Preface

Reading mail, writing articles, planing projects are typical content creating activities done in a Web-browser. Programming was not yet one of them, but environments and tools for scripting and authoring in the Web are becoming more popular. Web-based Programming environments are an approach to allow developers to program their applications from inside a Web-browser.

The idea is not new. The first occasion I have seen someone typing code into his Web-browser and executing it was in a Simulations and Animations course during my studies. The lecturer taught us GPSSH, an old simulation language. He used the white board a lot, but occasionally he opened an example on a Web-site, edited and executed source code. It was a simple but working setup. GPSSH was not indented to be an interactive language so this approach was rather unconventional and at the same time very interesting. The second time I came into contact with Web-based programming was while I wrote my diploma thesis. There I learned about Active Essay [2]. The essay explained evolutionary algorithms in the form of scripts that could be run and changed by the reader from within the Web-browser. Years later these ideas came back in the form of TinLizzy [4], a wiki which allowed to edit rich text and scripts visually. This environment allowed to create such active essays right in the Web [5], no special authoring environments or any additional setup were needed to create the text or scripts.

Our project seminar Web-based development environments was focused on the Lively Kernel, one of such environments [1]. For the Seminar we used the Lively Kernel-based Wiki [3] WebWerkstatt. This wiki allows to collaboratively create and edit textual, graphical, and programable content from within the Web-browser. The students worked on various topics extending that environment: They worked on synchronous collaboration, interactive layouting, text composition, source code management, and portable rendering of the Lively Kernel. They all developed their projects inside WebWerkstatt. There they could program, document, and present their progress and results. The Web-based development environment allowed them to integrate their demos with their presentations in a very interactive way.

The feedback of the students using the a Web-based development environment helped us to see the strength and weaknesses of this approach, allowing us to evolve the system as it was used.

June 2011 Jens Lincke

References

1. 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.) S3 2008. LNCS 5146, Springer-Verlag Berlin Heidelberg (2008)
2. Kay, A.: Active Essays—What is 'Serious Discourse' and 'Literacy'? (1996)
3. Krahn, R., Ingalls, D., Hirschfeld, R., Lincke, J., Palacz, K.: Lively Wiki A Development Environment for Creating and Sharing Active Web Content. In: WikiSym '09. ACM (2009)
4. Ohshima, Y., Yamamiya, T., Wallace, S., Raab, A.: TinLizzie WysiWiki and WikiPhone: Alternative approaches to asynchronous and synchronous collaboration on the Web. In: C5 '07: Proceedings of the Fifth International Conference on Creating, Connecting and Collaborating through Computing. pp. 36–46. IEEE Computer Society, Washington, DC, USA (2007)
5. Yamamiya, T., Warth, A., Kaehler, T.: Active Essays on the Web. In: C5 '09: Proceedings of the Seventh International Conference on Creating, Connecting and Collaborating through Computing. pp. 3–10. IEEE Computer Society, Los Alamitos, CA, USA (2009)

Colophon

This report was typeset by LaTeX 2ε with pdf/ε-TeX using the Lecture Notes in Computer Science (LNCS) LaTeX class. The body text is set 10/12 pt on a 28.8 pc measure. The body type face is *Latin Modern*, a Type 1 PostScript version of *Computer Modern* by Donald E. Knuth. The listing type face is *Bera Mono*, based on the *Vera* family by Bitstream, Inc.; Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr. —*Tobias Pape*

Table of Contents

Web-based Development Environments

Lively HTML

Seminar Web-based Development Environments

David Jaeger and Robert Krahn

Hasso-Plattner-Institut, Potsdam
{firstname.lastname}@student.hpi.uni-potsdam.de

Abstract. The Lively Kernel is a web-based development and runtime environment using the Morphic framework as its user interface. Morphic requires a graphic system that is capable of applying 2D transformations to graphical objects as well as support the modification of other graphical attributes. Currently, the Lively Kernel uses SVG for rendering. However, when displaying many graphical objects the performance of current SVG rendering systems is not sufficient to provide a satisfying usability. This is especially the case when a lot of textual content is rendered.
The goal of this project is to implement and evaluate an HTML-based rendering system. We assume that Web browsers perform better rendering HTML. Additionally, we think that this approach will have other advantages, like enabling the usage of native widgets. We present our implementation approach and describe how the graphical requirements of an interactive system like the Lively Kernel can be mapped to HTML.

1 Introduction

The Lively Kernel (also Lively) is a collaborative and self-sustaining development and runtime environment for Web applications. It provides a rich and interactive graphics system by implementing the Morphic user interface [7, 8]. Currently Lively's rendering system is based on SVG[1] integrated into an XHTML document. The XHTML document can be loaded and modified in a Web browser.

The current rendering system is very flexible, giving a Lively developer direct control over how graphical elements are displayed. However, when visualizing larger amounts of objects like displaying several hundred classes in a system browser, the system becomes slow and unresponsive. This lessens the usability of Lively as a development platform.

In this paper we present a prototypical implementation of a rendering system for Lively that is completely based on HTML. With this experiment we want to find out whether Web browsers are able to display interactive and graphical objects required by Lively Kernel[2]. We assume that HTML rendering is more

[1] Scalable vector graphics (SVG) is an XML-based description language for two dimensional graphical objects

[2] First with HTML version 5 and CSS version 3 Web browsers are able to display complex graphical objects like polygons and apply custom transformations without relying on SVG. Lively Kernel requires these capabilities for its user interface.

optimized in Web browsers. Especially when displaying textual content we think that HTML rendering can provide better performance since Lively currently implements a manual text rendering process.

In addition to better performance for text content, an HTML scene graph can bring more significant benefits.

Compatibility The usage of HTML promises higher compatibility between Web browser, as HTML is more standardized and accepted in browsers than extensions like SVG. There is also a chance that Lively will run on the Internet Explorer.

HTML Integration Lively can integrate HTML elements residing in a surrounding HTML document, by *livelifying* the HTML element and inserting it into its scene graph.

We first present an overview on Lively's rendering model in section 2. We introduce our approach for integrating HTML rendering into Lively Kernel in section 3. In section 4 we describe how morphs are mapped to HTML elements. Section 5 evaluates the performance of the implementation. In 6 we introduce related work to HTML rendering in Lively. The last section concludes this report and provides ideas for future work.

2 Lively's Rendering Subsystem

The Lively Kernel has a modular rendering subsystem, which allows to replace facets of the graphical subsystem with moderate effort. Figure 1 provides a rough overview of the subsystem with its components.

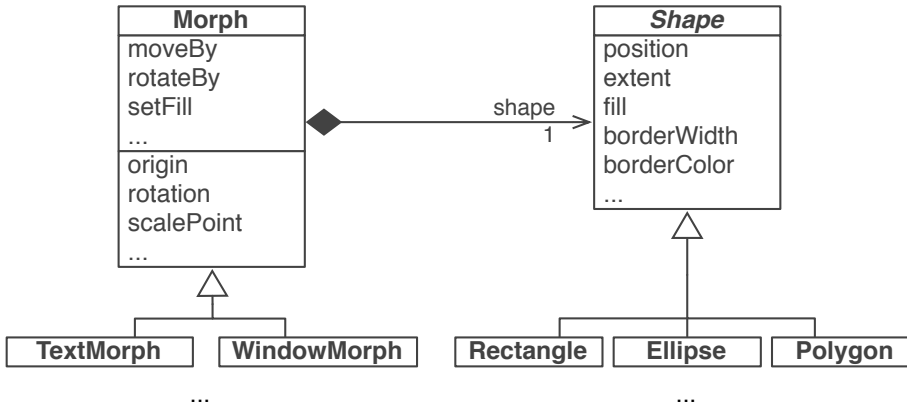

Fig. 1: The rendering system in Lively Kernel uses the bridge pattern to separate the Morphic interface from its implementation.

The structure of the rendering subsystem follows the bridge pattern [5] which separates the Morphic interface from its concrete shape implementation. Morphic

provides a variety of abstract graphical objects which are called *morphs*. The concrete rendering is performed with the help of a graphical primitive called *shape*. A morph does not need to have a concrete graphical representation. Two morphs of the same type can have different shapes. However, a morph and a shape always have a one-to-one relationship. The appearance of a shape is controlled by visual properties provided by the owning morph.

3 Concept

We have emphasized the modularity of Lively's rendering subsystem. Essentially, the separation of abstract morphs from their concrete graphical representation allows the switching to HTML rendering by only replacing the rendering specific shapes. In this section, we first show how this can be achieved in general and then show our approaches for a selection of core problems.

3.1 Replacing the Rendering Subsystem

The bridge pattern in the rendering subsystem narrows the needed reimplementation in Lively down to shapes and their supporting objects. Ideally, the Morphic interface and all code depending on morphs will remain untouched. This means, that only a minor part of the Lively Kernel needs to be changed.

A mechanism that allows us to replace the rendering subsystem while we still use the Lively Kernel for development of the same is context oriented programming (COP). The details of this approach are described in section 4.

3.2 Core Issues for Implementing a Rendering Subsystem

The implementation of the HTML-based rendering requires various system changes. We identified five issues and discuss how they are addressed.

HTML representation of existing graphical objects and their properties Lively uses an SVG scene graph, which is not intended for adding HTML elements. An HTML scene graph better fits this requirement and therefore should replace its SVG counterpart. SVG shape elements can be simply translated into HTML shape elements. Unfortunately, even a part of the Morphic implementation influences elements in the SVG scene graph. Thus, we should keep the structure of the HTML and SVG scene graph similar, so that changes in the Morphic code stays at a minimum.

Transformations A transformation consists of three components, which are *scaling*, *rotation* and *translation*. Lively stores a transformation in two forms: as a combined transformation matrix and as separate attributes. The matrix simplifies graphical operations, whereas attributes allow fast access to the components of a transformation. Each Morph has its own transformation, which is relative to the transformation of its parent morph. We should encourage the same ideas for our HTML specific shapes.

Serialization The current serialization of the Lively Kernel is based on the SVG DOM. However, the basic principle of serialization should be similar for HTML. Possible incompatibilities have to be found in advance. The only important thing is that all properties are serialized, so that a serialized Lively world can be completely restored.

Text Text rendering will directly use the DOM tree for its representation, rather than shapes. The layouting of the text is handled by Lively. By following this approach, we can use a major part of the existing SVG text rendering mechanisms.

Event System The event dispatching mechanism of Lively uses the Morphic hierarchy. The only point the dispatching interferes with interferes with the shape logic is the mapping of point in the world to the underlying shape. So the dispatching asks a shape for containment of a given point.

