific information recognize what has to be done by the kind of event that delivered them. By that our client application is a thick client [25]. With this, it is possible to create a responsive drawing surface.

[60] like any other morph
[61] We augmented the sent information by line style, width and color

Specific content is again optimistically synchronized to keep the corresponing interactions responsive.

Creation of Awareness Working together with a software that synchronizes its contents can sometimes be surprising as a remote user makes an action whose result is displayed on the user's client. To reduce the effect of being surprised, the user actually has to see the action happening on the remote client. Besides, the user should be able to anticipate what action a collaborator will do. This concept is called awareness. If a user is not surprised by what happens on the synchronized board he or she will be able to plan his or her own work in a better way. Good planning of work by each user also leads to less editing conflicts as well as less duplicate work. Consequently, awareness is an important concept in collaboration systems [17,18]. We implemented two concepts to create awareness.

Telepointers Looking at how users anticipate what and where a co-worker is doing something when working at a physically shared work space[62], we found a solution for the anticipation problem. When a user sees another one approaching a certain object on the work space he or she will think that it is likely that his or her co-worker is going to interact with it[63]. Furthermore in a physically shared workspace users are able to point on objects to talk about them.

Both actions are performed with the hands of the user. Consequently, it is suggesting to synchronize the representation of the hand in a distributed workspace. By doing so, a user gets a telepointer for each collaborator.

For the sake of being able to distinguish different users, each telepointer has a unique color that is different enough from all colors of the other collaborators. Since we have a synchronization server as a central communication unit that knows all clients we implemented the user – color mapping there. When a new user logs into a shared workspace the server will assign a new color to this user.

Telepointers are, like drawings, specific content that is optimistically synchronized with a reduced set of information since the mouse and touch interaction needs to be highly responsive with respect to a usable system.

Chat With the given system, it is straight forward to design a chat application. We use our data exchange format to send textual messages between users. To enable the user to see if the message was really sent, the messages are pessimisticly synchronized. This means that a message is first sent to the server which distributes it to all clients. Each client executes the action corresponding to the type of message.[64] So if the connection is broken, the user will recognize it by seeing that his or her chat message does not show up in his or her chat client. The server itself stores chat messages not only by saving the user name - message mapping. It extends the information by the time the message was received by the server. With that, a chat log is created on the server that can be replayed to clients that have been offline.

---

[62] such as a table or a whiteboard

[63] e.g. draw on the area; manipulate the object; move the object

[64] In case of a chat message this would be to display the message and the username in the chat GUI.

4.3 Implementation

In this chapter, the implementation of the system is illustrated. The implementation is described feature by feature. For each feature we discuss the benefits it conducts and the limitations that remain. The features can be divided in three main categories of features: synchronization of morphs, synchronization of drawings and awareness features. Each feature set is part of the same software system.

Synchronization of Morphs As morphs are the objects with which the users implement their applications, it is important to use the synchronization approach of generic content in order not to restrict the possibilities Lively Kernel is offering.

Fast Sharing At the time we implemented the first version of the SyncMorph, we routinely used CouchDB [4] for storing data that we created in Lively Kernel. Consequently, we used CouchDB here as well to store serialized morphs. The serialization was already implemented and we just had to use it. It basically linearizes the object tree and writes it into a JSON string.

On top of serialization, we needed some interaction event on that we would start the synchronization process. Here again we were looking for something that Lively Kernel was already offering. Since our application was implemented as a morph, we wanted it to be as simple as adding another morph to it. After a morph is added our application should take care of its distribution. The *onDropOn* method seemed to be a good place to hook into since it is called on the target morph when a user drops a morph onto it. We patched the behavior to the class *Morph* with a layer. Code example 19 shows this. If a morph is dropped onto the SyncMorph, the method *saveMorph* will be called on the SyncMorph. For convenience, we wrapped the usage of the built-in serializer in a *serialize* method on the class *Morph*.

```
1  module('projects.BP2012.SyncMorph').requires().toRun(function() {
2    cop.create('SyncMorph').refineClass(lively.morphic.Morph, {
3      onDropOn: function (aMorph) {
4        if (aMorph.saveMorph) {
5          this.disableHalos();
6          aMorph.saveMorph(this);
7        } else {
8          this.enableHalos();
9        }
10     },
11     serialize: function () {
12       // ... implementation intentionally left out
13       return serializedObject;
14     },
15   });
16   SyncMorph.beGlobal();
17 })
```

Code Example 19: extension of class Morph

The functionality of saving a dropped morph to the database is of course implemented on the SyncMorph. Apart from saving the morph to the database, metadata will be added to the morph object in order to be able to manage[65] the content as updates come in.

```
function saveMorph(aMorph) {
  aMorph.databaseID = undefined;
  aMorph.databaseRev = undefined;
  if (this.active) {
    var newMorph = aMorph.serialize();
    var result = this.getDB().save(newMorph);
    if(result.error=="conflict"){
      alert("an error occured while synching the morph");
    } else {
      aMorph.databaseID = result.id;
      aMorph.databaseRev = result.rev;
    }
    this.updateDBObjectIDs();
  }
}
```

Code Example 20: save functionality of SyncMorph

After this iteration, we were able to distribute morphs fast between worlds without using the PartsBin or a saved page. Though the conflict avoidance strategy was to disallow interaction except grabbing for the morphs on the SyncMorph. So for being able to edit the morph, the user had to grab it and drop it into his or her own world to enable the editing features again. As a consequence synchronous collaboration was not possible with this revision, but we fastened up the sharing process.

Message Passing Since the CouchDB server was quite slow if the SyncMorph contained many morphs, we were looking for a faster alternative. We decided to do the message passing ourselves and implement a synchronization server. We used node.js [59] for that. Fortunately, Lively Kernel is offering a mechanism to create and run node.js servers. This made it easy to develop this application completely within Lively Kernel.

The implementation of the synchronization server was straightforward since node.js and socket.io [51] offer all features we need for a message passing server. First we need to hold a reference to all clients that are connected to the server. Since node.js abstracts different technologies to sockets, it was easy to accept incoming connections and close them again as the client disconnects.

[65] e.g. delete the morph if it was deleted on another client

The channel feature of socket.io was a way to implement different workspaces within one system. Also, it was easy to send a message to all clients of a certain channel, as the library offers a broadcast mechanism that sends a message to all clients[66] of a channel or the whole system. Further, we needed to define different message types for different actions that should be synchronized. Figure 21 shows an exerpt of the implementation of the synchronization server. We wrapped storage and message broadcasting functionality in an object called *WhiteboardServer*. The set of events that we need for the communication between clients was defined with the *io* object of socket.io. Lines $16 - 18$ show an example of such an event definition. The first parameter of the *socket.on* call is the name of the event. Additionally, the second parameter is the functionality description of the event as a Javascript function.

```
1  WhiteboardServer = {
2    port: 4000,
3    // ...
4    send: function (socket, channel, messageType, data) {
5      if (data.broadcast) {
6        socket.broadcast.to(channel).emit(messageType, data);
7      }
8      if (data.me) {
9        socket.emit(messageType, data);
10     }
11   },
12 }
13
14 io.sockets.on('connection', function (socket) {
15   // ... code intentionally left out
16   socket.on('ping', function (data) {
17     WhiteboardServer.send(socket, /*channel*/, 'pong', data);
18   });
19   // ... code intentionally left out
20 });
```

Code Example 21: code exerpt from synchornization server implementation

The version of our application after this iteration improved the performance of the application when it contains many morphs. Still, the synchronous collaboration features were not present at this time.

[66] except the sender

Enabling Collaboration Finally we decided to integrate a diffing and merging algorithm for objects (see section 3) in order to be able to only communicate changes instead of whole morphs. Unfortunately an event for a change that happened on an object is missing in Javascript. That is why we used a ticking script that observes the changes on all morphs[67] that are synchronized via the SyncMorph. This ticking script is executed in a well defined time interval. It performs a diff to the current version of a morph. To have a reference to compare to, we copy the morph after diffing as a current version.

When a change is detected, it is sent to the synchronization server that distributes that message to all other connected clients. A client receiving a change merges that change into the regarding morph.

After this iteration the SyncMorph is finally capable to support synchronous work. The users will get updates of the morph shortly after they are done on the client of a collaborator.

Synchronization of Drawings Drawings are an example for content that has a well defined set of features and which therefore can be synchronized as specific content in order to save network bandwith and by that improve the responsiveness of the system.

Drawing on Canvases The drawing facility needs to have a surface on which the users can draw. We used the HTML5 canvas element for that purpose. The canvas element offers an API to draw on it. Since Lively Kernel did not include a canvas morph at the time we implemented this, we needed to create such a morph. Fortunately, Lively Kernel is offering a flexible mechanism to create morphs. The only task we had was to create a canvas element that we used as the shape for the morph.

For the ablity to send events to the synchronization server, we encapsulated the API calls in own methods on the new morph that we called Whiteboard. Conveniently, the API functionality of the canvas element matches our drawing metaphor as it has a function *lineTo* that acts as you would have a pencil. You can call it several times giving it a point on the canvas in order to draw a line. The last drawn point is the point where the next stroke would start as if there would be a virtual pencil. Aside from that function *moveTo* that sets the position of the virtual pencil to a given point on the canvas.

In addition, we defined which interaction event[68] calls which API call.

First a client application sent out whole lines. This had the consequence that lines popped up at remote clients. When drawing larger shapes this actually spoiled the process of drawing together as the collaborators can not anticipate where another user is drawing. To solve that, we implemented that each stroke on one line was synchronized. Consequently, if a client receives a message from the synchronization server, it will call the assotiated method on itself to draw the stroke.

With this implementation the users are able to make drawings together. Unfortunately, the canvas does not seem to be fast enough when synchronizing single strokes of a line. More importantly, this implementation does not work on the iPad. That is why we decided to use a different technology for the drawing surface.

Drawing Lines Lively Kernel supports SVG rendering[69] which was the reason why we decided to use SVG paths for the drawings.

---

[67] Morphs are usual Javascript objects as well.

[68] mouse down, mouse move, mouse up and touch start, touch move, touch end

[69] SVG rendering was actually the default rendering mechanism before HTML rendering was implemented

As SVG does not have a convenient API for drawings such as HTML5's canvas element, we implemented a workaround in order to map the code to the drawing metaphor. When an event arrives, that starts the drawing of a new line[70], a new SVG line morph is created. With each arriving event[71] that indicates the continuation of the drawing, the current position of the mouse or finger, which is stored in the regarding event, is added to the vertices of the SVG line.

Additionally, we abstracted the call of drawing functions and the regarding events that should trigger them. For example, we mapped the mouse move event to a method that handles the process of drawing a new stroke. When we added touch interaction we only had to map the touch move event to the same method.

Actually drawing at the same time on different clients requires that each message contains a point[72] and an identifier for a line. By providing these information, the client application is capable of drawing multiple lines at the same time.

This version of the Whiteboard offers a way to draw together on a morph with a better performance than the previous iteration.

Additional Features To better support the expressiveness of the drawings, we implemented a virtual pencil that could be styled in different ways. We implemented the customization of line thickness, color and line style. With that users have a versatile tool at hand for different drawing tasks.

To implement this additional feature, we only had to set the width and color as well as the style of the border of the SVG path. Moreover, we built a GUI tool to set all those parameters in a convenient way. This GUI can be seen in figure 50 on the right.

Finally after this iteration we have a software that supports the collaborative drawing on desktop and touch devices. Likewise, simultaneous drawing on different clients and with a customizable pencil style is possible.

Awareness Features To increase awareness for the people collaborating via our application we implemented some features described in the following.

Telepointers Javascript does contain events for mouse interaction and Lively Kernel does contain events for touch interaction as well, which is an enhancement we made that is described in the bachelor's thesis of Sebastian Stamm (see section 2). Looking at the move events for each interaction method one gets the current position of the representation of the hand in the system. Since telepointers are specific content, they are synchronized optimistically with a limited set of information. The mouse event that is distributed by the server to its connected clients contains the position and identification information for the telepointer. An example of how a mouse message event looks like can be seen in code example 22.

The remote clients create and display a telepointer for each other client. With the identifier that is provided in the mouse message, the corresponding indicator can be found. With each arriving event the remote clients update the position of the corresponding indicator.

---

[70] mouse down + mouse move or touch start
[71] mouse move or touch move
[72] that should be added to the vertices array of the SVG line

```
{
  message: {
    indicator: 120938479283,
    position: {x: 42, y: 23}
  },
  me: false,
  broadcast: true,
}
```

Code Example 22: an example of a mouse message event

Chat The implementation of a chat system was straightforward, since there is already synchronization server that can distribute messages, and that is already able to communicate complex information. On the server side a new message type had to be added that does nothing but simple routing of messages to the clients.

On the client side, the sending and displaying of messages had to be implemented. For that, we build a separate GUI pane that users can trigger from the main view.[73] The chat pane consists of two elements for its two tasks: First, an input field where users can type in text and send it by pressing the return key. To have this behavior implemented, we watched on each key stroke[74] if the return key was pressed. Second, a pane to display incoming messages. This message log was realized by taking the data that is coming in and concatenating the message's content to the current content of the text pane.

In order to support shortcut commands for power users, we added commands that can be entered in the same input field. Every command begins with a slash that is followed by the name of the command. The commands are stored in an object that has the command name as a key and the description of the functionality of the command as a value. The key is a string and the functionality description is a Javascript function. An exerpt of the definition of this object can be seen in code example 23.