4 Implementation

We have identified five major problems for the implementation of HTML rendering. The mechanisms for the event system, text rendering and serialization are similar to the SVG implementation. In the first part of this section we will have a closer look at the problem of HTML representation of graphical objects. The second part covers the use of context oriented programming during our implementation phase. At the end, we show how native HTML widgets are integrated into Lively

4.1 Mapping of Morphs into an HTML DOM Tree

Figure 2 shows how a rectangular morph is rendered in Lively HTML. The graphical appearance of the morph seen by the user is shown at the top. The morph instance and its shape provide the interface for programmatically changing graphical attributes like the extent or color. They are shown below. The shape object is responsible for mapping those attributes to DOM elements. The generated subtree of the DOM is shown at the bottom.

For introducing a new rendering system the shape implementation needs to be changed because it is directly responsible for creating the graphical representation. In the following, we therefore describe how the shape implementation was modified to allow HTML-based rendering.

A shape renders itself to the DOM tree by creating an element appropriate for its shape. The relation between a shape object and its element in the DOM can be seen in the table of figure 3.

A shape object basically translates to two distinct HTML elements. `div`s are used for simple shapes, whereas `canvas` [2] elements are used for complex shapes like polygons or polylines.

When the shape element has been created by the shape object it is subsequently wrapped into a `div` element representing the morph. It is important to

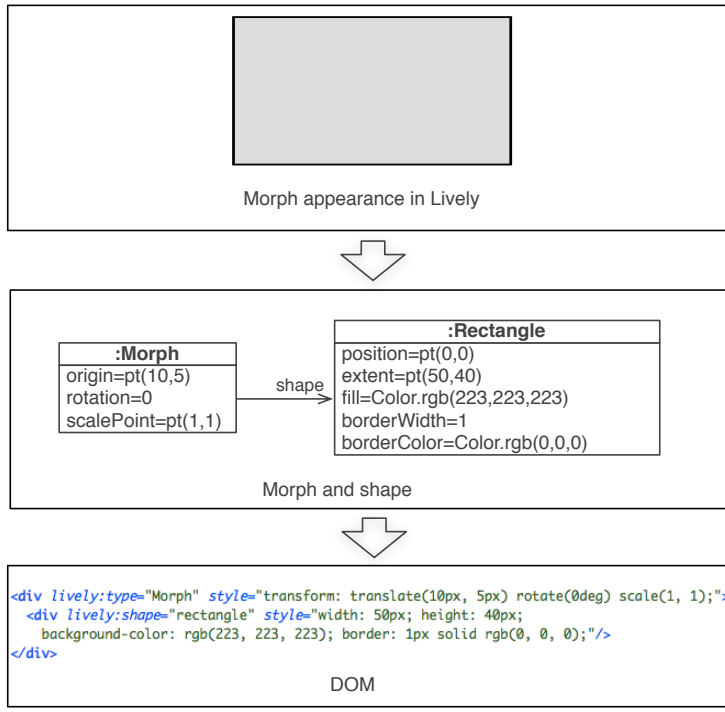

Fig. 2: This figure shows how a `Morph` with a `Rectangle` shape is displayed in Lively Kernel, represented programmatically in Lively, and rendered using the HTML DOM.

notice that the morph object does not render itself into the DOM tree[3]. Rendering is the responsibility of the shape.

The visual properties of a shape are added to the shape element by assigning a CSS declaration to their `style` attribute. In case of `canvas` elements, properties like color are set using the Canvas API. Table 4 shows the mapping of visual properties to CSS attributes.

Transformations are set with the `transform` [1] property. It is a new property of CSS3 and its value is very similar to the transformation value of SVG elements. Therefore, many code involving transformations can be used from the existing implementation.

Extent and the relative position of a shape in the morph are translated to CSS's standard positioning and extent attributes. For ellipses, we use rounded corners described with the `border-radius` CSS3 attribute value.

Color properties are assigned to the corresponding `foreground` and `background` attributes. A color can be of multiple types in the Lively Kernel. It can be a

[3] Several specialized morphs, however, directly interact with the DOM. This breaks the abstraction created by the bridge pattern and needs to be considered when modifying the DOM representation

Shape Type (lively:type)	HTML element	Remarks
rectangle	`div`	
ellipse	`div`	Use CSS3 `border-radius` property for round shape
polygon	`canvas`	Set all display properties via JavaScript
polyline	`canvas`	Set all display properties via JavaScript

Fig. 3: Mapping of shape types to HTML elements

visual property	CSS attribute	CSS3
origin, rotation, scaling	transform	yes
extent	width, height	no
pivot point [4]	top, left	no
fill color and opacity	background	partly
stroke color and opacity	foreground	partly
border color, opacity, radius, and width	border	partly
vertices [5]	canvas.beginPath	no

Fig. 4: Visual properties are mapped to CSS attributes.

simple color consisting of a *red*, *green* and *blue* value or it can be a gradient. Gradients are currently only supported for WebKit-based Web browsers, which support the `webkit-gradient` CSS attribute. Anyway, this can be extended to also support other Web browsers.

Vertices cannot be mapped to HTML `div` elements. Vertices have to be set using the canvas API introduced in HTML5.

4.2 Using ContextJS

We want to use the Lively Kernel while we are changing the rendering subsystem. Since the rendering subsystem is essential for a Lively world to render, an iterative development approach in not feasible. Additionally, we want to allow displaying an HTML and SVG world on the same Web document simultaneously. We were then able to modify the HTML world using running in the SVG world.

A solution that fulfills both requirements is context oriented programming. With this, we can execute rendering specific code either in an HTML context or an SVG context. As long as the HTML rendering subsystem is not finished, the Lively Kernel is executed in an SVG context, whereas all testing for the new HTML rendering subsystem is done in an HTML context. Once the HTML rendering subsystem is completed, the context can be assigned individually on world creation.

Context-specific Code We have identified the shape classes and their supporting objects as the primary point requiring reimplementation in order to support HTML rendering. The Morphic interface, with some exceptions, does not need to be changed. As a result, we only need to make the shape code context-aware. This can be seen in figure 5.

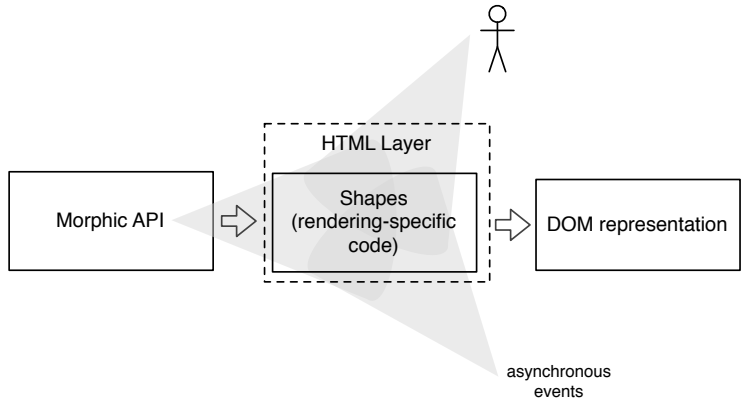

Fig. 5: Locations for layered code and the different types of layer activation

With context oriented programming, there will be one shape implementation for HTML and another one for SVG. In the figure, the HTML context is used by activating the HTML layer.

Layer Activation and Context Propagation In order to execute the shape code within a specific context, the corresponding layer first has to be activated. A layer can be activated in the ways listed below and as shown in figure 5.

Object Scope The layer can be activated programmatically by explicitly calling `setWithLayers` on an object. If this object initiates the execution of code involving the layered code, this code is executed under the layer's context.

Asynchronous Events Layer activation is lost when code is called asynchronously. Reasons for such asynchronous code executions are system events or scheduling. In order to avoid execution of shape code without a context, we have ensured that the same layer is activated in the asynchronous call, which was activated at the initiation of an asynchronous call.

User Events These events are similar to the *asynchronous code* above and are initiated by user interaction like mouse moves or keyboard usage. Here it also has to be ensured that the activated layer is propagated across user event callbacks.

4.3 Widgets based on HTML form and text elements

HTML provides form and text elements specialized for certain tasks. Input elements, for instance, cannot only appear as text inputs but can also be rendered as sliders, search panels, or buttons. These widgets are rendered like native operating system widgets, allowing application developers to create a more familiar look-and-feel.

To really benefit from HTML-based rendering, Lively users should be able to use these widgets as normal morphs. We implemented morphs providing access to widget attributes for some of these widgets. Figure 6 shows the widget set currently supported by Lively HTML. In figure 7 a system browser that uses these new morphs can be seen. The browser provides a better usability than existing Lively tools when a lot of data, for example a big class like `TextMorph`, is viewed.

The current implementation of the HTML widgets can be considered prototypical since the morphs directly interact with the DOM elements. This scheme breaks the abstraction of the bridge pattern and shape elements should be used to wrap those elements instead.

Fig. 6: HTML widgets.

5 Evaluation

We measured the performance of our implementation by running four different benchmarks[6]. Each benchmark was run with the HTML version (both with ContextJS deactivated and activated) and with the SVG version.

Ellipse Benchmark This benchmark creates moving and spinning ellipses to measure the performance of transformations. We record the frames per second during a fixed interval.

Mouse click and mouse move benchmark Measures how fast mouse events are dispatched. This simulates typical user inputs. Measurements were recorded in milliseconds.

[6] Specification of the test system:
Macbook Pro, 2.16 GHz Core 2 Duo, 3 GB RAM
Google Chrome 5 (Mozilla5.0 (Macintosh; U; Intel Mac OS X 10_6_4; en-US) AppleWebKit533.4 (KHTML, like Gecko) Chrome5.0.375.99 Safari533.4)

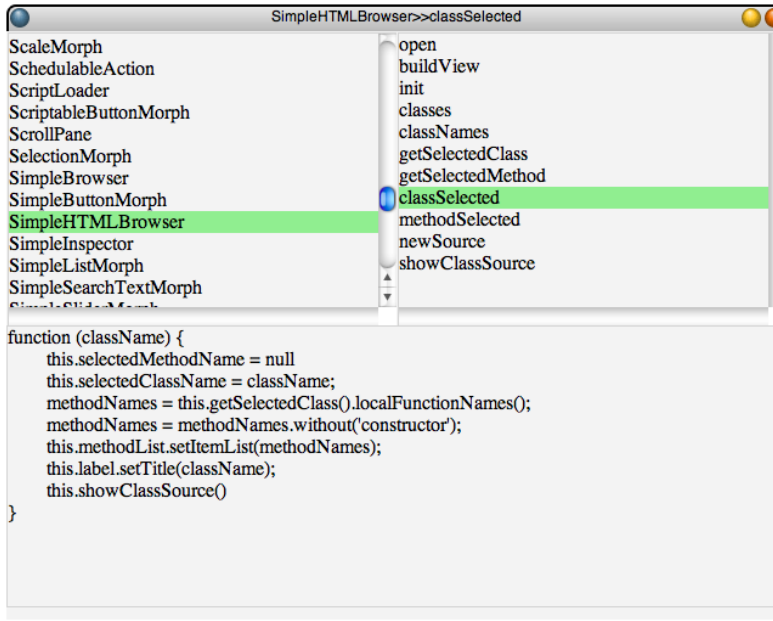

Fig. 7: HTML browser.

List selection benchmark We compare how fast list elements can be selected. The list element in the HTML version is a new list morph that uses only one shape being able to manage a list of text items. In the SVG version the list morph uses scroll morphs, clip morphs, and textmorphs for that task. This can be observed, for example, when interacting with lists having many entries. Measurements were recorded in milliseconds.

Figure 8 shows the result of the benchmarks. The ellipse benchmark was twice as fast in the SVG version than in HTML. This is unexpected and negates the theory that HTML rendering is better optimized than SVG rendering. The reason for that is probably the following: For modifying the CSS transformations the transformation data has to be converted into a string representation. The API for setting the CSS attributes without the need for string conversion is part of the DOM Level 2 CSS specification[7] , however, in none of the browsers we used for development was this interface implemented or working. SVG on the other hand, directly uses matrices for transformation which is significantly faster. The ellipse benchmark also shows the performance impact of ContextJS. When activating the HTML rendering layer the system was 50 percent slower than without it.

The mouse event benchmarks show that the HTML version is more than 25 percent faster than the HTML version. This is also not surprising since the event

[7] `http://www.w3.org/TR/DOM-Level-2-Stylecss.html#CSSCSSStyleDeclaration`

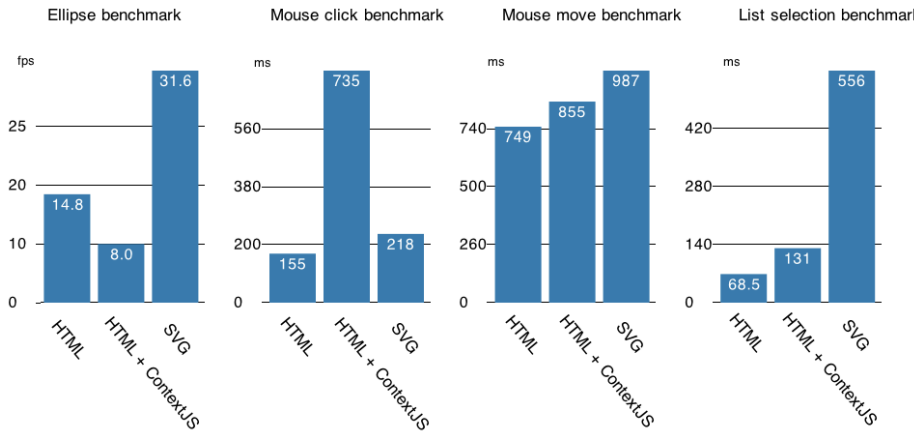

Fig. 8: Results of the benchmark comparing the performance of the HTML and SVG version.

dispatch mechanism that manually dispatches events using the morphic hierarchy is mainly the same as in the SVG-based Lively. However, the events are first captured by the canvas element that is in HTML Lively a HTML element. This element seems to be able to process the events faster. The results indicate that replacing the existing manual event dispatch mechanism with the browser-based event dispatch might be an effective optimization.

In the list selection benchmark the HTML version has a significant performance advantage. Selecting list items is almost 10 times faster than in the SVG version. Since we used a special HTML element that is able to scroll and directly manages its list items it becomes apparent that having native elements is an advantage compared to SVG where those elements are not existing.

Our evaluation results show, that we could improve performance for user interaction and for elements that play an important role in the tooling of Lively. We noticed a performance loss for transformations. Transformations are especially important for displaying animated non-textual content and we conclude that Lively HTML is not so well suited for these task than the SVG-based version. However, support for transformations is still in development and no browser seems to be fully specification compliant. We assume that this will change in the future because there is a trend towards rich graphic Web applications.

6 Related Work

Several projects experimented with the rendering system of the Lively Kernel.

To create a standalone version of Lively that is independent of Web browsers, the Gezira 2D graphic system was used for rendering [4]. This project was used for gaining inside into a minimal graphical system and was not continued.

Based on the DOM emulation module implemented by the Gezira project there also exists an implementation using the JavaFX scene graph for rendering morphs[9]. This project was not continued.

The HTML5 canvas element was used to render an entire Lively world using only raster graphics [6]. Also this project was not continued because the rendering performance of the canvas element turned out to be inferior compared with SVG rendering.

The Qt framework was used as a rendering system for Lively [3]. It allows to display morphs using Qt widgets. For running Lively Qt an installation of the Qt platform or a browser plugin is necessary.

7 Conclusion

We have successfully created a prototypical version of HTML-based rendering for the Lively Kernel. We showed that it is possible to implement all necessary graphical operations and structures using CSS and HTML.

We evaluated our approach to determine the performance characteristics. We found out that the HTML approach is much faster for text and list based tools (i.e. for development tools) since we were able to use specialized HTML elements. Thereby, the overall usability of the system is improved. The execution of transformations, however, is slower and it can be expected that a HTML-based system will not perform as well as the SVG version for Lively worlds with animated graphical objects.

We used ContextJS to be able to work with Lively's development tools and change its underlaying rendering system at the same time. We were able to mitigate the performance impact of ContextJS by merging layered code together with the base layer to extend the existing system statically.

Although we have not implemented SVG-based shapes into the HTML version, we consider that this is an option to still support vector graphics. This would make sense for objects that can be better rendered in SVG as in HTML, for example curves.

7.1 Future Work

We think it would be a valuable contribution to the Lively Kernel to integrate the new rendering system. The overall gain of usability and the capability to better support mobile devices like the iPad (because of the improved performance and native widgets) compensates for the transformation performance loss. We also expect that new Web browser versions will better support CSS transformations. Steps towards an integration are the following: Create a converter that transforms existing Lively SVG worlds into HTML documents, improve the text support for native HTML text (rich text is not yet supported), and replace the old morphs with the more performant native widgets.

We started experimenting with augmenting HTML documents. Loading Lively Kernel on-demand into an existing document enables to embed morphs as well

as modify and script existing HTML elements. We think further experiments with that approach can give valuable insights concerning document scripting and end-user programming.

References

1. CSS 2D Transforms Module Level 3. W3C Working Draft (2009), http://www.w3.org/TR/css3-2d-transforms/
2. HTML5 - The canvas element. W3C Working Draft (2010), http://www.w3.org/TR/html5/the-canvas-element.html
3. et. al, A.T.: Lively for Qt (2009), http://lively.cs.tut.fi/qt/
4. Alan Kay, e.a.: Steps toward the reinvention of programming. Tech. rep., Viewpoints Research Instititute (2008)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
6. Ingalls, D.: Lively Canvas Implementation. Source Code (2009), http://www.lively-kernel.org/repository/lively-wiki/lively/CanvasExpt.js
7. Maloney, J.H.: Morphic: The Self User Interface Framework. Sun Microsystems, Inc. (1995), ftp://ftp.squeak.org/docs/Self-4.0-UI-Framework.pdf
8. Maloney, J.H., Smith, R.B.: Directness and Liveness in the Morphic User Interface Construction Environment. In: UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology. pp. 21–28. ACM, New York, NY, USA (1995)
9. Palacz, K.: Lively FX (2008), http://lively-kernel.org/repository/lively-kernel/trunk/source/fx/ and http://blogs.sun.com/roger/entry/lively_fx

Lively Code Database

Seminar Web-based Development Environments

Tilman Giese and Marko Röder

Hasso-Plattner-Institut, Potsdam
{tilman.giese,marko.roeder}@student.hpi.uni-potsdam.de

Abstract. The Lively Kernel is a web-based development environment that is increasingly gaining popularity. Its server-side persistency is currently based on Subversion. As a file-based revision control system, Subversion does not allow for a more fine-granular revision control for JavaScript modules, classes, or methods. This paper presents an alternative persistency layer based on CouchDB that was integrated into the Lively Kernel to allow developers to store JavaScript code objects in a database.

1 Motivation and Goals

With the increasing capabilities and performance of today's web browsers web-based development environments have become popular. The approach to develop applications within the browser without any additional tool is very intriguing. The Lively Kernel is such a development environment that encourages developers to explore this new hands-on way of creating web applications.

The Lively Kernel is currently based on Subversion [1] as its persistency layer. The Subversion repository is directly accessed by the browser to retrieve all necessary files, in particular the XHTML and JavaScript files that contain the actual code to be executed. As Subversion is a file-oriented versioning control system, the versioning granularity in the Lively Kernel is also a file. However, using files as the smallest entities to contain JavaScript code entails several shortcomings. It also has a serious impact on how JavaScript code can be maintained by means of the built-in Lively Kernel source code browsers.

The first and foremost shortcoming of this approach is that JavaScript source code has to be parsed all the time in order to provide a fine-granular view on classes within a module and methods within a class. Syntax errors in the JavaScript file might render the entire file unparseable and thus no classes or methods might be visible in a code browser. By just relying on the means of JavaScript source code, it is also quite difficult to introduce metadata on classes or methods (like documentation or method categories). The current approach requires the developer to follow certain conventions, e.g. by declaring a class property with a particular name or by adding a special comment on top of a method definition.

And further issues arise from files being the entity of versioning control. Every small change to a method will always result in the entire file being saved and

assigned a new Subversion revision number. This can eventually lead to a very huge database. But more importantly, the connection between changes that logically belong together is lost as each change will create a new Subversion revision. Without specific knowledge of how changes were done it is thus impossible to revert back to a consistent state.

The goal of this project was to provide a more fine-granular revision control for JavaScript code objects. A code object is a source code artifact within the Lively Kernel environment that has a semantical notion of its own. A single code object can be composed of other smaller code objects. Examples of such code objects are methods, classes, and modules. The code objects should then be stored in a separate database rather than the Subversion repository. The existing Lively Kernel source code browsers should be extended to read and write code objects without the developer noticing the change in persistency. An interface should be provided to easily access code objects in the database. The JavaScript source code should still be accessible as a file and modules should be loadable from the database in the same way they were previously loaded. Figure 1 summarizes the change in persistency.

Fig. 1: Persistency Change

2 Implementation

The following sections present our solution in more detail and explain how it integrates into the Lively Kernel. The overall architecture is briefly described followed by an overview of the three major parts of the solution: the Code Database API to access the database, the Code Database Browser as the tool to develop applications and the Lively Kernel core extension that allows developers to load code from the database.

2.1 High-Level Architecture

As a preset requirement, CouchDB [2] had been chosen very early as the database technology to store code objects. CouchDB and Subversion have a different notion of revision, therefore the term needs clarification. A code object revision is a particular persistent representation of a code object at a specific point in time. Each code object revision corresponds to a particular CouchDB document. A code

object revision is identified by a code object revision number that is sequentially incremented for each new revision starting with 1 for the first revision. The revisions are collected in a code object revision history. Code object revisions can be flagged as drafts meaning they are not visible to developers unless specifically requested. Listing 2.1 shows the CouchDB document for a class revision. The revision properties are self-explanatory. CouchDB documents are JavaScript objects in JSON form.