Having this object available, the functionality gets called by accessing the commands object with the command string the user had just entered as a key and calling the *apply* method on the function that was returned. The function will be applied to the chat pane so that the description of a command must be written with that in mind. All parameters given by the user will be routed to the function that is called on the chat pane.

Furthermore, the convenience feature of having a history of the entered text was implemented by again watching the pressed keys. If the up or down keys were pressed, the user will be able to cycle through an array of the entered messages. Those messages were saved by pressing enter[75].

4.4 Examples and Scenarios

This chapter points out some scenarios of usage that will be discussed on the basis of the current version of the collaboration system we implemented.

[73] This can be seen in figure 50
[74] onKeyDown
[75] The same mechanism as sending messages

```
this.commands = {
  'nick': function (name) {
    this.whiteboard.setUserName(name);
    this.showMessage("changed nick to " + name);
  },
  'names': function () {
    this.whiteboard.getConnectedUserNames();
  },
  'channel': function (channel) {
    this.whiteboard.setChannel(channel);
  },
  'chan': this.commands['channel'],
  'clear': function () {
    this.whiteboard.clear();
  },
  /*
    ... more commands intentionally left out
  */
};
```

Code Example 23: exerpt of the command object

Exchange of Content Between Worlds Let us assume the following collaborative scenario: Two users working in Lively Kernel on two different worlds want to exchange ideas in form of an application written within the system. Having the PartsBin, the creator of the application can publish it. The other user can from now on load it using the PartsBin.

This process invokes several problems when working together closely. First, the overhead of publishing the application and writing a commit message might be too high to justify the benefit of sharing the work. Second, the user that shall have a look at the work of his or her coworker might not want to invest the time to search the application in the PartsBin in order to load it. Besides, the progress made on the implementation might not be in a state where one wants to publish it to a broad audience[76].

Consequently, a fast exchange of applications[77] should have the least overhead to share the application with others.

The SyncMorph implements this workflow by letting the user simply drop the morph into the synchronization pane. The system will take care of the synchronization. As an effect, the morph appears in the coworker's application without the need that this person does anything except being connected to the server.

Developing Ideas Together Working with a system that follows the metaphor of a wiki, a user will create content on its own and save the content page in the interest of making it available to other users of the wiki system. The collaborators can open the saved page to see what the user created and add their own ideas to it.

---

[76] This is what the PartsBin actually does.
[77] generally Morphs which are Javascript objects

This kind of conflict handling is called “Single Active Participant” [20]. This process is relatively slow since every participant has to wait until someone saves the page to add own content to it. If the style of work is highly asynchronous, this will not be a problem. But as the system should support synchronous work, this approach does not fit the requirements of simulaneous editing. Figure 53 shows the difference of asynchronous and synchronous styles of working together.

Consequently, conflict handling in our application is not done in such a blocking manner. Our approach is aiming at group dynamics to solve editing conflicts by giving each user the information he or she needs to know where and what his or her coworkers are doing. Furthermore, communication is important to create group dynamics. That is why we implemented an instant chat that is located next to the synchronization pane. Besides, the actual editing happens in a fully synchronized way. If a person makes an update this change will be sent to every other client that is connected. With such a level of synchronization, we create the feeling that the group is working on the same content.

Communication Within the System Communication in wiki systems is often done via comments that are just another variation of content of a wiki page. Consequently, this comes along the same problems as other content[78] when working synchronously.

In favor of a synchronous working style, the system should distribute those messages instantly. As mentioned in the previous section, the SyncMorph implements this instant messaging with a chat interface that distributes messages to all users of a channel.

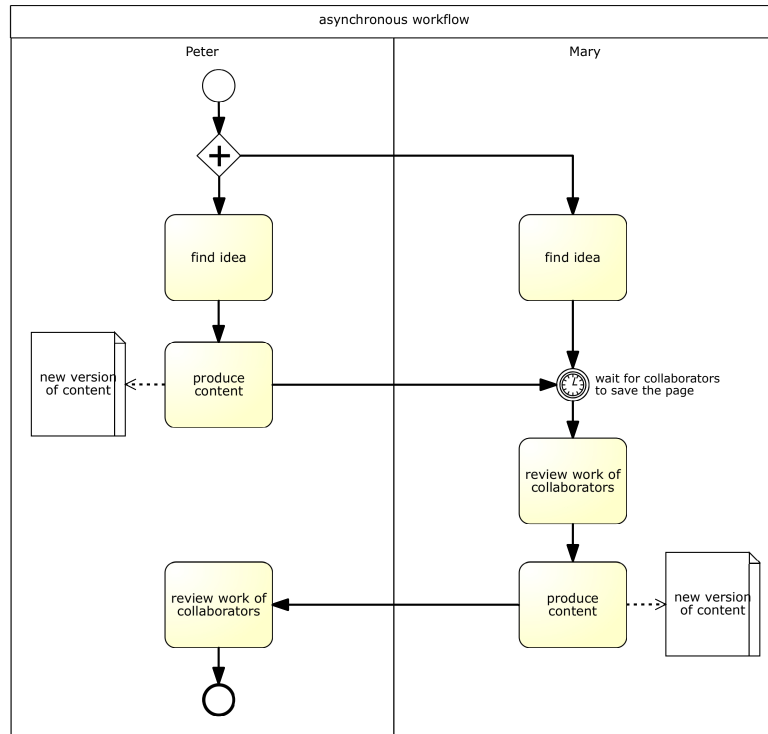
4.5 Performance Evaluation

This chapter deals with the evaluation of the performance of the system. Since synchronization of contents is time critical for synchonous collaboration, the focus lies on how fast messages are exchanged between clients using differnt networking technologies. The experiments were all done using the same laptop computer. This machine was connected over WiFi to an access point that was connected to the internet using one of the following technologies: local network[79], DSL, UMTS-Broadband, GSM. Further, the experiments were realized using two clients connected to the server. Both clients ran on the same machine and used the same networking technology.

For the sake of being able to interpret the results better, we first measured how long it takes for a simple message to get from one client to the other and back. For this roundtrip, we sent a *ping* message through the synchronization server to the other client. The other client then responded with a pong message. As the pong arrived at the original sender the time measurement was over. Figure 54 shows the results of the measurement.

The different technologies performed as expected. The roundtrip time in the local network was 5.4 ms in average. Consequently, a message arrived at the other client after approximately 2.7 ms. The other technologies performed worse as expected. For the DSL internet connection, the roundtrip time was 172.4 ms in average which means that a message was received by the other client after approximately 86.2 ms. UMTS-Broadband[80] was not that much slower than DSL with 232.7 ms in average for a roundtrip meaning that after approximately 116.4 ms the other client received the message. Using GSM as an internet connection slowed the roundtrips of messages down to 4346.5 ms in average. Consequently, a message arrives at the other client after more than 2 seconds.

[78] see previous sections
[79] refers to the same network the application server is in
[80] also known as 3G or HSDPA = High Speed Data Packet Access



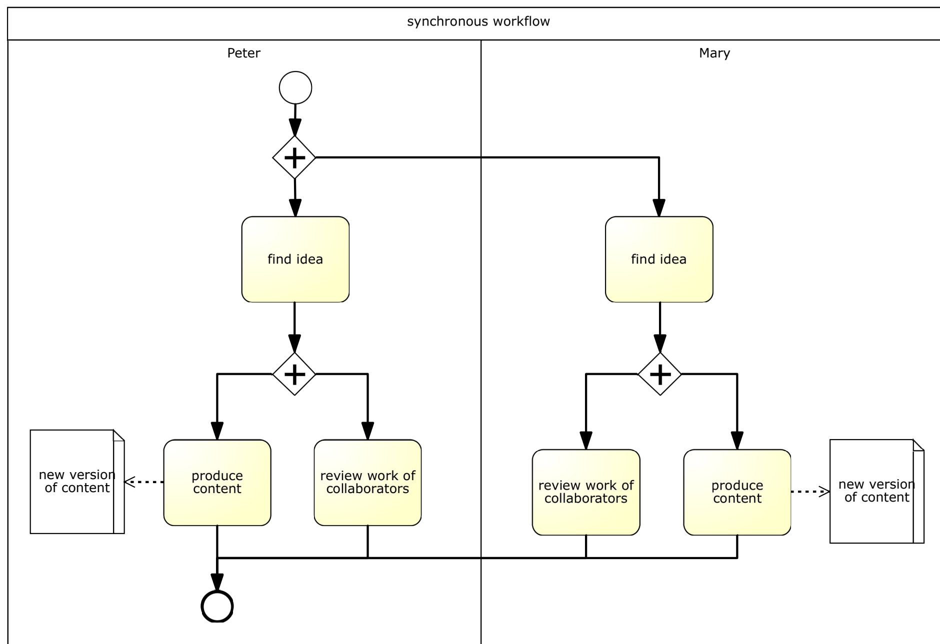


Figure 53: Examplary comparison of (1) asynchronous and (2) synchronous collaboration style

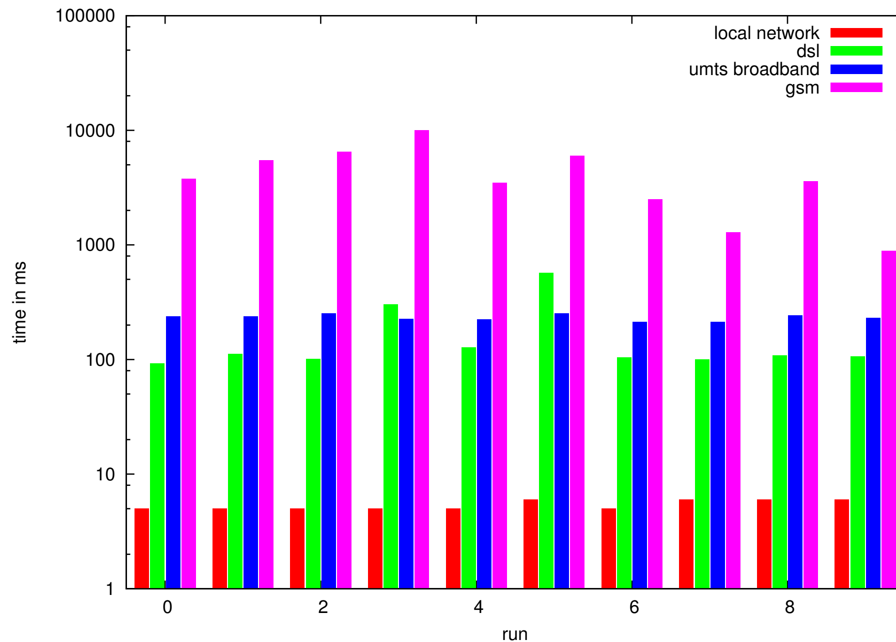


Figure 54: Roundtrip time of a simple ping message through the synchronization server, this diagram uses a logarithmic scale on the y-axis

For messages with a considerable amount of content the throughput of the network connection is important. Using the system ourselves we figured that a synchronization time around 500 ms seems to be enough not to break the synchronous workflow.

Adding content to the messages has an effect on the timings of the messages. The roundtrips in average were: 90.3 ms for the local network, which results in a time of 45.2 ms that the other user needs to wait for message arrival; 466.7 ms for the DSL connection, so that messages arrive at their destination after approximately 233.4 ms; 564.6 ms for the UMTS-Broadband connection, so that messages arrive after approximately 282.3 ms; 7569.3 ms for the GSM connection, meaning that messages reach their goal after approximately 3784.7 ms.

Conclusively, the throughput on connections using the local network, DSL or UMTS-Broadband is good enough to use the system. Whereas, connections using GSM are definitly too slim to use them for synchronous collaboration.

There are also messages that show a continuous action like the mouse moves on a client that are represented by telepointers on the other clients. For those messages, the throughput of the network connection is not important but speed of the network[81] Those message are not as large as a message with a morph as content since they basically only contain a point that says where the mouse pointer moved. But since the action that is shown is continuous and the users expect it to be continuous, the time it takes to synchronize should be low.

[81] How fast messages arrive at the destination.

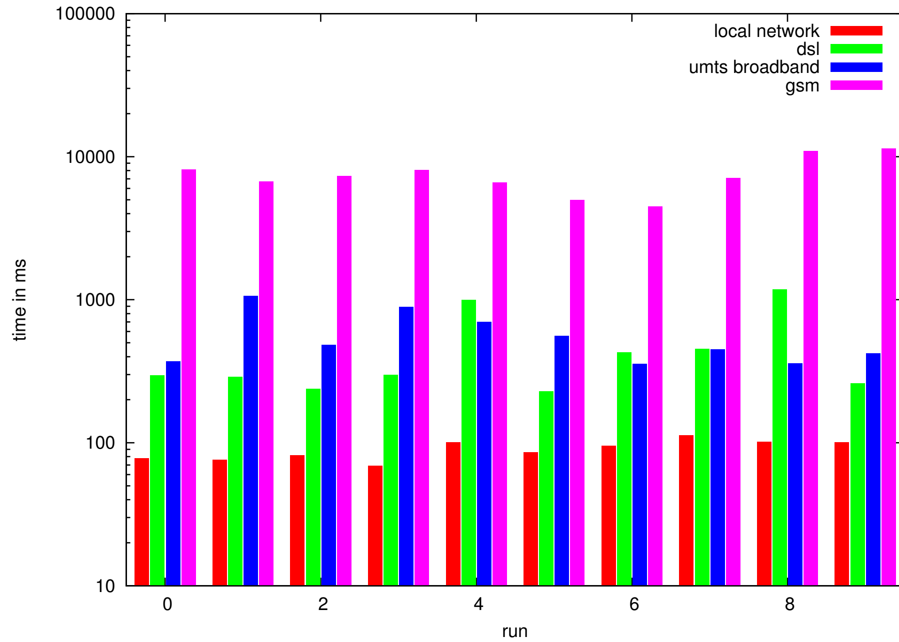


Figure 55: Roundtrip time of a message with a whole morph as content, this diagram uses a logarithmic scale on the y-axis

On the local network it takes 4.3 ms rountrip time in average. So that the message arrives approximately after 2.2 ms at the other client. This is as much time as it needs to send a ping message. On the DSL connection it takes 128.3 ms for one roundtrip in average. So that a mouse message will arrive after approximately 64.2 ms on the other clients. The UMTS-Broadband connection reaches 389.2 ms in average for a roundtrip. So that the message will approximately arrive 194.6 ms after sending it. On the GSM connection a roundtrip takes 12267.2 ms which means that the message will arrive 6133.6 ms after sending it at the destination. Again, as we used the system ourselves, we figure that a time around 200 ms is enough to create a reasonable smooth continuous action on remote clients.