Listing 2.1: CouchDB Document: Class Revision

```
{
   "_id": "Revision::25::TestModule::TestClass",
   "_rev": "25-2a4bb26e8b88a5447215e71e942f6870",
   "type": "class",
   "name": "TestClass",
   "documentation": "This is a class for testing purposes",
   "superclass": "Object",
   "methods": [
      "method1", "method2", "method3"
   ],
   "module": "TestModule"
}
```

In order to group several code objects together, there is also the concept of a change set. A set of changes to a set of code objects is called a change set thereby change set implicitly defines a logical unit of work. Persisting a change set in the database will create a change set revision. Similarly to a code object revision history there is also a change set revision history that keeps track of all the change sets. Listing 2.2 shows an extract of the CouchDB document that represents the change set revision history. It stores who committed the change set and which code objects were involved respectively what actions were performed on these code objects.

Listing 2.2: CouchDB Document: Change Set Revision History

```
{
   "_id": "ChangeSetHistory",
   "_rev": "200-c4a81c3b50ac4849e5595554a796e4fb",
   "currentRevision": 58,
   "revisionHistory": [
       ...
       {
           "revision": 36,
           "author": "m.roeder",
           "date": "2010-07-20T14:07:55Z",
           "message": "my commit message",
           "objects": [
               {
                   "name": "TestModule2",
                   "revision": 1,
                   "action": "add"
```

```
                },
                {
                    "name": "TestModule2::TestClass",
                    "revision": 1,
                    "action": "add"
                },
                {
                    "name": "TestModule1",
                    "revision": 2,
                    "action": "update"
                }
            ]
        },
        ...
    ]
}
```

In order to avoid name clashes when storing documents in CouchDB, there are the following conventions for document identifiers:

Listing 2.3: CouchDB Document Identifiers

```
- Revision::{RevNumber}::{ModuleName}[::{ClassName}[::{MethodName}]]
- RevisionHistory::{ModuleName}[::{ClassName}[::{MethodName}]]
- ChangeSetHistory
```

2.2 Code Database API

The Code Database API is the interface to all code objects stored in the database. It reflects the concepts described in the previous section. Figure 2 shows the classes that are involved.

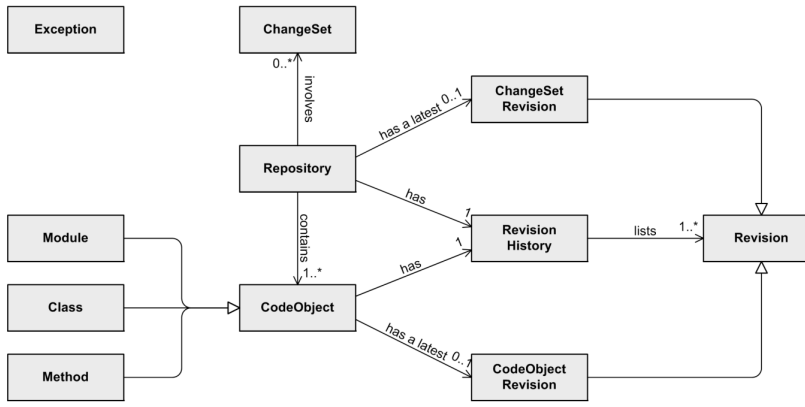

Fig. 2: Code Database API: Class Diagram

The main entry point is the class `Repository`. It provides methods to retrieve code objects and to create a new change set:

Listing 2.4: `Repository` Interface

```
# to retrieve a single code object
getCodeObject(type?, name+, includeDrafts?)
  e.g. getCodeObject(CDB.Module, 'TestModule')
  e.g. getCodeObject('TestModule', 'TestClass', true)

# to list code objects
listCodeObjects(type, name+, includeDrafts?)
  e.g. listCodeObject(CDB.Method, 'TestModule::TestClass')

# to create a new change set
createChangeSet()
```

A typical program flow is depicted in figure 3. First, a reference to the code database repository is created along with a new change set. Afterwards, the database can be queried for code objects using either the `getCodeObject` or `listCodeObjects` method. Before a code object can be modified, it has to be added to the change set. Saving the code object will persist all its properties in the database using a draft revision. Code objects can, of course, be saved several times until the change set is finally committed.

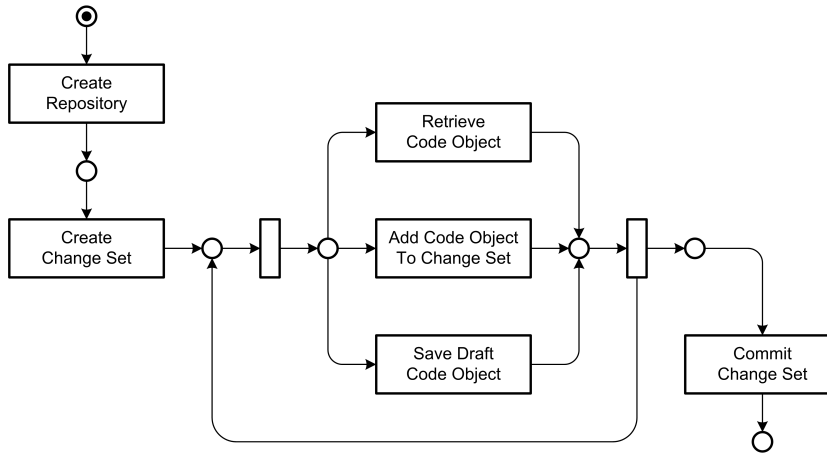

Fig. 3: Typical Program Flow

Draft revisions of code objects are snapshots that do not necessarily have to be in an executable state or consistent with other code objects. Saving a draft revision will create a new CouchDB document for the revision and modify (or create if the object does not exist yet) the CouchDB document for the code object revision history. Editing conflicts with other developers are detected upon

changing the revision history[1]. When the change set is committed, consistency is checked more properly, for example a class also has to be part of the change set if a method was added to it. After all consistency checks have successfully passed, the draft flag is removed from the latest revisions of all code objects and a new change set revision is created. Listing 2.5 shows a small example that adds a new method to an existing class, saves and commits the changes.

Listing 2.5: Example Program

```
var rep = new CDB.Repository();
var cs = rep.createChangeSet();

var klass = rep.getCodeObject(CDB.Klass, 'TestModule', 'TestClass');
var method = new CDB.Method('myNewMethod');
klass.addMethod(method);

method.documentation = 'Put here what the method does';
method.source = 'function() { ... }';

cs.add(klass);
cs.add(method);

klass.save();  // saves the changes as draft
method.save();

cs.commit();   // commits the changes
```

Throughout the Code Database API error conditions are signaled using exceptions. For example, `getCodeObject` will throw an `ObjectNotFoundException` if the specified code object is not in the database. commit can throw a `Consistency Exception` if any of the consistency constraints are not met or, like all database operations, a `DatabaseException` if there is technical problem with the database.

2.3 Code Database Browser

One implementation that uses the Code Database API is the Code Database Browser. It is the tool that lets a developer browse and edit the Lively Kernel source code stored inside a CouchDB database. By giving the user this kind of tool, there is no need to directly interact with the database. Everything that needs to be done – like adding or removing a code object (e.g. a method) and setting its attributes or contents – can be done with the browser. Figure 4 shows the Code Database Browser and names its parts.

To make use of the look and feel of the Lively Kernel browser-style, the Code Database Browser was built on top of already existing classes that are used to show the JavaScript files from the Subversion repository. Using this basic browser

[1] Whenever a CouchDB document is updated the document revision hash (which is a CouchDB identifier) of the previous revision has to be provided. If another revision was added in the meantime, CouchDB can thus detect an update conflict.

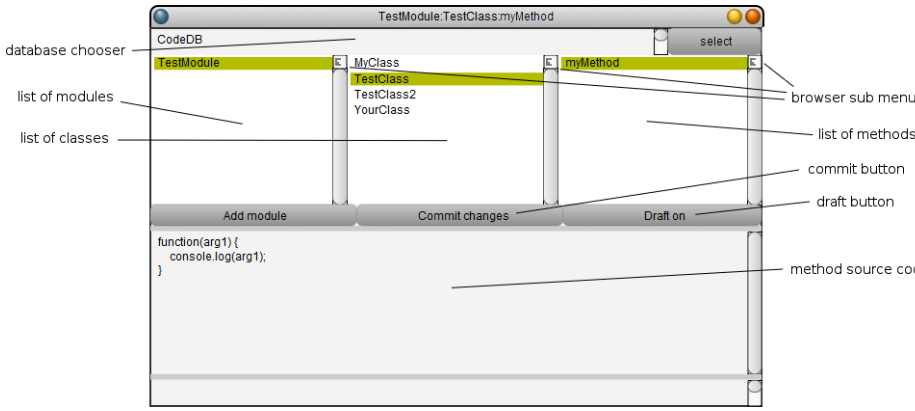

Fig. 4: The Code Database Browser

there is not only a common look and feel but also the same understanding of how to represent and work with each code object. Thus it will be easy to integrate with the default code browser or even replace it in one of the next steps.

A specialty of the Code Database Browser is its differentiation between saving and committing changes. Derived from the two ways code objects can be stored – either as draft or as a commit when semantically grouped – there are also two ways to persist changes. The first one is the default action which is carried out e.g. after changing a method and pressing the keyboard shortcut to save (depending on the operation system that could be CMD + S). This invokes the `save` method on the affected code objects and stores them as drafts inside the database. Having the `draft` status the changes have only been safely persisted but they do not affect the currently loaded and executed code. If the developer wants to activate the changes instead, then the commit button has to be pressed and all the changes done inside the current change set will be committed and activated. Since the developer might not always want to see the changes made as drafts, the draft button switches the browser mode between only displaying activated code and also displaying drafts.

Adding and removing a module, class or method which can be done by the "Add module" button and the browser sub menus automatically creates a draft for the corresponding code object.

2.4 Kernel Extension

In the last two sections we focused on how to manage code objects with the Code Database API and the Code Database Browser. In this section we will explain how the code that is stored inside CouchDB can be loaded into and executed inside the Lively Kernel.

Listing 2.6 shows the structure of a module that can be loaded from the Subversion repository into the Lively Kernel system. The module parameter creates a new module and namespace called `lively.Example` that has dependencies to

`lively.Tools` and `lively.Helper`. Inside that namespace, a class definition creates the new class `SubClass` which is derived from `SuperClass`. The class `SubClass` can have methods like `initialize` and `aMethod` and attributes like `documentation`.

Listing 2.6: Example for the module structure

```
module('lively.Example').requires('lively.Tools', 'lively.Helper').
    toRun(function(example, tools, help) {
  SuperClass.subclass('lively.Example.SubClass', {
    documentation: 'This is a subclass of SuperClass.',

    initialize: function($super) {
      ...
    },

    aMethod: function(arg1, arg2) {
      ...
    },
    ...
  });
  ...
});
```

Our goal was to extend the current core system to load source code from the code database in the same manner. So we introduced a new prefix for all the source code that is loaded from a database. This prefix starts with a $-sign which is followed by the name of the database and a dot. So the prefixed module name for the module of listing 2.6 would look like `$code_db.lively.Example` when loaded from the code database. In this case `code_db` is the database name of a CouchDB database containing Lively Kernel source code.

Furthermore the same style of references can be made inside requirements and when subclassing. This allows a developer of the Lively Kernel to adapt to the new persistency layer more easily.

What is done when the Loader of the Lively Kernel comes across one of the new prefixed module references is that the request of loading the JavaScript file from a URL is modified to not use the current Subversion repository but the CouchDB instance that has been configured. On that CouchDB the selected database is queried for a list (one of CouchDB's querying techniques) that builds a JavaScript file with the same kind of module structure used by the Lively Kernel until today.

3 Discussion

The previous sections introduced the changes we made to the Lively Kernel to make it use a CouchDB database as a (second) persistency layer. Now there shall be a discussion on the decision to use CouchDB as a revision control system (RCS) and the advantages and disadvantage we did discover working on it.

CouchDB itself is one of the new database technologies that arose within the NoSQL movement. Its document-oriented style and the use of JavaScript to define queries (based on the map/reduce algorithm) makes it far more appropriate to store the source code of the Lively Kernel inside it than a relational database. However, its simple key-document storage has also some disadvantages when being used as RCS for object-oriented source code because nowadays every object-oriented programming language has some kind of namespace concept. So the Lively Kernel built upon JavaScript has that too: methods and attributes belong to a class, classes belong to a module and modules have a path-like namespace structure. To support that we had to come up with a mapping of this whole structure to a single key (see section 2.1 for that).

Another important point which was already mentioned is how to manage the revisions. At first we completely wanted to rely on what CouchDB calls a document revision for our code objects. Doing this and storing a code object (e.g. a method) inside only one CouchDB document should lead to code object revisions stored as document revisions. What we did ignore following this approach was that CouchDB revisions are not revisions as in RCSs. Old revisions could for example be removed when compacting a database or omitted making a backup copy to another CouchDB instance. This surely is unwanted when storing source code where one day you might go back in time and restore an old version or use multiple versions of the same file/library in different parts of the system in parallel. So to get fully persistent revisions we ended up with a revisioning system that stores a document for each revision of a code object which was one of two possible ideas [3].

At last we needed a simple and efficient way to reconstruct JavaScript code from the code objects stored inside the CouchDB. This is a point where CouchDB again can show its advantages of document storage. Using map/reduce inside a mixture of views and lists (two of the querying techniques) we were able to construct JavaScript files for modules that look exactly the same as modules that are stored as files inside the Subversion repository. As an alternative to using map/reduce code to construct the JavaScript files, a template-based approach that is also supported by CouchDB could have been taken. But as long as the resulting structure of the JavaScript files is that simple creating a template would just be more overhead. However, in both cases the resulting CouchDB lists can easily be accessed by a URL which by now is the access paradigm for Subversion files too. Therefore no deep changes inside the Lively Kernel had to be done to execute source code that comes out of the CouchDB.

4 Related Work

Like other revision control systems such as ENVY/Developer [4], the Lively Code Database provides the developer with a toolset that is implemented on the core system to help with configuration management and version control. With a similar type of structuring code objects – modules, class and methods – and a browser to develop and maintain these objects the Code Database enables

changes on the method level. Unlike ENVY/Developer our approach does not have component ownership but an author for each revsion.

In contrast to RCS like Subversion [1], GIT [5] or Mercurial [6], our Code Database on top of a CouchDB database does not use files as the finest granularity for source code but instead it breaks it down into modules, classes and methods. All these parts are separatly versioned and kept together by an encapsulating change set. So there are less conflicts if more than one developer is working on the same part of the system.

We are working with revisions similarly to Perforce [7] in terms of letting the server have a database with meta information on the versioned source code (e.g. revision numbers and relations between different revisions) and storing the source code as separate documents. Additionally we have change sets (that are called change lists in Perforce) that group multiple changes on code objects and name the action that is carried out (like `added`, `deleted`, `updated`). However, we have a much finer grained look on code objects as we do not use files to store the source code.

5 Summary and Outlook

With the work done so far, there are three libraries to enable CouchDB as one of the persistency layers that the Lively Kernel can rely on. These three libraries are: the Code Database API which is based on the simple JavaScript API to interact with a CouchDB instance, the Code Database Browser which was created on top of the Code Database API and the small library of core enhancements of the Lively Kernel to make CouchDB databases a valid source to load and execute source code from.

On the CouchDB side there is only one design document that contains all the map/reduce functions to let the core extension and the Code Database API query the database. And this document can easily be installed on a new CouchDB database by simply pointing the Code Database API or the Code Database Browser to the database URL and instructing it to "livelyfy" that database.

To conclude by now there is a transparent replacement of the current persistency from the perspective of the source code and revision management.

Nevertheless there are still some points missing that need further work. One of it is that the entire Lively Kernel source code has to be imported into one database. Doing this there also has to be done some clean up and extension work since the current JavaScript code used inside the Lively Kernel does not only consist of modules, classes and methods/attributes but of some plain JavaScript code to create the base of the system (like the namespaces etc.). Furthermore there are not only JavaScript files that make the Lively Kernel but also XHTML files containing all the elements inside a "world". Either there has to be a way to convert that XHTML files to CouchDB documents too or these files have to stay inside a parallel Subversion repository.

As final ideas of what might be coming next, there still is some work to do on transactions and a better conflict resolution when saving source code to the code database. Also the previously discussed option to use different revisions of the same module or class and the tagging of a revision as version might add a great flexibility to the Lively Kernel.

References

1. Pilato, M.: Version Control With Subversion. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2004)
2. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide Time to Relax. O'Reilly Media, Inc. (2010)
3. Anderson, J.C.: Simple document versioning with couchdb. `http://blog.couch.io/post/632718824/simple-document-versioning-with-couchdb` (2010)
4. Pelrine, J., Knight, A., Cho, A.: Mastering ENVY/Developer. Cambridge University Press, Camebridge, United Kingdom (2001)
5. Loeliger, J.: Version Control with Git. O'Reilly Media, Inc., Sebastopol, CA, USA (2009)
6. O'Sullivan, B.: Mercurial: The Definitive Guide. O'Reilly Media, Inc., Sebastopol, CA, USA (2009)
7. Wingerd, L.: Practical Perforce. O'Reilly Media, Inc., Sebastopol, CA, USA (2006)

A web based GridBagLayout

Seminar Web-based Development Environments

Alexander Hold and Stefan Reichel

Hasso-Plattner-Institut, Potsdam
alexander.hold@student.hpi.uni-potsdam.de
stefan.reichel@student.hpi.uni-potsdam.de

Abstract. Lively Kernel is one of the first web based development environments, which makes the creation of powerful interactive web applications possible. One key challenge is the layouting of graphical elements, which can now be freely dragged, moved and scaled. Our layout manager solves this issue and allows it in effect to create complex arrangements with windows, labels and editor boxes. In this paper a GridBagLayout algorithm is described and the special challenges of a web based development are discussed.

1 Motivation

Modern web based development environments like Lively Kernel provide several tools to create complex graphical elements. These elements are called Morphs in Lively and can be arranged to form complex structures like editor windows, system class browsers etc. Of course you are able to resize and move such windows and their contents, but this creates a new challenge. When the user resizes an editor window for example, he expects, that its text box will also be resized automatically. At the same time the position of the buttons should be altered but the size should still be the same. This complex behavior is normally realized by a layout manager.

Lively Kernel already contains two simple layout managers for horizontal and vertical layouting. Those two layouts can be combined by putting a container morph with the horizontal layout into another one with the vertical layout and the other way arround. Nevertheless it would be difficult to align items inside the main container in both directions. For example if the horizontal layout would be used for the inner container it would be necessary to align the items vertically manually. A layout manager is usefull, which can perform vertical and horizontal layouting at the same time and allows the definition of fixed areas, which will not be resized. In addition to that the manager should be configured in a lively way, which means just by clicking and not by editing large portions of source code. These challenges are covered by the developed GridBagLayout manager.

The remainder of the paper is structured as follows. First of all the layout manager concept is presented, which is followed by the related work. In the second part special concepts and challenges of the layout implementation are discussed. Afterwards the usage of the new layout is shown and finally a conclusion is drawn.

2 The concept

2.1 The layout manager concept

Currently there exist various concepts for layouting, one of the most famous is the GridLayout. The basic idea behind the concept is the division of the container element into equal cells. Every cell gets its own row and column number assigned. To add an element to the container it has to be put into one of those cells. The element will cover the whole space of the cell, which means it has to be extended or shrinked to fit. Often the resulting size is too small for the element e.g. for a text box, therefore it can be spanned over several cells. When the container is resized every cell will be altered, which in result will also effect the elements in the cells. One major disadvantage of this approach is the equal size of every cell because this makes it very inflexible.

This behavior is changed in the GridBagLayout concept, which allows variable heights (for rows) and widths (for columns). Columns and rows can be set to a specific size which will not change when resizing the container. They can also be set to be resizable, which means that its size may change when the container is scaled. By default all resizable rows or columns have the same size, which can be zero if the containers size equals the size of all fixed rows/colums. If all rows or columns have a fixed size, a new resizing row or column will be added to fill the remaining space (which may be zero).

2.2 Related work

The GridBag layout was not invented by computer scientists and despite the several user interaction challenges it is quiet easy to understand in principle. In effect there exist not much related theoretic work in the software industry. The background of the grid layout idea is much older than computers itself. The problem of arranging huge amount of information in form of text and images raised in the printing industry. That is why most of the theoretical work such as the book Making and Breaking the Grid [5] has such a background. Nevertheless those ideas were also adopted in the computer industry and resulted in various different implementations such as Ext JS[1], Nokia QT[3] [4] and JAVA[2].

Almost every bigger framework that contains graphics has some kind of grid layout. For example in Ext JS[1], a large graphic framework for JavaScript, you can add the table to almost every panel. This table layout works much like the GridLayout. In this layout you do not specify where a panel should be placed to. The position is calculated by the column and row spanning of the panels and their order in which they are added to the layout's panel. This layout is pretty useful, but one thing is missing: you cannot edit your layout by using a graphical UI.

In the QT framework[3] [4] , the most used graphics framework for C++, many designers use the "QGridLayout" for layouting their windows and dialogs. The QT designer tool often uses the grid as default layout when you add Widgets into another Widget. This tool was used as blueprint for many of the usability

questions in the Lively GridBagLayout. It always tries to resize dropped Widgets to fit into the layout. That is useful for filling windows completely instead of having empty parts. The new layout for Lively tries to avoid resizing the morphs after dropping, because the user may not like to have his predesigned morphs to be resized automatically. Other parts like the way to handle dropping onto an occupied cell works more like in the QT designer.