Conclusively, connections using the local network, DSL or UMTS-Broadband are fast enough to use them with our system. But connections using GSM are definitly too slow.

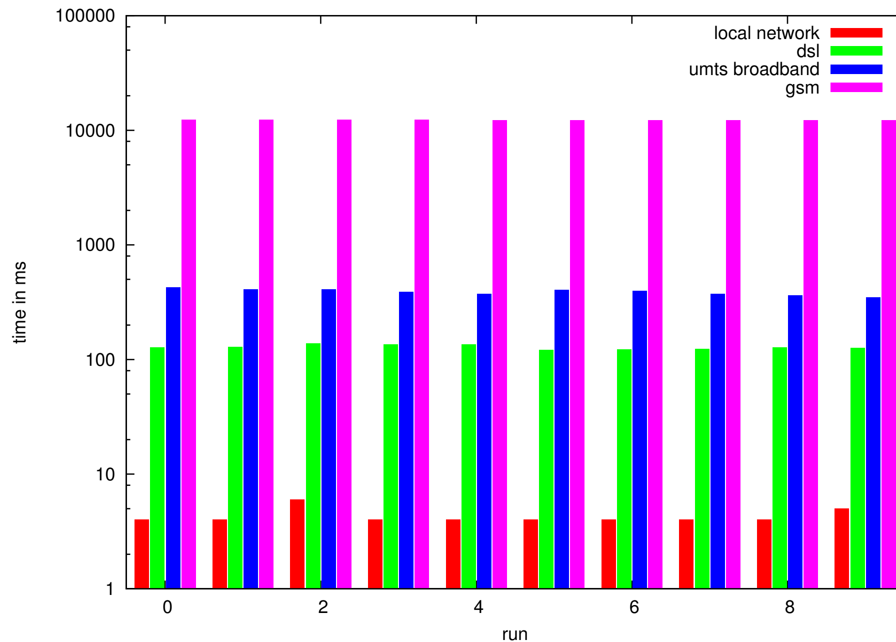


Figure 56: Roundtrip time of a mouse message, this diagram uses a logarithmic scale on the y-axis

Another question is if the system is usable and responsive enough to collaborate with people from all around the world. As an example we assume a work group that is in Potsdam[82] and Palo Alto. The distance between the two cities is about 9154 km[83]. The fastest connection between the two cities would be a roundtrip time around $2 \times 9154\ km/speed\,of\ light = 2 \times 9154\ km/299.792\ km/ms \approx 60\ ms$. The people working in Potsdam should not have a problem as they might use the local network. But the question is if the system is usable and by that fast enough for the people in Palo Alto.

The actual roundtrip time would not be as low as 60 ms since the information has to travel with the speed of light. Additionally, we can not assume the same network performance as in the local network. Consequently, we will assume that the network connection performs as well as a DSL connection. Adding the additional latency to the values of the DSL connection results in times of 116.2 ms[84] for a small message and 263.35 ms[85] for a larger message to arrive. Since this estimation is based on the speed of light and the direct connection between the two cities it is very optimistic. The latency in reality might be higher. Conclusively, the user experience might not be good enough that far away from the server.

---

[82] Where the server of lively-kernel.org is located.

[83] Queried at `http://www.wolframalpha.com/input/?i=distance+from+potsdam+to+palo+alto` (visited 29.06.2012)

[84] $172.4\ ms + 60\ ms/2$

[85] $466.7\ ms + 60\ ms/2$

4.6 Related Work

In this chapter we shortly point out some related work that has been done by other research groups. The work discussed here was done in the 1980s and 1990s.

WYSIWIS[86] In 1987 a group of researchers at XEROX Palo Alto Research Center worked on a collaboration system for meetings [55]. For that, they took a closer look at the WYSIWIS principle that in their opinion supports many features an analog system such as a whiteboard[87] has. But they also hold that WYSIWIS interpreted strictly is too inflexible. This work is an example of how to handle problems evoked by the strict interpretation of the principle. It gives as well ideas for a collaborative drawing system.

Cognoter The same group who did the research on the WYISIWIS principle developed an application named Cognoter that serves as a support system for collaborative organization of ideas [56]. They point out the benefits of working in a group. Further, they have a closer look on how to support the process of finding ideas[88], as well as organizing and evaluating them.

GROVE[89] In 1989 a group of researchers at the Microelectronics and Computer Technology Corporation in Austin, Texas worked on a collaborative outline editor named GROVE [20]. In their findings they describe the difference between shared and private workspaces. Along with that they developed synchronization boundaries for their system. The work also contains a discussion about WYSIWIS. Concurrency control in synchronous collaboration systems is also discussed by taking different approaches like locking of content and operational transformations[90] into account.

Portholes In 1992 researchers at Xerox PARC[91] developed a system to increase awareness of distributed work groups called Portholes [18]. The basic idea is that each collaborator has a camera and a microphone installed at his or her work place. The system shows the images of the whole group in one view. Consequently one can see who is actually working at a given time. With the microphones short audio messages could be recorded. The system they developed made it possible to be aware of ones collaborators without much need of information gathering. In their work they also describe the architecture of the system which consists of serveral servers for the different locations. Those servers are responsible for the image processing of their connected clients. In addition, the servers synchronize to the servers of the other locations so that a client only has to query one server to get the information being stored by the whole system.

---

[86] What You See Is What I See
[87] Their metaphor is a chalkboard.
[88] Brainstorming
[89] GRoup Outline Viewing Editor
[90] Optimistic approach to synchronization where an operation is executed immediately with the possiblity to undo it later.
[91] Locations in Cambridge, Great Britan and Palo Alto, CA, USA

ShrEdit In 1992 researchers at Xerox PARC in Cambridge did research on a collaborative text editor as well [17]. In their findings they compare different approaches made by other research teams. Moreover, their approach focuses on the concept of shared feedback. The main statement of the group is that giving a group enough information on what is happening on the shared workspace and enabling them to communicate informally is better than predefined roles[92].

TeamWorkStation In 1990 a group of researchers at NTT Human Interface Laboratories developed an interesting approach to integrate virtual and actual workspaces [50]. The basic concept is to make video overlays of different workspaces. This creates a high acceptence since each user can use the tools he or she is most comfortable with. Obviously a problem is to merge the work a group has done into one artifact. They integrated microphone and camera as well to make communication between collaborators as easy as they would sit next to each other.

Single Display Groupware In 1999 a group of researchers at the University of Maryland developed a collaboartion system that has its focus on local collaboration on one computer [57]. The challenge of this work was to design a system that offers individial input possibilities for each collaborator. At this time multitouch enabled devices where not present and the guiding input principle was one mouse and one keyboard on one machine. This work offers an insight on how collaboration support works on a level of most narrow cooperation.

4.7 Future Work

As this work is only a part of a project that took place at Hasso-Plattner-Institut in 2012 there are some ideas that not have been implemented. This chapter will suggest some of the work that might be done in order to improve our system.

Software Quality As the system served as a platform to experiment and try out different approaches, the software quality is not very high. In order to make the system ready for productive use, there should be some work done to increase the quality of the software. To achieve that, the used algorithms and architecture should be much more robust since it seems to fail unexpectedly when using it. Note that the amount of tests is rather small. To ensure that the written software does what we expect, there should be at least a unit test suite that tests the functionality of the system.

Furthermore, to ensure a good extensibility and understanding of the system there should be additional documentation. Especially the architecture and functionality should be documented well for creation of better understanding.

Client Features As seen in the chapters above, the client application has some features that support synchronous or asynchronous collaboration style. In order to create more possibilities of working styles and improve the collaborative work we have some features in mind that the client could support.

---

[92] which would mean additional management overhead for the collaborative work session

Chat System The chat is a way to communicate within the system. However, the text messages that are sent and displayed to all participants of a working session form a single stream. As there might be groups of people within the workgroup working focused on special features, a stream for the whole group might not work out well. Since different threads in a single stream are difficult to read, an additional information to the chat message can help to split the stream up into different topics. This additional information could be a position within the shared workspace. We imagine a chat system which we called ObjectTalk in which conversations happen around the object of interest. By doing so, everyone who is interested in this coversation can scroll the view to the object where the chat message are displayed as well.

Drawing The drawing of sketches works quite well in the current version of the system. But an opposite operation is missing in the system. There is a way to clear the whiteboard by deleting all contents, but when drawing users also want to erase single strokes that are wrong. For deletion, we imagine to different kinds of erasers: The first approach is an eraser that works just as a eraser on paper. Usability wise, it is just a special pencil that deletes content instead of creating it. The second approach is a tool to delete whole lines. The basic idea is that the user draws a line across all lines that he or she wants to delete. With that approach, much content could be deleted with one action by also being able to select what should be deleted.

Change Detection The detection of changes on properties[93] which are part of the Morphic Framework do not have to be discovered by diffing the objects over and over again. For that pupose the Morphic's getter and setter functions can be utilized. In addition, other than properties, the addition of scripts to an object can be detected via the *addScript* function of a morph. The need of diffing and merging of objects does not cancel out since the adding and changing of arbitrary properties of the object must be detected as well.

Having this utilization of Morphic's getter and setter functions, it is possible to keep a log of the changes that have been made. This log allows to implement an undo-redo feature for the changes on Morphic properties.

Conflict Handling Currently, the handling of conflicts is not actively done by the application. Our approach is that the group will coordinate itself to avoid such editing conflict. Since human communication is not free of misunderstandings, the system should somehow handle editing conflicts. A possible solution would be to accept the most current change that happened on an object. For example, if two users change the size of a morph in an overlapping way, the change that happened lastly will be applied. Doing so the optimistic synchronization approach can still be used. The client which made the change that is going to be accepted will discard the update it is receiving from the other client because it's change is more recent. The other client whose change is discarded will apply the change it is receiving to its synchronization state.

Connections The current implementation does not synchronize connections since the synchronization boundary does not allow to synchronize objects that were not dropped onto the synchronization pane. In order to have connections available, we could at least allow connections to objects that are already synchronized.

[93] e.g. color, size, position

Creating Awareness of Changes Another problem occures when one user is rejoining an existing session. The work that has been done in the time he or she was offline will be synchronized completely to his or her client. This hard synchronization of the state is lacking of awareness of what is changed. We imagine to create a timeslider with wich the user can replay the actions at his or her own speed. Additionally, diffing to the latest version of an object can create an understanding of what changed with the updates that the user just received.

Snapshot of Drawings Currently drawings could not be reused since they can not be accessed as usual morphs when the user interacts on the whiteboard. For recomposition puposes we thought of a tool that would allow the user to take photos of the drawings which are then pasted to a new morph that can be dragged around just like any other morph.

Alternative Clients The design allows to implement different clients that communicate over the same synchronization server. It would be interesting to implement clients in systems different from Lively Kernel. That could attract new users and by that create more freedom of the tools the users have to use.

Server Improvements The software of the synchronization server could improve as well. By now when the server crashes it has to be restared by hand. By executing the server in a loop that restarts the server as it crashes, this problem can be solved.

Further, we made sure that broadcasting is possible, but only on the level of channels. To be able to send messages to every client that is connected to the server for example for maintainance reasons there should be a way to broadcast messages to all connected clients.

In order to enable a better integration into other existing systems we could use a standard instant messaging protocol like XMPP.

Lastly, it could be interesting to implement the system without a synchronization server. Communication would have to happen in a peer-to-peer manner. A problem with that approach could be that it would possibly use more bandwidth since one client has to send its changes to multiple clients instead of just the server. Another problem could be how the clients get to know each other.

4.8 Conclusion

This work builds a basic implementation of synchronous collaboration support in Lively Kernel. We found solutions for fast sharing of content. This enables users to work together on synchronized objects. Further we illustrated how collaborative work can be supported on the levels of idea finding and implementation.

Nevertheless two main problems remain that need to be faced in order to create a productive collaboration system. First, the problem of overlapping actions [20] needs to be solved. The present system does not handle actions that happen at nearly the same time on the same object. As an example, one user might change the color of a morph. If his or her collaborator is changing the color of the same morph before the change arrives at his or her client both synchronization states will not be the same. This is because both clients receive the update of the other client. An approach to solve this was given in Future Work.