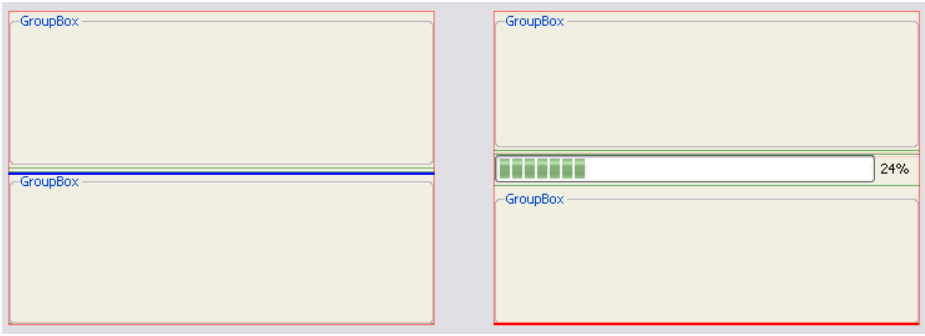

Fig. 1: Dropping a Widget between two others in the QT designer.

In the QT designer the user can drag items from the menu or already existing items into the layout. He can see a graphical effect when dragging, which shows where the Widget will be placed to. When marking the layout object or its parent, the user can set preferences for it in a separate part of the UI. Margins to the borders can be set, spacings between the cell and minimum sizes for the rows and columns. In addition to that a stretching factor can be added for single rows and columns, to modify how their size changes if the layout's size is changed. The Widgets itself can have a minimum and maximum size, too. Size constraint values can be set to define how the layout should react if it is resized.

The designer creates a XML file from the settings you make. The User Interface Compiler uses this file to create corresponding C++ header or source files. You do not need to use the QT designer to create windows or dialogs. The designer is just a tool to simplify the UI creation. Once the header or source files are created, they can be used in the C++ project.

Every setting which could be changed in the designer can also be changed at runtime. For example Widgets can be added, margins, spacings or the Geometry of the layout can be set. The layout provides information about the current settings like the position of a specific cell, the count of rows or the minimum width of a column.

The GridBagLayout concept is also implemented in Java **??** and provides similar methods. The key of the this approach is the so called "GridConstaint", which must be supplied whenever a new item is added. This constraint object contains information about the current cell position, spanning, padding and also its margins. It also determines the behavior if the added element is smaller than

the corresponding cell e.g. after resizing of its container. In that case it can be specified whether the element should be stretched horizontally, vertically or both. If the element is not resized in both directions the developer can align the element for example to the bottom or top. Those “GridConstraints” can be altered at runtime to provide maximal flexibility.

```
GridBagConstraints c = new GridBagConstraints();
c.fill = GridBagConstraints.HORIZONTAL;
c.gridx = 0;
c.gridy = 0;

JButton button = new JButton("Button 1");
container.add(button, c);
```

Listing 3.1: Adding a new button to a container with GridBagLayout

Listing 3.1 shows how a button is added to a container element which uses the GridBagLayout. The button will be added to the first column(“gridx”) and first row(“gridy”). If the container is resized the button will only be scaled horizontally.

3 The layout manager implementation

3.1 The dropping concept

A main purpose was to implement a grid layout that can be build easily by using drag and drop. For this we used the functionality of lively which already detects where the dragged morph was dropped. All inserted morphs start with a column and row spanning of one. If a morph is inserted at an empty row or column, this one is set to the morphs size and set to fixed. The size of a row or column is increased to the size of a new morph, if it is larger than the row’s or column’s size before the drop.

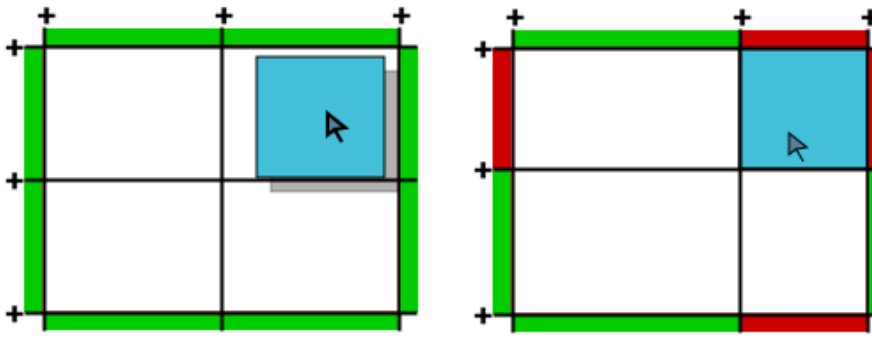

Fig. 2: If the row and column are empty they use the morphs size after dropping.

If a morph was dropped onto the container morph, a function is called which determines to which cell the morph should be added to. As dropping point the center of the dropped morph is used. The algorithm begins by retrieving the cell to which the point belongs to. If this cell is empty, the morph can be added here. Otherwise the morph will be added to a new row or column.

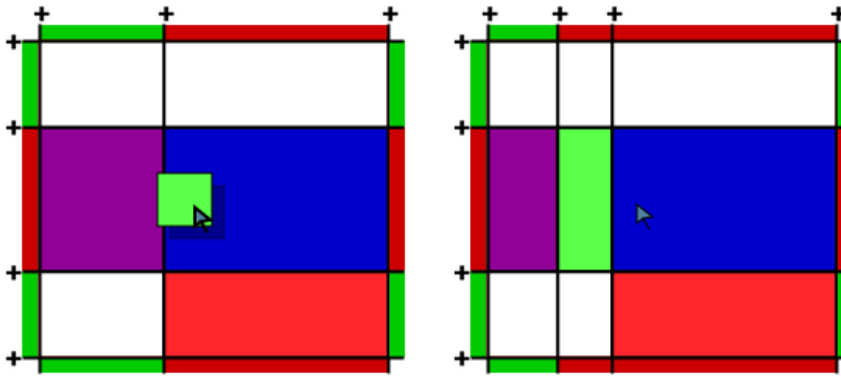

Fig. 3: Dropping a Morph onto an occupied cell creates new cells.

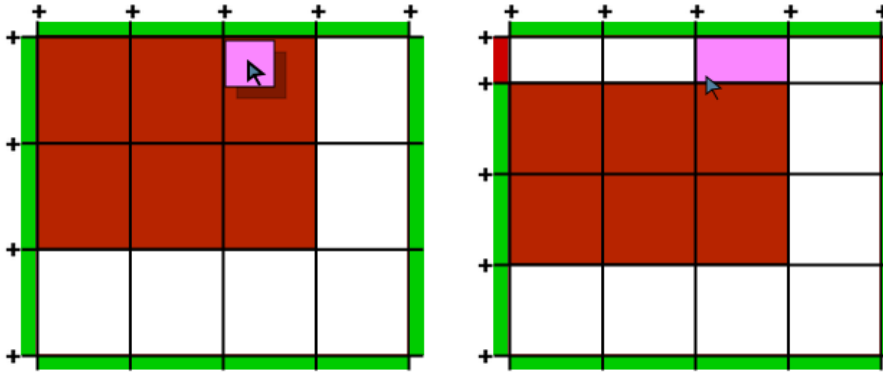

Fig. 4: The morph is inserted into the column where it was dropped in.

The dropping point's distance to the four borders of the occupied cell's morph (target morph) are compared. The distanced are normalized by height/width of the target morph which is necessary if it has a big difference between its height and width. Next to the nearest border a new row/column is inserted, in which

the morph will be placed. A dropped morph will always be placed in the same row (column) as dropped, if the nearest border is the left or right (top or bottom) one.

3.2 User interaction for grid resizing

The user can toggle to show or hide the borders of all rows and columns (GridLines). These GridLines protrude from the container. This part of the line can be used to resize a row or column by dragging. Dragging a line between two resizable rows will not change anything. Changing the position of an outer line may change the containers size, but not to a width (height) less than the total size of all fixed columns (rows).

When displaying the GridLines, additional buttons are shown at the sides for every row and column. Red buttons mean that this row or column has a fixed size and green buttons are next to resizing rows and columns. Pushing one of these buttons toggles the corresponding row or column resizing policy (fixed / flexible).

At the top or left of a GridLine a plus sign is shown. Clicking that sign adds a resizing row or column at this position.

3.3 Spanning of Morphs

A morph can span multiple rows and columns at once. The user can drag the GridLine to the right or bottom of a morph to change this spanning. The morph will use the complete area of all rows and columns it lays in. Increasing the spanning is prohibited, if the new spanning includes an already occupied cell. If a new row or column is inserted in the middle of a morph, the morph's spanning will increase.

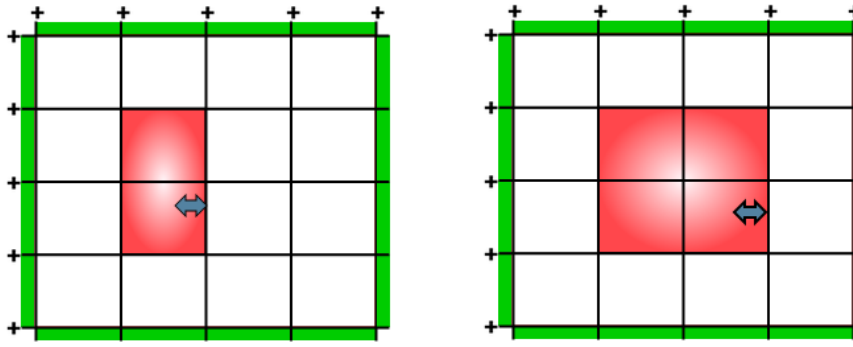

Fig. 5: Changing the spanning of a morph.

The implementation of spanning allows the user more flexible designs. A morph can now be in a flexible and a fixed row at once, which guarantees a minimum height. The design in figure 6 would not be possible without spanning.

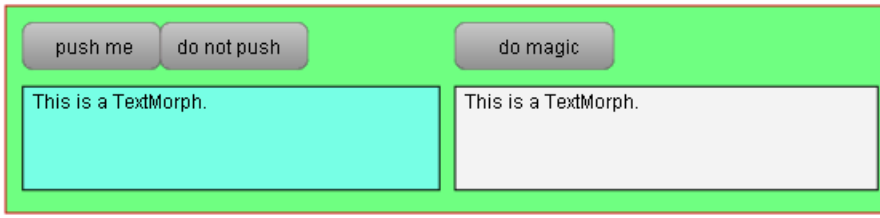

Fig. 6: This would not be possible without spanning.

3.4 Context based menus

The user interaction is mainly done via GridLines, but also other interaction mechanism were needed. A well known concept is the usage of context menus. This makes it possible to perform a right click on a morph and to select an option to hide the grid or to remove the selected column. Every morph has already a predefined context menu, which is inherited from the world morph. Nevertheless for our purpose it has to be altered. The easiest way to do this, is to overwrite the "morphMenu" method, which is used to create the menu . This idea has a major disadvantage, the inherited world morph context menu and its items are lost, so they wont be shown when right clicking on the altered morph. A real solution should combine the new items, for example "Remove column", and the old items inherited from the world morph.

This is done by context based programming in Javascript, in Lively it is called ContextJS. At first a so called "layer" has to be created. In this layer the class of the object, which should be altered is specified. In the above mentioned use case this is the Morph class itself. In a second step the methods of the class can be overwritten. In comparison to the first approach, this time ContextJS provides a reference to the original implementation. In effect the old implementation can be called and afterwards user defined additional code. To alter the context menu the old morphMenu method is called, this will create a menu with all world morph items. Afterwards additional items like "Remove column" are added to the resulting menu, as shown in listing 3.2.

```
layerClass(GridBagLayer, Morph, {

            morphMenu: function ($proceed, evt) {
                var menu = $proceed(evt);
                menu.addItem(["Remove column", this.removeCol]);
```

```
            return menu;
        },
};
```

Listing 3.2: Altering morph menus with ContextJS

The layer and its effects are by default turned off. To activate it for example after the Morph was added to a container with a GridBagLayout manager, the Morph has to be just informed, that the layer is active by calling its "setWithLayers" method. This new context is not only active for the Morph itself but also for any of its children. An extract of the code which was used to perform this task can be found in listing 3.3.

```
beforeAddMorph: function (supermorph, morph, isFront) {
        ....
    morph.setWithLayers([GridBagLayer]);
}
```

Listing 3.3: Activating the ContextJS layer

3.5 Saving a layout

In comparison to normal interactive web sites, the Lively development environment has special challenges to master. One of them is saving of content. Interactive web sites also tend to save information like data entered in a web form. Nevertheless within the scope of web based IDEs this challenge gets a new quality. It is not only necessary to save simple text, but also complex data structures, in Lively a lot of those structures exist.

Saving itself is not the easiest task to perform. Nevertheless the perhaps most complex task is the loading and recovering of the saved data back to working JavaScript code.

For the first part of the saving challenge, Lively Kernel uses the JavaScript Object Notation[1] technology. JSON is a very lightweight data interchange format, which was developed with the goal to create an easy to save and machine readable format. This data representation is itself valid JavaScript code, which can be executed by calling the JavaScript method "eval". Today various implementations for common used programming languages exist, one of them is JavaScript. By using JSON and its ability to save data structures like lists and dictionaries.

In that way Lively is able to save for example parts of the ContextJS layers and later on to unserialize and in effect to recover them. Every Object can implement a method called "toLiteral" which controls the serialization and method "fromLiteral" for the unserialization. This works for Layers out of the box, nevertheless for the developed layout manager, those methods had to be overwritten.

One of the main concepts of the layout implementation is to store adminstration data mainly in the manager and not in its controlled Morphs. Therefore saving and recovering of the layout manager is essential, this is done in two

[1] http://www.json.org/

steps. At first the general column and row data are stored, these are mainly sizes and constrains. The saving and recovering are done with the above mentioned "toLiteral" and "fromLiteral" methods.

```
"rows":"[{\"size\":30,\"fixed\":true,\"baseSize\":0,\"notEmpty\":true
    },{\"width\":0,\"fixed\":false,\"baseSize\":0,\"size\":120}]"
```

Listing 3.4: JSON representation of the saved row information of the GridBagLayout

The second step is performed after the basic layout manager and its Morphs have been restored. At this time the manager knows the size and some additional information of the cells, but not which (container-) Morph is assigned to it. This relation is recreated by asking the container Morph for its children, which will be assigned to a column and row. The crucial point of the method is the container itself, which is not available after the first step. Normally a layout manager in Lively does by design not know its container, this information is later supplied by special user triggered events, for example when a new child morph was added. Those events are also used by the developed GridBagLayout so in effect the user will have a working layout at every time.

4 Using the layout

For using the GridBagLayoutManager it is need to include the module *GridBagLayout*. To add the layout to a morph use the following line (*aMorph* is the the morph you want apply the layout to):

```
aMorph.layoutManager = new GridBagLayoutManager(aMorph);
```

By default the helping morphs and lines described in section 3.2 are visible, but can be hidden by using the point “Hide grid” of the morph’s menu.

The GridBagLayoutManager is designed for creating and editing a layout in the UI of Lively. How to use the UI functionality is described in the sections 3.1 to 3.4. Nevertheless it is possible to create a design automatically by using JavaScript. Every needed method is directly invoked in the layout manager, a list of all methods for creating and editing a grid is shown in the tables 1 to 3.

The layout manager API can be divided into three main parts. First of all there are methods to assign a morph to a cell and to change its spanning.

`moveMorphToCell (aMorph, x, y)`	Moves a Morph to the specified cell if it exists and is not empty.
`setRowSpanForMorph (aMorph, span)`	Sets a Morph’s row spanning.
`setColSpanForMorph (aMorph, span)`	Sets a Morph’s column spanning.

Table 1: Methods for a morph of the grid layout manager

The second part of the API is represented by several getters, which help the developer to get a morph based on its position in the grid. If no morph is present at the supplied position the undefined value will be returned.

`getMorphAt (x, y)`	Return the Morph contained in this cell.
`getMorphsInColumn (index)`	Returns the Morphs of this column.
`getMorphsInRow (index)`	Returns the Morphs of this row.

Table 2: Methods to obtain a Morph

The last part of the programming interface is used to control the size and resizing mode of cells. If a row or column is set to a fixed mode it won't be resized, instead it will keep its size. The following API can also be used to add new rows and columns. By adding rows and columns with a small width or height and a fixed size policy it is possible to create padding between cells.

`addRowAt (index)`	Adds a row at the specified index.
`addColAt (index)`	Adds a column at the specified index.
`removeRow (index)`	Removes a row if it is empty.
`removeCol (index)`	Removes a column if it is empty.
`setRowFixed (index, value)`	Sets a row to fixed as default or to resizing if value is false.
`setColFixed (index, value)`	Sets a column to fixed as default or to resizing if value is false.
`toggleRowFixed (index)`	Toggles a row between resizable and fixed.
`toggleColFixed (index)`	Toggles a row between resizable and fixed.
`setRowSize (index, size)`	Sets a row to a fixed size.
`setColSize (index, size)`	Sets a column to a fixed size.

Table 3: Methods for row and column access

5 Conclusion

Developing for and in a web based environment creates a lot of challenges. One of them is now solved. The GridBagLayout of Lively Kernel provides the user the ability to create complex arrangements of morphs, which can together act as one. Whether they are scaled or moved, whether they are put in horizontal or vertical order, the layout will always guarantee a proper position and size. At the same time the administration of such a complex piece of software is kept simple and can be controlled by a few lines of code. In addition to that a user interface was created lively, which means to change cell size, mode and control morphs with just a click.

Although the developed layout manager is for most of the users an invisible component in the background, it enriches their Lively Kernel experience every day.

References

1. Extjs layout tutorial (August 2010), http://www.sencha.com/learn/Tutorial:Using_Layouts_with_Ext_-_Part_1
2. Java gridlayout tutorial (August 2010), http://download.oracle.com/javase/tutorial/uiswing/layout/grid.html
3. Nokia qt official website (August 2010), http://qt.nokia.com/
4. Qt layout tutorial (August 2010), http://doc.qt.nokia.com/4.6/layout.html
5. Samara, T.: Making and Breaking the Grid: A Graphic Design Layout Workshop. Rockport Publishers (2003)

Collaboration in Lively

Seminar Web-based Development Environments

Fabian Garagnon and Kai Schlichting

Hasso-Plattner-Institut, Potsdam
{fabian.garagnon,kai.schlichting}@student.hpi.uni-potsdam.de

Abstract. Wiki pages and developer journals are the home of dynamic and frequently changing content. Such applications written with Lively Kernel lack of the possibility that several users can edit different parts of a page at the same time.
We introduce a collaboration framework for Lively that allow users to see other users' changes in real-time. This report discusses a command-based approach that addresses this topic and shows one concrete implementation for this problem.

1 Introduction

Typical web activities like editing of a web page by several users at the same time are editing Wiki pages, writing papers, creating presentations and others. Those applications are made possible by Lively Kernel out-of-the-box since it allows changing the interface (and its behavior) of a web page in the browser. When saving a page, the whole DOM structure is saved into a subversion repository whereby changes made by others simultaneously get lost.

Collaboration tools enables several people working on the same thing in real-time. In Lively, such a framework could make possible that two users are writing a text together and a third user is changing the design on the same page. This way, changes can be applied in parallel without having to wait for other users finishing their work. Additionally, ideas can be shared immediately and other users can give feedback directly.

The main goal of this project is to solve frequent synchronization issues in a multi-user environment like the *Lively Kernel Webwerkstatt Wiki*[1]. This synchronization should be in nearly real-time so that every user can see the others working on a Wiki page. Normal activities like adding of new morphs, change position, resizing or editing text is synchronized. There should be no spontaneously changes (e.g. a morphing "jumping" from one place to another), therefore users see other mouse cursors and know in which part of the page the others are working.

Although no direct goal of this project, the resulting framework should be extensible by more advanced collaboration features: Conflict management won't

[1] http://lively-kernel.org/repository/webwerkstatt/webwerkstatt.xhtml

be available in the first step, but the solution should allow to use operational transformations [5] (which is an optimistic approach) to solve conflicts effective. Another neat but not as necessary feature would be a time-slider which gives each user the opportunity to travel back in time and view the Wiki page a few clicks behind (as seen in EtherPad[2] [1]).

The remainder of this paper is structured as follows: The next section introduces our command-based collaboration approach. Section 3 presents the framework design and architecture, while section 4 discusses special implementation considerations. Finally, section 5 concludes and points to future work.

2 Approach

This section explains our approach for implementing basic collaboration features into the Lively kernel. For this purpose a framework is needed that allows synchronizing changes of one user with others.

A first question that have to be discussed is the level on which changes should be observed and recorded. In browser environments, there are at least three potentially suitable levels: (a) Events from input devices (mouse, keyboard), (b) changes in DOM structure or (c) calls to the JavaScript API. Propagating mouse and keyboard events from one users to another (a) has the drawback that button clicks with non-local side effects (e.g. deleting a record on a server) would be called twice. Synchronizing DOM changes like a newly added SVG node (b) would create a common view on the currently opened page, but doesn't execute related JavaScript code: A new button is visible to all users, but no action handlers is assigned to the button since this requires JavaScript. The most promising solution to our problem is to listen to basic Lively API calls (c) (like `TextMorph#setTextString`, `Morph#setPosition` or `Morph#addMorph`) and propagate them to other clients. This way, the JavaScript object state keeps synchronized while the propagated method calls care about updating the underlying DOM.