Secondly, the awareness for people (re-)joining a session needs to be improved. Currently there is no way to see what happend in the meantime[94]. We thought of a solution for this problem as well. An approach could be a timeslider that enables the user to scroll through the actions that happend.

The chapter Future Work showed that there are several ideas for features to improve the system. Finally, built on Lively Kernel the system is a suitable platform for collaboration research.

[94] The time the participant was offline

5 Benchmarking Lively Kernel on Various Systems

5.1 Introduction

Lively Kernel is a highly interactive prototyping and development platform. It is designed to enable intuitive and fluent development of both prototypes and rich applications. To enable fluent development you need fluent interaction. To have fluent interaction you need fluent action processing. A low performance can undermine the very essence of Lively Kernel. To prevent this from happening, we have to be able to get notified about, find and remove performance issues.

Although most performance issues show themselves sooner or later, most developers tend to overlook them, because they are focused on other tasks at the moment. In addition it is very difficult to tell, which issue is the most important. Furthermore, issues need to be found, before they strike the user. To enable this, we decided to use benchmarks. Benchmarks provide us with a numerical score that varies over platform and implementation.

There are already JavaScript benchmark suites, for example SunSpider, V8, Kraken and Dromaeo. However, these focus on overall JavaScript performance while we want a benchmark focused on Lively Kernel performance. In fact, we want it to change with our implementation, so we can see any improvement or deterioration of performance. So we need to implement our own, Lively Kernel specific benchmark suite.

When having found an issue, we want to fix it. To remove a performance issue, we need to find the bottle neck, that causes the issue. A common way to find bottle necks is profiling. Measuring the execution time of all called functions will unveil, which take the most time. Profiling is available in some browsers already. In fact, we used the profiler of Google Chrome successfully in the past. However, there are some features that can only be achieved by a Lively Kernel specific solution. Google Chrome knows nothing about the internal architecture of Lively Kernel, but we do. So we can provide additional information like object or class names. Furthermore we can use this knowledge to exclude irrelevant functions. These are for example temporary functions created to serve as callbacks in functions like `setTimeout`, `forEach`, map or `reduce`. Additionally to the mere functionality, we will present a decent user interface.

Section 5.1 contains relevant details about Lively Kernel. Section 5.2 lists the features we included in our benchmark suite and motivates the decision. Section 5.2 describes the general approach we followed in our benchmark implementation. Sections 5.2 to 5.2 detail the implementation of the benchmarks representing the individual features. Section 5.3 shows the implementation of the profiler and its user interface. Finally, section 5.4 Presents the results of the benchmarks and insights from the profiler. Section 5.6 compares our benchmark to other JavaScript benchmarks. Section 5.5 shows our plans for improvements and section 5.7 draws a conclusion.

Lively Kernel This Section shortly describes Lively Kernel and gives some details on the most important features.

Lively Kernel is a self-sufficient developement plattform, that runs directly in the browser. It enables you to create and run prototypes and even rich applications right in the browser. These applications do not need to be deployed seperately, but run directly in Lively Kernel, even while you edit them.

The fundament of Lively Kernel is the morphic system. Morphic is a user interface framework originally invented for squeak. It focuses on direct manipulation of graphical objects. These Objects are called morphs. Each part of the user interface is a morph. A window is a morph, a box is a morph, a text is a morph, a button is a morph. A morph may be composed of other morphs, these are called its submorphs. The sumbmorph relation forms a tree. The root of the tree of displayed morphs is the world. Morphs can react on user behaviour and/or perform actions at regular intervalls. [45]

As an extension to the morphic System, Lively uses its web-based nature to provide an online repository of morphs. This repository is called the PartsBin and the morphs published there are called parts. The PartsBin includes virtually everything, from simple shapes to UI widgets, system tools and even whole applications. As Lively is completely self-sufficient, even the PartsBin is a part in the PartsBin. [44]

The PartsBin enables asynchronous cooperation in a basic way. When programmers make different changes on the same object, Lively kernel can find the differences of the versions and aid in merging them. (see section 3) Additionaly, Lively is beginning to support live collaboration in form of a part, that synchronizes all its submorphs and provides a simple whiteboard drawing interface. (see section 4)

Building on these collaboration features, Lively Kernel provides a wiki like environment called Webwerksatt. In the Webwerkstatt, users can create their own pages, create prototypes or applications and save them. Then they can send a link to all their friends, so they can try it and give feedback. These friends could also jump right into developement and improve the application right where they tried it.

All in all, Lively Kernel is a vision of people from all over the world being able to create working prototypes and applications easy, fast and together.

5.2 Benchmarking

Benchmark Categories To Calculate a meaningful score, we need to find a representative set of functionality to test. This section describes what components of the system are part of the benchmark suite and why we included them. The implementation is given in the following subsections.

The most prominent and most important part of Lively Kernel is its implementation of morphic. Morphic is a system that allows direct user manipulation of graphical objects. Each morph can be edited directly. Even the Tools in Lively Kernel are morphs. When working with Lively Kernel, hundreds of morphs are created and destroyed each minute. Therefore we want to include the creation and destruction of morphs in our benchmark.

When working on projects in Lively Kernel, you work on morphs. The usual workflow includes a lot of morph editing. (see section 1) That is why we want to measure how long it takes to resize or rotate a morph, move it around, change its color, change its border or add scripts.

Another characterizing feature of Lively Kernel is its ability, to save and load Morphs. This ranges from single rectangles to whole worlds. While worlds are represented as XHTML pages, other morphs are "published" and can be loaded into any world from the PartsBin. The normal start of a workflow in Lively Kernel is loading a part, the normal end is publishing a new one or a new version. (see section 1) Morphs are stored as JSON files on the server, which requires serialization and deserialization. Therefore these two have to be included in our benchmark suite, too.

Lively provides a very easy way to use observer patterns, called connections. A connection has a source and a target object and property. It is fired, whenever the source property changes. It can directly write on the target property or call a function, using the changed value as a parameter. Connections can be applied to each and every property. One does not even need to write code to establish these connections, it can be done via a menu entry of the morph menu. This feature is heavily used in Lively applications, so we want it to be part of our benchmark suite.

Every text field in Lively Kernel, that can be edited, has the ability to evaluate the written code, often in a custom context. A prominent example is the Workspace that is used for nothing else but running experimental snippets or sharing snippets to execute before using an application. Another one is the ObjectInspector, that can show the properties of any JavaScript object. It has a text field to enter code, that can be executed via a keyboard shortcut. When evaluating this code, the variable `this` Even adding Scripts or changing classes involves evaluating the new code. Evaluation is therefore so present in Lively Kernel, that it cannot be left out of the benchmark suite.

Benchmark Framework All our benchmarks use the same base mechanism, that is explained in this subsection.

Most operations in Lively Kernel are quite simple ones. Moving a Morph, changing a color or displaying a script does not need a lot of computation time. The load caused by Lively originates in the number of these operations. An average world contains 200 to 300 Morphs, that have to be created and rendered on load. Whenever the Mouse is moved or a Finger swipes over the Touchscreen, hundreds of events are fired and need to be processed. This type of load needs to be reflected by our Benchmarks. Therefore we want a possibility to execute relatively small operations several times in a row and measure, how fast the browser completes this task. This is called a micro benchmark.

Micro benchmark come with another ability. We can minimize variation of the results by running the operation more times. We can even run for a certain time and count the number of successful runs instead of completing a certain task and taking the time.

We chose a path, that combines both ways of Benchmarking. Each benchmark has a repeatable operation, that can be executed several times in a row. This operation is executed a minimum number of times and for a minimum amount of time. This way a fast browser can run it a lot of times, while a slow browser is not stuck in the Benchmark for hours.

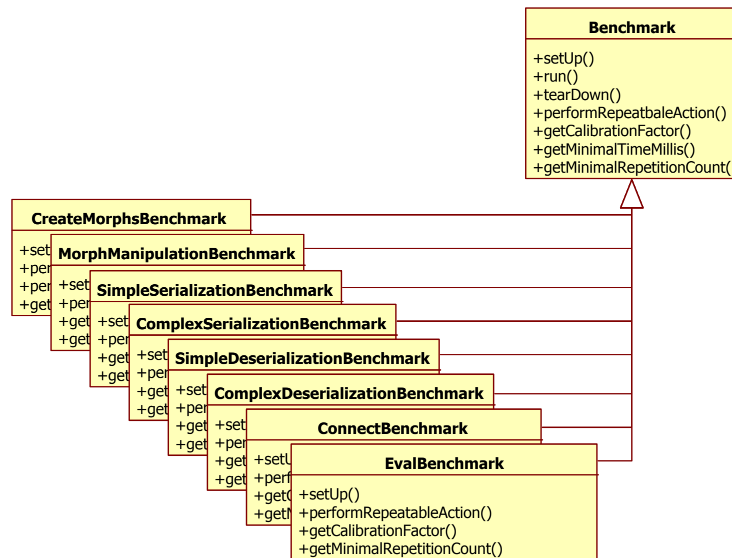


Figure 57: Benchmark Classes

This behavior is realized by a number of classes. The class `Benchmark` specifies the interface and implements the running framework described above. Each Benchmark overrides `performRepeatableAction`, `setUp` and `tearDown` as needed. The subclasses also provide the minimum number of runs and the calibration factor themselves, whereas the minimum running time is always set to 10 seconds, because some browsers tend to stop, if a script runs for more time. (See section 5.4)

```
run: function(){
    var before = new Date();
    var minTime = before.valueOf() + this.getMinimalTimeMillis();
    var minCount = this.getMinimalRepetitionCount();
    var c = 0;
    do {
        this.performRepeatableAction();
        c++;
    } while ((c < minCount || new Date().valueOf() < minTime));
    var after = new Date();
    var delta = after.valueOf() - before.valueOf();
    this.result = this.getCalibrationFactor() * c / delta;
}
```

Code Example 24: running a benchmark

The run Method can be seen in code example 24. Before the real benchmark Operation starts, we remember target time and the minimal number of runs. Then we continue executing the repeatable action specified by the subclass, till both requirements are met. Meanwhile, we count the number of runs. Finally we calculate the time passed since the beginning and calculate the resulting score.

The Score is calculated from the number of successful runs, the time needed and a calibration factor. This factor is used to normalize the results of the Benchmarks, so they can be compared among each other. Instead of a time you have to evaluate yourself you get a score. Instead of serialization taking far longer then morph creation (which is normal), you now see both having a similar score. The exact formula used for calculation of the score is $calibrationfactor \times \frac{numberofruns}{timepassed}$. The resulting score is better, the higher it is.

To make the calculated Score meaningful, we need to find meaningful calibration factors. We decided to use a reference system that should receive 100 Points on each benchmark. This reference System should satisfy some criteria:

1. Used with Lively Kernel
2. Common Software Configuration
3. Common Hardware Configuration

Lively Kernel users use a variety of Systems, one can nevertheless find some tendencies:

1. Windows and Mac dominate
2. Hardware of Windows machines varies largely
3. Hardware of Macs is mostly limited to iMacs and MacBooks
4. most users use the Google Chrome web browser

As our goal was a widely known and used configuration, we decided to use Google Chrome on an iMac 11. The full System specification of the iMac can be found in the appendix. Google Chrome was at version 19.0.1084.56. The calibration was performed on 2012-06-13.

The following sections show the implementation details of each benchmark, that is part of our suite, individually.

Benchmarking Morph Creation The morphic system is the base of the whole Lively Kernel user interface. A morph is a graphical object, that can be manipulated directly on screen. You can for example move, rotate or resize it. Everything visible in Lively Kernel is a morph. opening a tool means creating dozens of morphs and adding them to the world. It is essential that this operation is fast, otherwise Lively Kernel is of no use. Therefore, it is included in our Benchmark.

```
setUp: function($super, world){
    $super(world);
    this.morphClasses = [
        lively.morphic.Box,
        lively.morphic.Text,
        lively.morphic.Button
    ];
    this.bounds = rect(pt(0,0), pt(50,50))
}
```

Code Example 25: setting up the morph creation benchmark

```
performRepeatableAction: function(){
    var self = this;
    this.morphClasses.forEach(function(klass){
        var morph = new klass(self.bounds);
        morph.setExtent(pt(100,100));
        self.world.addMorph(morph);
        morph.moveBy(pt(50,50));
        morph.setExtent(pt(20,20));
        morph.remove();
    });
}
```

Code Example 26: running the morph creation benchmark

To get a representative value, we decided to use the most common morphs of the system, a box, a text and a button. These classes are stored in an array when the benchmark is set up. Additionally a rectangle providing the bounds is stored, because we do not want to measure rectangle instantiation.

The measured action itself performs typical steps, that are executed, when a morph is created. The morph is instantiated, resized and moved. To ensure the world is in the same state after the action, we finally remove the morph.

Resizing and moving the morph also belong to morph manipulation, so they are taken into account twice. This is on purpose, as these are used far more often than other manipulations like changing the fill or rotating the morph.

Benchmarking Morph Manipulation The Morphic System is the base of the whole Lively-Kernel user Interface. A Morph is a graphical Object, that can be manipulated directly on Screen. Everything visible in Lively Kernel is a Morph. Whenever anything in the user interface changes, morphs are manipulated in several ways. It is of high importance, that this happens very fast. That is why it is part of our benchmark.

To manipulate morphs, we need morphs and actions. We use the same basic morphs for this task as for the other benchmarks, a box, a text and a button. To ensure the morphs can be reset after each execution of the repeatable action, we define a base style and apply it to each morph. After completing this step, we add the morphs to the world.

```
setUp: function($super, world){
        $super(world);
        var bounds = rect(pt(0,0), pt(50,50));
        this.morphs = [
            new lively.morphic.Box(bounds),
            new lively.morphic.Text(bounds),
            new lively.morphic.Button(bounds)
        ];
        this.baseStyle = {
            position: pt(0,0), extent: pt(50,50),
            scale: 1, origin: pt(0,0),
            fill: Color.blue,
            borderWidth: 1, borderColor: Color.black,
            rotation: 0, borderRadius: 0
        }
        var self = this;
        this.morphs.forEach(function(morph){
            morph.applyStyle(self.baseStyle);
            self.world.addMorph(morph);
        });
        var proto = lively.morphic.Morph.prototype;
        this.actions = [
            //... see Code Example 1.5
        ]
    }
```

Code Example 27: setting up the morph manipulation benchmark

The actions used in this benchmark should include all important ways of morph manipulation. The actions we identified are: changing size and position by different methods, changing the origin of a morph, changing fill and border color, changing the border width and radius and adding scripts. We specified each of this actions as an object that includes the function to call and the arguments, that are to be passed to the function.

Having created the morphs and specified the actions, running the benchmark is only iterating over both and executing the actions. We use the method `apply` of the **`Function`** object to bind the function to the morph, pass the arguments and run it. After all actions have been executed, we reset the morphs to their base style.

Benchmarking Morph Serialization Lively Kernel is able to transmit whole morphs from one machine to another in several ways. One way is saving a world containing that morph and opening it on another machine. Another way, that allows transmission between worlds, is publishing it to the PartsBin and loading it from there to any world. While this is asynchronous, a synchronous way of sharing morphs has recently been introduced by the SyncMorph. The SyncMorphs allow it to share a part of your screen with coworkers. (see section 4)

```
1          var proto = lively.morphic.Morph.prototype;
2          this.actions = [
3              {func: proto.setExtent,
4                  args: [pt(100,100)]},
5              {func: proto.moveBy,
6                  args: [pt(20,20)]},
7              {func: proto.setBounds,
8                  args: [rect(pt(70,70), pt(150,150))]},
9              {func: proto.setScale,
10                 args: [2]},
11             {func: proto.setOrigin,
12                 args: [pt(40,40)]},
13             {func: proto.setFill,
14                 args: [Color.green]},
15             {func: proto.setBorderWidth,
16                 args: [2]},
17             {func: proto.setBorderColor,
18                 args: [Color.red]},
19             {func: proto.setRotation,
20                 args: [1.07]},
21             {func: proto.setBorderRadius,
22                 args: [5]},
23             {func: proto.addScript,
24                 args: [function testScript(x, y){
25                     this.moveBy(pt(x-y, y-x));
26                 }
27             ]}
28         ]
```

Code Example 28: The manipulation actions performed on the morphs

```
1 performRepeatableAction: function(world){
2     var self = this;
3     this.morphs.forEach(function(morph){
4         self.actions.forEach(function(action){
5             action.func.apply(morph, action.args);
6         });
7         morph.applyStyle(self.baseStyle);
8     });
9 }
```

Code Example 29: running the morph manipulation benchmark

All these ways of sharing need to transform the morphs in a transmittable form of data. In our case, this is a JSON string. It is therefore of high importance, that this serialization can be performed quickly. So we included it in our benchmark suite.

While serialization itself can be a complex and expensive task, benchmarking it is quite simple. We just need to take some morphs and call the serialization method. However, as serialization is quite expensive, getting a good sample of data with morphs of different complexity might consume a lot of time. Web browsers tend to stop script execution, if a script takes too long to complete a task, therefore we split the serialization benchmark in two parts: simple morphs and complex morphs.

The simple morphs are the same as in the other benchmarks, a box, a button and a text.

```
1  setUp: function($super, world){
2    $super(world);
3    this.morphs = [];
4    var bounds = rect(pt(0,0),pt(100,100));
5    this.morphs.push(new lively.morphic.Box(bounds));
6    this.morphs.push(new lively.morphic.Button(bounds));
7    this.morphs.push(new lively.morphic.Text(bounds));
8  }
```

Code Example 30: setting up the simple serialization benchmark

As complex morphs, we selected three commonly used parts from the tools category of the PartsBin. The ObjectEditor is the tool that can add scripts to morphs. The ObjectInspecor gives detailed information about JavaScript objects and is a valuable debugging tool. The StyleEditor is used to change the visual properties of morphs. We load them from the PartsBin using the method `loadPartItem` of the world.

```
1  setUp: function($super, world){
2    $super(world);
3    this.morphs = [];
4    this.morphs.push(
5      this.world.loadPartItem("ObjectEditor","PartsBin/Tools")
6    );
7    this.morphs.push(
8      this.world.loadPartItem("ObjectInspector","PartsBin/Tools")
9    );
10   this.morphs.push(
11     this.world.loadPartItem("StyleEditor","PartsBin/Tools")
12   );
13 }
```

Code Example 31: setting up the complex serialization benchmark

As said before, the execution of the benchmark consists only of calling the serialization method for each morph.

```
performRepeatableAction: function(world){
    this.morphs.forEach(function(morph){
        lively.persistence.Serializer.serialize(morph);
    });
}
```

Code Example 32: running the serialization benchmark

Benchmarking Morph Deserialization Transmitting morphs from one computer to another. It is used for the PartsBin, the SyncMorph and the saving and loading of worlds. To send a morph, it needs to be serialized. So the receiver needs to recreate the morph from the JSON string. This deserialization takes place, when one loads a world, opens a tool or loads a part to work on. It is crucial, that this does not block the user from using Lively Kernel for long. Therefore we included it in our benchmark.

Similar to serialization, deserialization is easy to benchmark, but it takes a lot of time. As we do not want the browser to interfere with our script because it is running too long, we divided our benchmark into two smaller benchmarks: deserialization of simple morphs and complex morphs.

The simple morphs are the same as in the other benchmarks, a box, a button and a text. We serialize them and keep the JSON strings to deserialize them when running the benchmark.

```
setUp: function($super, world){
  $super(world);
  this.jsons = [];

  var bounds = rect(pt(0,0),pt(100,100));

  var box = new lively.morphic.Box(bounds);
  box = lively.persistence.Serializer.serialize(box);
  this.jsons.push(box);

  var button = new lively.morphic.Button(bounds);
  button = lively.persistence.Serializer.serialize(button);
  this.jsons.push(button);

  var text = new lively.morphic.Text(bounds);
  text = lively.persistence.Serializer.serialize(text);
  this.jsons.push(text);
}
```

Code Example 33: setting up the simple deserialization benchmark

As complex morphs, we selected three commonly used parts from the tools category of the PartsBin. Two of them are identical to the ones in the serialization benchmark. The ObjectEditor and the ObjectInspector. The StyleEditor tends to throw errors on mobile devices. This may interfere with the benchmark, so we use the PartTestRunner instead. It is used for the execution of automated tests on parts from the PartsBin. Again, we only keep the serialized versions.

```
1  setUp: function($super, world){
2    $super(world);
3    this.jsons = [];
4
5    var objectEditor =
6      this.world.loadPartItem("ObjectEditor", "PartsBin/Tools");
7    objectEditor =
8      lively.persistence.Serializer.serialize(objectEditor);
9    this.jsons.push(objectEditor);
10
11   var objectInspector =
12     this.world.loadPartItem("ObjectInspector", "PartsBin/Tools");
13   objectInspector =
14     lively.persistence.Serializer.serialize(objectInspector);
15   this.jsons.push(objectInspector);
16
17   var partTestRunner =
18     this.world.loadPartItem("PertTestRunner", "PartsBin/Tools");
19   partTestRunner =
20     lively.persistence.Serializer.serialize(partTestRunner);
21   this.jsons.push(partTestRunner);
22 }
```

Code Example 34: setting up the complex deserialization benchmark

The execution is also similar to the serialization benchmark. We only need to call the deserialization function for each JSON string.

```
1  performRepeatableAction: function(world){
2      this.jsons.forEach(function(json){
3          lively.persistence.Serializer.deserialize(json);
4      });
5  }
```

Code Example 35: running the deserialization benchmark

Benchmarking Connections Lively Kernel allows the implementation of observer patterns in an easy to use way called connections. A connection has a source and target object and property. If you connect an attribute with another attribute, each change on the first one is applied to the second one. If you connect an attribute to a method, the method is called on each change, providing the new value as the only argument. Connections are widely used in Lively Kernel, mostly to keep the user interface in sync with internal variables. There is even a possibility to "fire" a connection without changing the attribute, that is e.g. used to fire buttons. It is spread so far through the system, that it cannot be ignored by the benchmark.

Although every property of every object can be connected, we decided to use only morphs, because connections between morphs are more common. Our Example set consists of a box, a text, and a button. The box has a script, so we can include all kinds of connections.

```
1 setUp: function($super, world){
2     $super(world);
3     var bounds = rect(pt(0,0), pt(50,50));
4     this.box = new lively.morphic.Box(bounds);
5     this.text = new lively.morphic.Text(bounds);
6     this.button = new lively.morphic.Button(bounds);
7     this.box.addScript(function onFire() {});
8 }
```

Code Example 36: setting up the connection benchmark

There are for connections we want to benchmark. The first one connects a custom fired signal to a script. The second one connects an attribute to a method, providing the new value as parameter. The third one connects an attribute to another attribute. The forth one again connects an attribute to a method. This type is included twice, because it is more common. We use `connect()` to set the connections. It is important to use `disconnect` here, as we want everything to be in the same state before and after the repeatable action.

Benchmarking Eval Lively Kernel is a dynamic and self-sufficient system. That means especially that one can edit every part of the system out from inside the system. This is possible, because javascript is a scripting language, that can execute code provided as a string at runtime. This is not only used to load the lively Kernel, but also when adding scripts to morphs and when modifying modules. The user can also execute code snippets in text fields. So it is very important to us, that this can be done fast and should therefore be included in our benchmarks.

We chose two snippets to execute, that shall represent eval actions without contatining expensive operations. The first one executes some basic arithemtic operations and checks the result for validity. The second one defines a function and executes it to check, if it was created correctly. Allthough we do not expect any modern browser to produce wrong results, we included this validity checks to be sure everything went right.

```
performRepeatableAction: function(){
    connect(this.button, "fire", this.box, "onFire");
    connect(this.box, "scale", this.text, "setScale");
    connect(this.button, "rotation", this.text, "textString");
    connect(this.text, "position", this.button, "setLabel");
    disconnect(this.button, "fire", this.box, "onFire");
    disconnect(this.box, "scale", this.text, "setScale");
    disconnect(this.button, "rotation", this.text, "textString");
    disconnect(this.text, "position", this.button, "setLabel");
}
```

Code Example 37: running the connection benchmark

```
setUp: function($super, world){
    $super(world);
    this.snippets = [
        "var t = 5;\n"+
        "var t2 = 5*t;\n"+
        "var t3 = 7*t2 + 2*t;\n"+
        "t++; t2++; t3++;\n"+
        "if(t3 !== 361){\n"+
        "    console.error(\"FAIL!\");\n"+
        "}",

        "var solveSqr = function(a, b, c){\n"+
        "    var p2 = b/a/2;\n"+
        "    var q = c/a;\n"+
        "    var det = p2*p2 - q;\n"+
        "    if(det < 0){\n"+
        "        return [];\n"+
        "    }\n"+
        "    var sqrt = Math.sqrt(det);\n"+
        "    return [-p2+sqrt, -p2-sqrt];\n"+
        "}\n"+
        "var res = solveSqr(2,2,-12)\n"+
        "if( res[0] !== 2 || res[1] != -3){\n"+
        "    console.error(\"FAIL!\");\n"+
        "}"
    ];
}
```

Code Example 38: settng up the eval benchmark

Running this benchmark is quite simple, as it only involves calling the native function `eval()` for each snippet.

```
performRepeatableAction: function(){
    this.snippets.forEach(function(ea){
        eval(ea);
    });
}
```

Code Example 39: running the evaluation benchmark

Lively Score on different Systems We have run the described benchmark on different platforms using different browsers. The following subsections shows the results and draws conclusions.

Figure 58 shows the benchmark results of the iMac 11 using different browsers. The browser versions are:

- Google Chrome 19.0.1084.56
- Firefox 13.0.1
- Safari 5.1.7

We had to increase the configuration value `dom.max_script_run_time` for Firefox, as it shows a dialog to stop the script after 10 seconds, that pauses script execution. Google Chrome does similarly, but its dialog does not pause the script. Safari did not show a dialog at all during our tests.

As we used Google Chrome on this machine as the reference system, results with Google Chrome are at a constant score of 100. The overall result of Safari is 135. That means Safari performs better than Google Chrome. To be exact: Safari took $1/1.35 = 0.74$ as long to execute an action as Google Chrome on average. Firefox got a result of 98, which means it performs about as good as Google Chrome.
The detailed results of the different benchmarks provide even more interesting insights. Apparently, Google Chrome is especially good in evaluating scripts(about twice as fast as Safari and six times as fast as Firefox), while Safari and Firefox are better in Serialization (Firefox is about twice as fast as Google Chrome, Safari even 2.3 times).

The results of the same benchmarks run on a comparable windows machine can be seen in figure 59. They are very similar to the iMac. Safari performs a bit better on the iMac while Firefox profits from Windows, this is probably caused by the primary target platforms for these browsers. Google Chrome performs almost equally on both machines, with one exception: the evaluation of scripts under windows is twice as fast as under Mac. We are not sure where this huge difference comes from, but suspect either a platform specific optimization trick or the all-in-one binary of the Mac version to be the source.

As Lively Kernel is being adjusted to provide a decent mobile interface, the performance on mobile devices is especially interesting. We were able to get our hands on some mobile devices to run the benchmarks on. An iPad2 with iOS 5.1, an iPhone 4S with iOS 5.1, and an HTC Desire S with Android.

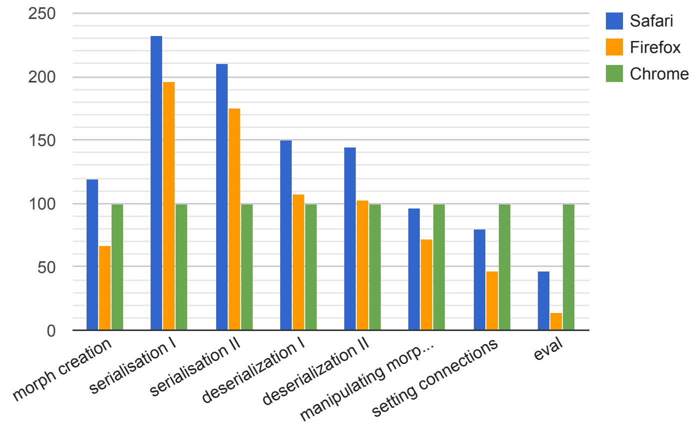


Figure 58: Benchmark results on the IMac 11, different browsers. Google Chrome sets the reference value of 100. Higher is better.

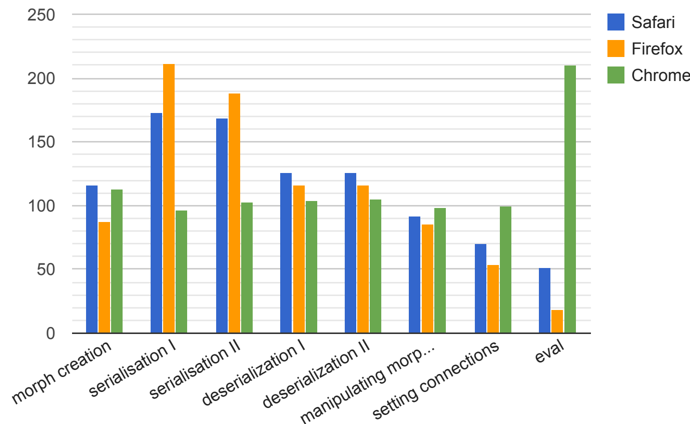


Figure 59: Benchmark results on a Windows machine, different browsers. Higher is better.

As expected, the iPad2 performs best, it has a total score of 18, which means it needs about 5 times as long to perform an action as the iMac with Google Chrome. As the browser is a Safari, the individual results match those from the iMac version of Safari. They are high for serialization and deserialization and low for connections and eval. The iPhone is a bit slower but has a very similar distribution. That was expected, as they vary mostly in hardware and only slightly in software.

The Android phone performs about 0.57 times as good as the iPhone which is about 0.44 times as good as the iPad. This was also quite expected, as the HTC Desire S is an older phone than the iPhone4S and was sold in a lower price category. The score distribution varies from the one of the apple devices. Again, the Google Software performs better in eval and worse in serialization.

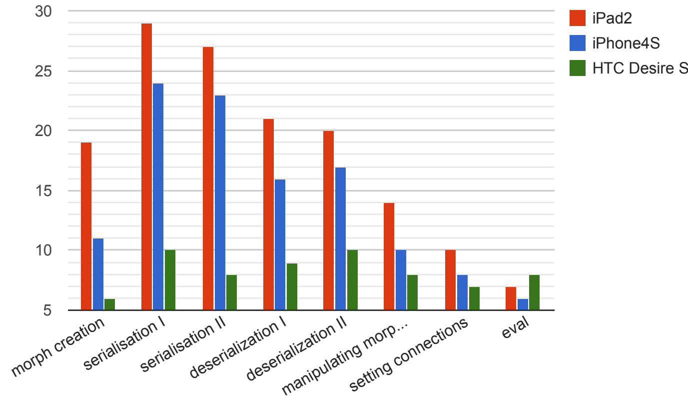


Figure 60: Benchmark results on different platforms, using Google Chrome where possible, Safari on the iPad2. Google Chrome on the iMac sets the reference value of 100. Higher is better.

The benchmarks confirm, what we felt when using Lively Kernel on different devices ourselves. It is usable on the iPad, but has yet to be more optimized. Smartphones are not fast enough to support the current system. So an overall score between 12 and 17 seems to be the minimal score indicating a suitable system.

5.3 Profiling

This section describes, how we measure the execution time of methods in order to find bottle necks.

Now that we are able to find out, if there are performance issues and measure increases ore decreases of performance, we need a way to track down the issues. Virtually every performance issue is caused by a single bottle neck, a system component that can not process the data as fast as it is provided. To find these bottlenecks, it is very helpful to monitor the execution time of methods, to profile function calls.

Profiling single functions To measure the execution time of the individual functions, we need to track when we enter and leave them. Having this information we can calculate the time spent in each function. There are several methods to gather this information.

The first and probably most efficient one would be to utilize the JavaScript VM. In fact most modern browsers provide a profiler with access to the VM either native or as a plug-in. However, we want a cross-browser solution, that is optimized on the Lively Kernel architecture. As we have no access to the VM itself, we need an other option.

Fortunately, JavaScript is very dynamic and self-reflective. Each Method and each function is an object, that can be replaced. So we can replace it with a new function, that performs additional tracking steps before and after the execution of the original function.

From here on, we will use the following bottles of beer function as an example to illustrate the explanations. The function `alertOK(text)` is a lively function, that shows `text` in a green box.

```
bottles = function(n) {
    if(n>0) {
        alertOK(
            n + " bottles of beer on the wall, " +
            n + " bottles of beer.\n" +
            "Take one down and pass it around. " +
            (n-1) + " bottles of beer on the wall."
        );
        bottles(n-1);
    } else {
        alertOK(
            "No more bottles of beer on the wall," +
            "no more bottles of beer.\n" +
            "Go to the shop and buy some more," +
            "99 bottles of beer on the wall."
        );
    }
}
```

Code Example 40: bottles of beer - example used for illustration purposes

The first idea to implement this approach would be just remembering the time of entrance and then calculating the time spent in the function after the execution has finished, adding up these results to retrieve the overall time spent in the function.

The method `apply(context,arguments)` is JavaScript native and executes a function. The variable this inside the function is bound to `context`.The contents of the array `arguments` are passed as arguments to the function. As we want to preserve the context, we pass the context of the new function to the old one. The variable `arguments` is a magic variable in JavaScript. It contains all arguments passed to the current function call, no matter how many arguments are expected in the definition. As we want to preserve them, we pass them to the original function unchanged. We can put the original function back in place by calling setting `window[functionName]` to `window[functionName].orig;`.

This approach gives correct results, as long as no recursion occurs. Have a look at figure 61. It shows a possible call tree of the beer function. The data provided is not actual meassured data, it is just an example. The numbers at the edges of the tree denote the total execution time of the called function. The numbers next to the nodes denote the execution time of the functions without the time taken by subroutines.

```
var functionNames = ["bottles", "alertOK"];
functionNames.forEach(function(functionName){

  var newFunction = function () {
    var before = new Date().valueOf();
    var out = newFunction.orig.apply(this, arguments);
    newFunction.totalTime += new Date().valueOf() - before;
    return out;
  }

  newFunction.totalTime = 0;
  newFunction.orig = window[functionName];
  window[functionName] = newFunction;
}
```

Code Example 41: naive profiling approach

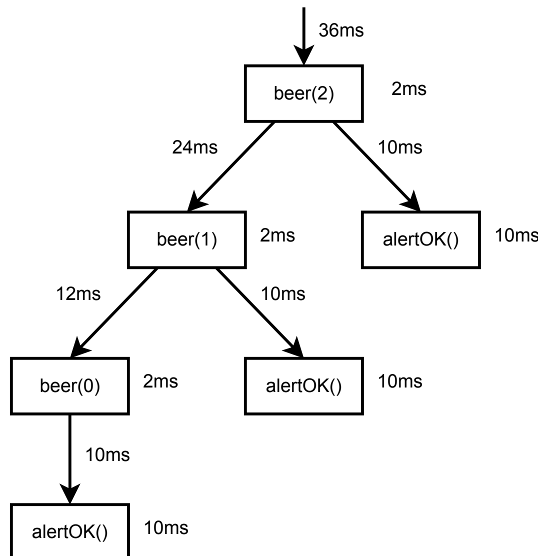


Figure 61: Call tree of an example run of the beer function (no actual data). Numbers on edges denote the running time of the called function. Numbers next to nodes denote the running time of the function without subroutines.

The naive approach described above would now add up all execution times of the beer function. That means it tells beer ran for $36\ ms + 24\ ms + 12\ ms = 72\ ms$, which is 200% of the total running time. This is obviously not the value, we wanted to see. That is the reason we need to remember, if a function has been called in the current call stack already. Only the execution time of the first call per stack should be added to the overall sum.

```
var functionNames = ["bottles", "alertOK"];
functionNames.forEach(function(functionName){

  newFunction = function () {
    var before = new Date().valueOf();
    var newFunction.recursionDepth++;
    var out = newFunction.orig.apply(this, arguments);
    newFunction.recursionDepth--;
    if (recursionDepth === 0) {
      newFunction.totalTime += new Date().valueOf() - before;
    }
    return out;
  }

  newFunction.totalTime = 0;
  newFunction.recursionDepth = 0;
  newFunction.orig = window[functionName];
  window[functionName] = newFunction;
}
```

Code Example 42: profiling approach that takes recursion into account

This approach counts 36 ms for beer and 30 ms for `alertOK`. These are the correct total execution times of the functions. However, it is not very helpful, when looking for bottlenecks. The function `beer` has a higher execution time than `alertOK`, nevertheless, most time is spent executing `alertOK`. This is because `beer` includes `alertOK`, but does not perform expensive operations itself. The time spend in `beer` without `alertOK` is only 6 ms. According to the Google Chrome profiler, we call this time the self time.

To measure the self time, wee need to keep track of the call stack. Normally this feature is provided by JavaScript, but different browsers vary in their interpretation of the standards. So this feature is not reliable and we need to implement our own tracking.

Fortunately, we already replaced all the functions, that are interesting to us, so we can build a stack of these functions and thereby have a reference to the calling function. So we can subtract the time the called function needed from the self time of the calling function. As we subtract recursive calls, too, self time measurement does not need to handle recursion different than other calls.

Instrumenting whole objects We now have achieved the ability to measure the execution time of single functions, but this is not all we want. When looking for bottle necks, the programmer seldom knows a small set of functions, that contain it. So we need to be able to instrument all methods of multiple objects to find the functions taking the most time. There is no need to build an extra method to instrument classes, because they are only an abstraction of the prototypical inheritance of JavaScript. This results in every class holding a reference to a prototype object for all its instances that can be instrumented in the same way as normal objects.

```
var callStack = [];
var functionNames = ["bottles", "alertOK"];
functionNames.forEach(function(functionName){

  var newFunction = function () {
    var before = new Date().valueOf();
    newFunction.recursionDepth++;
    callStack.push(newFunction);
    var out = newFunction.orig.apply(this, arguments);
    callStack.pop();
    newFunction.recursionDepth--;
    var delta = new Date().valueOf() - before;
    if (recursionDepth === 0) {
      newFunction.totalTime += delta;
    }
    newFunction.selfTime += delta;
    if (callStack.last()){
      callStack.last().selfTimes -= delta;
    }
    return out;
  }

  newFunction.totalTime = 0;
  newFunction.selfTime = 0;
  newFunction.recursionDepth = 0;
  newFunction.orig = window[functionName];
  window[functionName] = newFunction;

}
```

Code Example 43: profiling approach with self time

Instrumenting whole objects is quite simple. First we use the Lively Kernel method `Functions.own` to get a list of the names of all functions defined on this object itself. The list does not contain functions inherited from prototypes, as we want to instrument these via instrumenting the prototype. The only other change is replacing the global namespace `window` with our object.

Profiling the whole system Now that we can instrument whole objects, we want to be able to instrument all classes of the system at once. As said before, instrumenting a class can be achieved by instrumenting the prototype-object in the same way as any other object thanks to JavaScript prototypical inheritance. Getting a list of all classes is also quite easy, we can get a list by calling `Global.classes(true)`.

```
1  var callStack = [];
2  var functionNames = Functions.own(obj);
3  functionNames.forEach(function(functionName){
4
5    newFunction = function () {
6      //as before ...
7    }
8
9    newFunction.totalTime = 0;
10   newFunction.selfTime = 0;
11   newFunction.recursionDepth = 0;
12   newFunction.orig = obj[functionName];
13   obj[functionName] = newFunction;
14
15 }
```

Code Example 44: instrumenting a whole object for profiling

```
1  var callStack = [];
2  Global.classes(true).forEach(function(klass){
3    var obj = klass.prototype;
4    var functionNames = Functions.own(obj);
5    functionNames.forEach(function(functionName){
6
7    //as before ...
8
9    }
10 }
```

Code Example 45: instrumenting a whole object for profiling

This is already very promising, however, running this code would break the system. When instrumenting classes, especially the basic ones, we have several problems. Firstly, the prototype holds a reference to the constructor. Although the constructor is a function, it is not a method of the prototype and must therefore be excluded from the list of methods wrapped.

When wrapping all methods of all classes, we may also wrap e.g. `pop` and `push` of the class `Array`. As these are called by our profiling function, this results in endless recursion. Yet we want to include these functions, as they might be important in the profiled application.

To be able to include functions in the profiling, that are used by the profiler itself, we need to be able to turn the profiling on and of for snippets of code without uninstrumenting the functions.

This is exactly what context oriented programming (COP) does. [27] Lively Kernel includes ContextJS, a JavaScript COP implementation, so it would be quite easy to use. A method tracer has even been implemented before using ContextJS. [41] However, ContextJS has a decent amount of execution time overhead. [38] As functions with deep call trees are affected by the overhead more than flat ones, too much overhead may change the results, rendering the profiler useless.

We decided to solve the problem with a minimalistic implementation of a COP like functionality. A variable accessible to all profiling functions holds a Boolean value that defines if the profiling steps should be executed or not.

```
1  var newFunction = function () {
2    if(active){
3      active = false:
4      //as before...
5      active = true;
6    }
7
8    var out = newFunction.apply(this, arguments);
9
10   if(active){
11     active = false;
12     //as before...
13     active = true;
14   }
15   return out;
16 }
```

Code Example 46: instrumenting a whole object for profiling

As you can see, we can not avoid calling the method `apply` of the system class `Function` with `active` set to `true`. However, as this function should normally be native and very fast, we can safely exclude it from our list of functions we want to instrument.

User Interface In this section we described, how to measure execution times in order to find bottle necks. Collecting this data is only half the work, though. In order for humans to be able to read it and find the problem, we need to present it in a comprehensible fashion. The user should furthermore be able to trigger the profiling easy and intuitively.

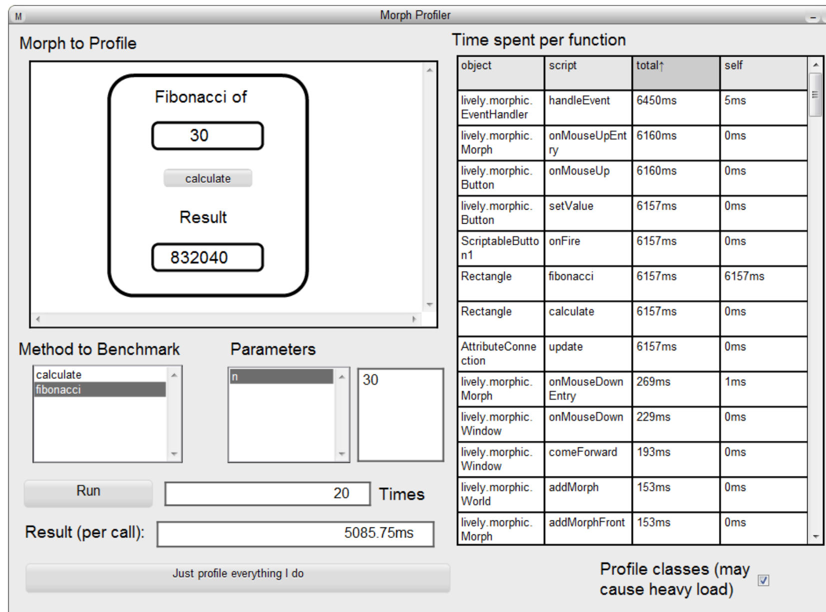


Figure 62: User Interface of the Morph Profiler. Example data retrieved via "Just profile everything I do".

The user interface we ended up with so far can be seen in figure 62. It consists primarily of two parts: on the left side one can choose what to profile and how, on the right side the results are presented. We decided to base the Profiling on morphs, so users need to put the morph under test in the upper left area. The profiler then automatically displays all scripts of that morph. To run a script, users select it in the left list and provide parameters via the right list and the small text pane. Note that the provided text is evaluated before running the script, so they can be retrieved by a script, but will not change throughout the profiling process. Finally, the users configure the number of times to run the script in a row, this can help reducing variances in the execution time. The users can start the profiling by pressing "Run".

The profiler then copies the morph to make sure the original is not changed. The chosen script will be executed the chosen number of times with the described profiling process enabled. Finally the copy and the wrapper functions will be removed and the results displayed. They are displayed as average per-run values, so the users are not bothered with unnecessary large numbers.

However, not every user knows exactly which function to run or which arguments to supply. Some times only the workflow causing trouble is known. For this case we created the "Just profile everything I do" button. When pressing this button, profiling will be turned on and all user interaction and resulting actions are measured. This is still limited to the morph under test. When finished, the user presses the button again (It will now read "I am done") and the profiling is finished. All results will then be displayed in the right pane. Note that the "Result (per call)" field is not updated, as there is no such information.

The result view on the right displays the name of the object (or class), the name of the function and the measured times for each function that has been called during the profiling process. To find the bottle necks faster, the list can be sorted by any column in any order by just clicking the header.

Instrumenting all classes can be very expensive in multiple ways. It needs time to install all the wrapper functions, the functions themselves produce some overhead and the huge amount of generation will need some time to be displayed. Apart from the performance, it might even hide the true bottle neck amongst all those entries. That is why profiling classes is optional, it is turned of by default, as this profiler is primarily designed for the development of parts.

This user interface still needs much improvement.(see section 5.5). Anyhow, it enables all Lively Kernel users to profile their parts and makes the search for bottle necks a real search rather than a guess and test scenario.

5.4 Results of the profiler

This section provides some examples of information gathered using the described profiler.

Insights from the Profiler When developing the profiler, we actually had some performance issues with the User Interface. The profiling itself is smooth, but showing the results needs quite a lot of time. So we decided to use the profiler on itself. We did not profile the profiling process itself, because this would lead to way too many levels of indirection. In fact even one Profiler may exceed the maximum call stack size, as the stack depth of the call is almost doubled. Two nested profilers cause way to many of these exceptions to deliver reliable meassurements. We could nevertheless profile the aggregation and rendering of the results. This way we found out, that the creation and rendering of the morphs building the table eats up the time. See section 5.5 for our approach to solve this problem.

We also used the profiler as an experimental tool, one example: We encountered three ways to iterate over arrays, that were described having different speeds by different sources. To find out what was really happening, we created a box morph, that executed all three of them as seperate functions. The profiling result for this functions could then be compared to compare the speed of the iteration. We could see, that all three methods performed equally good and there is no performance argument for choosing one over the other.

5.5 Future Work

This section describes which steps could be taken to improve the benchmarks and the profiler described above.

The benchmarks provide a good impression of the overall performance of Lively Kernel on various Systems. However, the selection of system components represented in the benchmarks is made solely on manual observance of usage. This does not guarantee the selection to be representative. A better way would be collecting usage statistics over a couple of weeks. These statistics can be used to create benchmarks resembling real interaction. Nevertheless, to preserve comparability, these would have to be new benchmarks and should not replace the others.

As described in section 5.4, the rendering of the information of a profiling session takes a lot of time. This originates in the huge number of morphs that need to be created, styled, positioned and rendered to build up the table. A possible solution to this performance issue could be a "table-morph", that would not build the table out of submorphs but rather use an html table.

The profiler gives a lot of information, too much in some cases. Allthough it provides a sortable view of the results, it can get difficult to find the bottle neck among the dozens of called functions. The profiler of the Google Chrome debugging tools solves this problem with a tree view, that resembles a call tree with the additional information of runtime per node. This could be usefull for the Lively Kernel profiler, too. As this requires more information to be stored, we would need to evaluate the cost of the higher overhead againts the use of the tree view.

The userinterface of the profiler is only a very simple one, one could say a first draft. It would probably enhance the usability to revise it in multiple design iterations. Possible changes could inlcude for example a possibility to enter a list of classes to instrument or a recording feature for workflows, so one can retest them automatically.

5.6 Related Work

We have created a benchmark suite specialized on Lively Kernel, but it is still a JavaScript benchmark. There are several other benchmarks that aim to achieve a comparison of the overall performance. This section compares these to our benchmark.

There are several different JavaScript benchmarks. We have chosen some prominent examples to compare our results with, namely SunSpider, V8 benchmarks, Kraken and Dromaeo.

The SunSpider benchmark was developed by the WebKit team and was one of the first attempts to measure the performance of "actual problems developers solve with JavaScript today". [11] It avoids micro benchmarks, which repeat small tasks severall times, because the designers think these are not representative. This is a huge difference to our Lively Kernel benchmark suite. SunSpider runs tasks like ray tracing, cryptography and decompression. It aims to keep a balance between different language features of JavaScript and determines an error range of the results. [11]

The V8 benchmarks are developed by Google as a reference to optimize their JavaScript engine of the same name. It includes some heavy tasks and some micro benchmarks and gives better results with Google Chrome than other browsers, because Google Chrome uses V8 and is therefore optimized to achieve good results in this suite. [22]

The Kraken benchmark from Mozilla is similar to the SunSpider, in the way it runs its small number of heavy tasks and determines error ranges. It uses other example tasks however, that are even more complex. Examples are image manipulation, path finding and audio processing. Mozilla developers aimed for more realism than other benchmarks. [49]

The Dromaeo test suite is an attempt of Mozilla to unify all the different Benchmarks and tests available. It executes micro benchmarks as well as larger tasks from different sources. SunSpider, V8 and several other suites are included in Dromaero. Mozilla expanded this already large collection by some own tests and implemented a test system with a minimal number of runs and a minimal running time similar to ours. [52]

We ran these 3 benchmark suites on our most relevant test devices to compare them to our benchmark. We used SunSpider 0.9.1, V8 benchmarks version 7, Kraken 1.1 and Dromaero at 2012-06-29. We did not run the DOM tests of Dromaero, because the mobile Safari kept timing out and crashing. We ran only the JavaScript tests (options: dromaero | sunspider | v8).

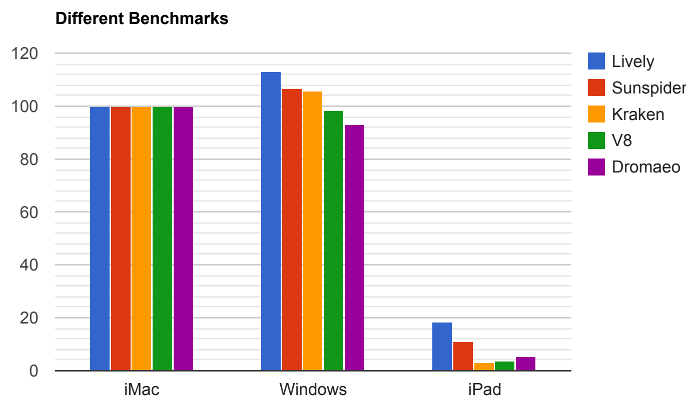


Figure 63: Results of different JavaScript Benchmarks. All run in Google Chrome, where possible, Safari on the iPad. Higher is better.

The results of these benchmarks compared to ours can be seen in figure 63. Note that we scaled all results to our reference system to achieve comparability. It is clearly visible, that our benchmark suite favors Windows over the iMac and grants the iPad a higher score than the other benchmarks. This means the subset of JavaScript tested by our suite is probably well supported by the mobile Safari, although it lacks in some features we did not use.

5.7 Conclusion

Performance is crucial to Lively Kernel. The highly interactive character and the goal of fluent programming require fast processing and computation.

In this thesis we described and implemented two tools, that can help us on the way to a perfomant system. A benchmark suite to provide objective information about performance and a profiler to track down bottle necks.

The benchmarks allow us to compare the performance of Lively Kernel on diferent devices, using different browsers and across time (and therefore state of Lively Kernel). It provides an objective meassurement of performance that could only be feeled and guessed before. We selected a representative set of features to base our benchmarks upon and created an implementation framework, that allows for easy extension and configuration of the benchmark suite. We implemented our suite and ran it on severall plattforms.

The profiler is helpfull when fixing performance issues. It provides an efficient and empirical way to find the bottle neck. We implemented the profiling mechanism using wrapper functions and build a decent user interface. All resulting data is presented human readable in a sortable table. We are now able to profile scripts of parts or the whole class system. We can even run a script severall times in order to compensate statistical variances.

Our work has paved the way towards a fluent user experience by providing objective meassurements and tools to notice, fund and fix performance issues.

A Specifications of test sytems

A.1 iMac

- iMac 11.2
- Intel Core i3
- 3.06 GHz
- 2 Cores
- 256 L2 Cache per Core
- 4 MB L3 Cache
- 4 GB 1333 MHz DDR3 RAM
- 5.86 GT/s
- Boot-ROM-Version IM112.0057.800
- SMC-version 1.64fS

A.2 Windows

- Windows 7 Enterprise SP 1
- Intel Core i5
- 2.66 GHz
- 4 Cores
- 4 x 32 KBytes L1 Cache
- 4 x 256 KBytes L2 Cache
- 8192 KBytes L3 Cache
- 6 GB 667 MHz DD3 RAM (3 x 2 GB)

References

1. Alan Kay, Dan Ingalls, T.K.: Squeak smalltalk (05 2012), http://squeak.org, visited 19.06.2012
2. Albinsson, P.A., Zhai, S.: High precision touch screen interaction. In: Proceedings of the SIGCHI conference on Human factors in computing systems. CHI '03, ACM, New York, NY, USA (2003)
3. Allen-Conn, B.J., Rose., K.: Powerful ideas in the classroom. Viewpoints Research Institute, Inc. (2003)
4. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide Time to Relax. O'Reilly Media, Inc., 1st edn. (2010)
5. Apple Inc.: Touchevent class reference (12 2010), http://developer.apple.com/library/safari/navigation/, visited 16.06.2012
6. Apple Inc.: Devicemotionevent class reference (06 2011), http://developer.apple.com/library/safari/navigation/, visited 16.06.2012
7. Apple Inc.: Deviceorientationevent class reference (10 2011), http://developer.apple.com/library/safari/navigation/, visited 16.06.2012
8. Apple Inc.: Safari web content guide (10 2011), http://developer.apple.com/library/safari/navigation/, visited 16.06.2012
9. Apple Inc.: iOS Human Interface Guidelines (06 2012), https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/MobileHIG.pdf, visited 19.06.2012
10. Apple Inc.: iOS Human Interface Guidelines (03 2012), http://developer.apple.com/library/safari/navigation/, visited 21.06.2012
11. Apple Inc: SunSpider JavaScript Benchmark (06 2012), http://www.webkit.org/perf/sunspider/sunspider.html, visited 29.06.2012
12. Benko, H., Wilson, A.D., Baudisch, P.: Precise selection techniques for multi-touch screens. In: Proceedings of the SIGCHI conference on Human Factors in computing systems. CHI '06, ACM, New York, NY, USA (2006)
13. Brubeck, M., Moon, S., Schepers, D.: Touch events (12 2011), http://www.w3.org/TR/2011/CR-touch-events-20111215/, visited 25.06.2012
14. Callahan, J., Hopkins, D., Weiser, M., Shneiderman, B.: An empirical comparison of pie vs. linear menus. In: Proceedings of the SIGCHI conference on Human factors in computing systems. CHI '88, ACM, New York, NY, USA (1988)
15. Czuchra, M.: Offline Worlds. Automated Client-Side Persistence in Lively Kernel. Master's thesis, Software Architecture Group, Hasso-Plattner-Institute, University of Potsdam, Germany (2012)
16. Dannert, J.: WebCards. Entwurf und Implementierung eines kollaborativen, graphischen Web-Entwicklungssystems für Endanwender. Master's thesis, Software Architecture Group, Hasso-Plattner-Institute, University of Potsdam, Germany (2009)
17. Dourish, P., Bellotti, V.: Awareness and coordination in shared workspaces. In: Proceedings of the 1992 ACM conference on Computer-supported cooperative work. CSCW '92, ACM, New York, NY, USA (1992)
18. Dourish, P., Bly, S.: Portholes: supporting awareness in a distributed work group. In: Proceedings of the SIGCHI conference on Human factors in computing systems. CHI '92, ACM, New York, NY, USA (1992)
19. Ecma International: Ecmascript language specification (06 2011), http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, visited 16.06.2012

20. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. SIGMOD Rec. 18(2), 399–407 (06 1989)
21. Fitts, P.M.: The information capacity of the human motor system in controlling the amplitude of movement. Journal of experimental psychology 47(6), 381–391 (06 1954)
22. Google Inc: V8 Benchmark Suite - version 7 (06 2012), http://v8.googlecode.com/svn/data/benchmarks/v7/run.html, visited 29.06.2012
23. Google Inc. and the Open Handset Alliance: Iconography | android developers (05 2012), https://developer.android.com/design/style/iconography.html, visited 19.06.2012
24. Google Inc. and the Open Handset Alliance: Selection | android developers (05 2012), http://developer.android.com/design/patterns/selection.html, visited 19.06.2012
25. Graham, T., Phillips, W., Wolfe, C.: Quality analysis of distribution architectures for synchronous groupware. International Conference on Collaborative Computing: Networking, Applications and Worksharing 0, 41 (2006)
26. Hick, W.E.: On the rate of gain of information. Quarterly Journal of Experimental Psychology 4(1), 11–26 (1952)
27. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented Programming (03 2008), http://www.hirschfeld.org/writings/index.html, visited 29.06.2012
28. Hopkins, D.: The design and implementation of pie menus. Dr. Dobb's J. 16(12), 16–26 (12 1991)
29. HPI Software Architecture Group: Lively - an explorative authoring environment, http://lively-kernel.org/lively/index.html, visited 16.06.2012
30. Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A., Mikkonen, T.: The lively kernel a self-supporting system on a web page. In: Hirschfeld, R., Rose, K. (eds.) Self-Sustaining Systems, Lecture Notes in Computer Science, vol. 5146. Springer Berlin / Heidelberg (2008), 10.1007/978-3-540-89275-5_2
31. Jonathan P. Munson, P.D.: A flexible object merging framework (1994)
32. Just, H.: ActiveParts: A Server-Side Lively Kernel Runtime. Master's thesis, Hasso-Plattner-Institut, Potsdam (05 2005)
33. Kay, A.C.: A personal computer for children of all ages. In: Proceedings of the ACM National Conference. Boston, Estados Unidos (08 1972)
34. Knittl-Frank, D.: Analysis and Comparison of Distributed Version Control Systems. bachelor thesis, University of Applied Sciences, Upper Austria (2010)
35. Koch, P.P.: The fifth position value (12 2010), http://www.quirksmode.org/blog/archives/2010/12/the_fifth_posit.html, visited 18.06.2012
36. Krahn, R.: Mouse and keyboard events in lively (08 2011), http://lively-kernel.org/repository/webwerkstatt/documentation/Events.xhtml, visited 16.06.2012
37. Krahn, R., Ingalls, D., Hirschfeld, R., Lincke, J., Palacz, K.: Lively wiki a development environment for creating and sharing active web content. In: Proceedings of the 5th International Symposium on Wikis and Open Collaboration. WikiSym '09, ACM, New York, NY, USA (2009)
38. Krahn, R., Lincke, J., Hirschfeld, R.: Efficient Layer Activation in ContextJS (01 2012), http://www.hirschfeld.org/writings/index.html, visited 27.06.2012
39. Leithead, T., Rossi, J., Schepers, D., Höhrmann, B., Hégaret, P.L., Pixley, T.: Document object model (dom) level 3 events specification (06 2012), http://www.w3.org/TR/DOM-Level-3-Events/, visited 16.06.2012

40. Leuf, B., Cunningham, W.: The Wiki way: quick collaboration on the Web. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
41. Lincke, J., Krahn, R., Hirschfeld, R.: Implementing Scoped Method Tracing with ContextJS (07 2012), http://www.hirschfeld.org/writings/index.html, visited 27.06.2012
42. Lincke, J., Krahn, R., Ingalls, D., Hirschfeld, R.: Lively fabrik a web-based end-user programming environment. In: Proceedings of the 2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing. C5 '09, IEEE Computer Society, Washington, DC, USA (2009)
43. Lincke, J., Krahn, R., Ingalls, D., Röder, M., Hirschfeld, R.: The lively partsbin–a cloud-based repository for collaborative development of active web content. Hawaii International Conference on System Sciences 0 (2012)
44. Lincke, J., Krahn, R., Ingalls, D., Röder, M., Hirschfeld, R.: The lively partsbin-a cloud-based repository for collaborative development of active web content. In: HICSS (2012)
45. Maloney, J.: Morphic: The Self User Interface Framework (07 1995), http://ftp.squeak.org/docs/Self-4.0-UI-Framework.pdf, visited 21.06.2012
46. Maloney, J.: Morphic: The Self User Interface Framework. Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043 USA (1995)
47. McDirmid, S.: Coding at the speed of touch. In: Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software. ONWARD '11, ACM, New York, NY, USA (2011)
48. Mikkonen, T., Taivalsaari, A.: Creating a mobile web application platform: the lively kernel experiences. In: Proceedings of the 2009 ACM symposium on Applied Computing. SAC '09, ACM, New York, NY, USA (2009)
49. Mozilla Foundation: Kraken JavaScript Benchmark (version 1.1) (06 2012), http://krakenbenchmark.mozilla.org/, visited 29.06.2012
50. Ohkubo, M., Ishii, H.: Design and implementation of a shared workspace by integrating individual workspaces. SIGOIS Bull. 11(2-3), 142–146 (03 1990)
51. Rauch, G.: socket.io (2012), http://socket.io, visited 27.06.2012
52. Resig, J.: Dromaeo: JavaScript Performance Testing (06 2012), http://dromaeo.com/, visited 29.06.2012
53. Sebastian Herzberg, D.D.: Content-Tracking mit Git. Beleg zur Lehrveranstaltung Softwareentwicklungswerkzeuge (2011)
54. Smus, B.: Multi-touch web development (06 2011), http://www.html5rocks.com/en/mobile/touch/, visited 16.06.2012
55. Stefik, M., Bobrow, D.G., Foster, G., Lanning, S., Tatar, D.: Wysiwis revised: early experiences with multiuser interfaces. ACM Trans. Inf. Syst. 5(2), 147–167 (04 1987)
56. Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S., Suchman, L.: Beyond the chalkboard: computer support for collaboration and problem solving in meetings. Commun. ACM 30(1), 32–47 (01 1987)
57. Stewart, J., Bederson, B.B., Druin, A.: Single display groupware: a model for co-present collaboration. In: Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit. CHI '99, ACM, New York, NY, USA (1999)
58. Taivalsaari, A., Mikkonen, T., Ingalls, D., Palacz, K.: Web browser as an application platform. Software Engineering and Advanced Applications, Euromicro Conference 0, 293–302 (2008)
59. Tilkov, S., Vinoski, S.: Node.js: Using javascript to build high-performance network programs. IEEE Internet Computing 14, 80–83 (2010)

60. Tillmann, N., Moskal, M., de Halleux, J., Fahndrich, M.: Touchdevelop: programming cloud-connected mobile devices via touchscreen. In: Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software. ONWARD '11, ACM, New York, NY, USA (2011)
61. Ungar, D., Smith, R.B.: Self. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages. HOPL III, ACM, New York, NY, USA (2007)
62. WHATWG: Timers (06 2012), `http://www.whatwg.org/specs/web-apps/current-work/multipage/webappapis.html`, visited 20.06.2012
63. World Wide Web Consortium: `http://www.w3.org/`, visited 16.06.2012

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
71	978-3-86956-231-5	**Vereinfachung der Entwicklung von Geschäftsanwendungen durch Konsolidierung von Programmier-konzepten und -technologien**	Lenoi Berov, Johannes Henning, Toni Mattis, Patrick Rein, Robin Schreiber, Eric Seckler, Bastian Steinert, Robert Hirschfeld
70	978-3-86956-230-8	**HPI Future SOC Lab - Proceedings 2011**	Christoph Meinel, Andreas Polze, Gerhard Oswald, Rolf Stromann, Ulrike Seibold, Doc D'Errico
69	978-3-86956-229-2	**Akzeptanz und Nutzerfreundlichkeit der AusweisApp: Eine qualitative Untersuchung**	Susanne Asheuer, Joy Belgassem, Wiete Eichorn, Rio Leipold, Lucas Licht, Christoph Meinel, Anne Schanz, Maxim Schnjakin
68	978-3-86956-225-4	**Fünfter Deutscher IPv6 Gipfel 2012**	Christoph Meinel, Harald Sack (Hrsg.)
67	978-3-86956-228-5	**Cache Conscious Column Organization in In-Memory Column Stores**	David Schalb, Jens Krüger, Hasso Plattner
66	978-3-86956-227-8	**Model-Driven Engineering of Adaptation Engines for Self-Adaptive Software**	Thomas Vogel, Holger Giese
65	978-3-86956-226-1	**Scalable Compatibility for Embedded Real-Time components via Language Progressive Timed Automata**	Stefan Neumann, Holger Giese
64	978-3-86956-217-9	**Cyber-Physical Systems with Dynamic Structure: Towards Modeling and Verification of Inductive Invariants**	Basil Becker, Holger Giese
63	978-3-86956-204-9	**Theories and Intricacies of Information Security Problems**	Anne V. D. M. Kayem, Christoph Meinel (Eds.)
62	978-3-86956-212-4	**Covering or Complete? Discovering Conditional Inclusion Dependencies**	Jana Bauckmann, Ziawasch Abedjan, Ulf Leser, Heiko Müller, Felix Naumann
61	978-3-86956-194-3	**Vierter Deutscher IPv6 Gipfel 2011**	Christoph Meinel, Harald Sack (Hrsg.)
60	978-3-86956-201-8	**Understanding Cryptic Schemata in Large Extract-Transform-Load Systems**	Alexander Albrecht, Felix Naumann
59	978-3-86956-193-6	**The JCop Language Specification**	Malte Appeltauer, Robert Hirschfeld
58	978-3-86956-192-9	**MDE Settings in SAP: A Descriptive Field Study**	Regina Hebig, Holger Giese
57	978-3-86956-191-2	**Industrial Case Study on the Integration of SysML and AUTOSAR with Triple Graph Grammars**	Holger Giese, Stephan Hildebrandt, Stefan Neumann, Sebastian Wätzoldt
56	978-3-86956-171-4	**Quantitative Modeling and Analysis of Service-Oriented Real-Time Systems using Interval Probabilistic Timed Automata**	Christian Krause, Holger Giese

ISBN 978-3-86956-232-2
ISSN 1613-5652