To record the method calls made to the Lively API, we decided for a command-based interaction (Figure 1): When a method is called, a command is created locally in the client encapsulating all the information needed to execute and undo the method later on other clients. This information includes particularly the arguments of the called function and the current state for undoing (e.g. a call to `Morph#setPosition` saves the new and the old position). After having created the command, it is executed locally to ensure fast feedback to the user (optimistic approach). Then the command object is serialized and send to the server so that other clients get informed about the change. The server assigns an ascending id to the command and puts it at the end of the global command queue which is solely managed on server side. The id of the command defines the order of the command queue, and clients have to assure that commands are executed in the given order. Finally, the command is sent to all clients, who arranges it in their local command queue and execute it according to the command's id. The client

[2] http://etherpad.com/

which created the command in the first place doesn't execute the command a second time, but the assigned id have to be checked with the order in its local command queue. If other commands have been received in the meantime, the command has to be undone and wait its turn.

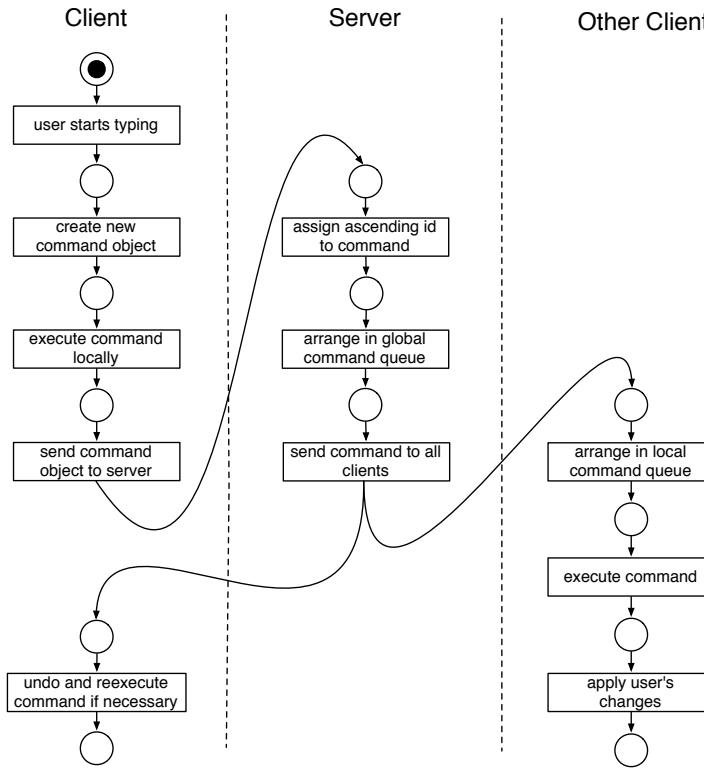

Fig. 1: Command life cycle using the example of an user who is typing text

During a collaboration session, all incidental commands are stored on the server side. As soon as a user presses Lively's built-in save button, a new milestone is created (which just references current command's id). When a new user joins an existing session, he receives a unique user id and a list of all commands that have been created since the last milestone. To implement a playback/ time-slider feature as mentioned in the introduction, a user could switch between milestones (and the accordant commands belonging to this milestones) and see all visual changes that have been made in the meantime.

3 Architecture

This chapter describes the architecture of the collaboration framework for the Lively kernel in detail. Figure 2 shows the overall architecture: A Lively application is embedded in the web environment and is therefore a client server architecture. On the client side, a `Command Manager` executes or undoes commands and queues new commands in the `Local Command Queue`. When the client creates a new command, it is serialized and send to the server as explained in section 2. On the server side, a web server is listening for new client connections and commands at which the latter are stored in the `Global Command Queue`.

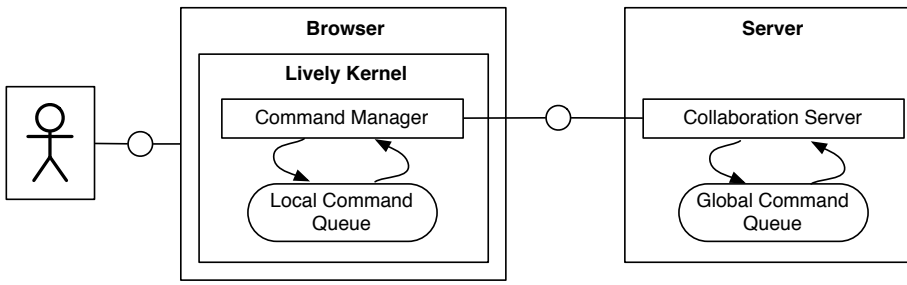

Fig. 2: High-level architecture of collaboration framework

Next subsections will explain the client and server architecture in more detail (see figure 3).

3.1 Client Architecture

The client intercepts some core Lively morph methods with *ContextJS* to record the user's (visual) changes (see table 1). These methods are grouped into three *ContextJS layers*, which can be separately switched on and off. Since some functionality (especially the `onMouseMove`) can produce much synchronization overload, this might be helpful in situations with slow internet connection.. The `Command Manager` queues the commands and sends them via the `Connection` singleton instance to the server.

The `Connection` singleton holds a persistent connection to the server which is established as soon as a client opens a collaboration-enabled Lively page. This connection is realized through the emerging WebSocket [3] technology, which allows bidirectional communication with a smaller footprint. Thus, pushing messages (in our case primarily commands) from server to clients and the other way round is easily possible without much overhead - it's much like using sockets in desktop programming.

Layer	Class	Method
CollabMorphLayer	Morph	rotateBy
		translateBy
		setPosition
		...
CollabMouseMove	Morph	onMouseMove
CollabTextMorphLayer	TextMorph	setTextString

Table 1: Overview of intercepted Lively methods

3.2 Server Architecture

The architecture of the server is shown on the right side of figure 3. The server is split into two main parts, the `Collaboration Server` acting as the web server and the `Redis Server` representing the database. The `Collaboration Server` is written in Node.js[3], an event-driven I/O framework that enables writing server-side JavaScript. Redis[4] is an in-memory key-value store that allows advanced data types as values (e.g. strings, lists, sets, ...) and can persist its data to the disk in configurable intervals. It perfectly fits in the collaboration use case since it provides fast access to the data while being semi-persistent so that most of the data can be restored when the server crashes.

The `Collaboration Server` accepts *WebSocket* connections and commands from clients. Every command is saved into the Redis database to enable the playback of commands and inform new clients about the stored commands. Table 2 gives an example snapshot of the Redis database to show which data is stored. Every key is scoped to the user's page URL so that commands of different pages aren't merged.

The `Collaboration Server` uses the *publish/ subscribe* mechanism (based on channels) of Redis to send new commands to every client. Three channels are exposed by the `Collaboration Server`: *commands*, *milestones* and *mouse*. All channels, except for the mouse cursor, are saved semi-persistent into the Redis key-value store. Mouse cursor events are directly propagated to all clients since they are only of interest in a short time slice. Thus, the Server subscribes to the *command* channel in Redis with a callback, which is called every time the server publishes a new command from a client. This callback function sends then the published command to every client.

[3] http://nodejs.org/
[4] http://code.google.com/p/Redis/

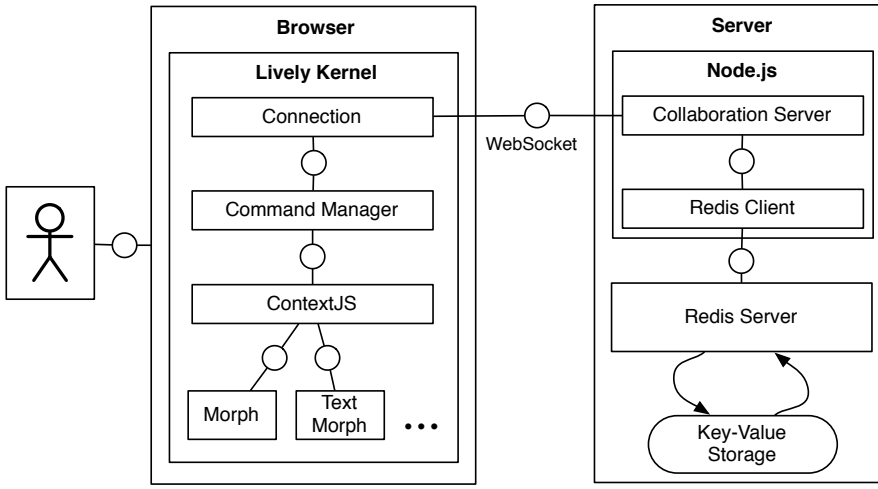

Fig. 3: Detailed client and server architecture of collaboration framework

4 Implementation Considerations

4.1 Globally unique IDs

When a user updates a morph on his page, the changes have to be sent to the accordant morph on the other users' pages. Therefore, globally unique IDs are assigned to all new morphs so that a morph can be unambiguously identified on each page. We extended the ID generation in Lively to avoid conflicts (i.e. same ID generated on different clients for different morphs) by scoping all IDs to the current user: Instead of generating IDs like `1234:Morph`, `lively.data.Wrapper#setId` add the user ID resulting in `user541234:Morph`[5].

4.2 Object (de)serialization

As described in section 3.1, we are intercepting relevant morph methods to listen to changes. Arguments are saved in command objects that have to be serialized when sending to the server. Especially when it comes to serializing whole morphs, i.e. for `Morph#addMorph` synchronization, a smarter serialization approach is necessary to handle circular references and already existing objects. [2] provides a Relaxer and Restorer class for (de)serialize Lively morphs that could be easily adopted for our use case: When adding a new morph (which might contain sub morphs, for example), all references to other morphs are only saved by their IDs so that a quite flat JSON structure is generated. Other clients can then deserialize this structure and restore all previous relationships.

[5] a colon as separator couldn't be used here since some existing code relied on the class name after the first colon

Key	Example Value
<url>//milestones	["140", "234"]
<url>//commands-current-id	"336"
<url>//commands	[{ id:336", commandType:"SetPositionCommand", morphId:"user_0366079:Morph", val:{x:10, y:10}, oldVal:{x:5, y:5}, timestamp:"1280353890466", userId:"user_0" }]
<url>//users	["user_1", "user_2"]

Table 2: Exemplary snapshot of Redis key-value database

4.3 WebSockets

Websockets are a fully bidirectional and slim protocol from the upcoming *HTML5* standard. The decision for *Websockets* as the protocol between the client and the server was taken, because of the need for nearly realtime bidirectional communication. There is a japanese website[6] on which everyone can test the speed difference between *Websockets* and XML HTTP requests (*Ajax*). With the latest *Google Chrome* browser the benchmark of the website shows that *WebSockets* are nearly 40 times faster than the *Ajax* requests.

5 Summary & Outlook

Some special events are intercepted by the use of *ContextJS*. These events are transformed into a command and send through *WebSockets* to the server, which saves the commands in the key/value store *Redis*. Via this mechanism nearly every Wiki page can be made collaborative.

At this stage of the project users can collaboratively work on a wikipage and add new morphs or drag them around, or even complex actions like adding new journal entries. All these actions are synchronized between the server and all users. Each user can even see the mouse cursors of the other users.

General Outlook In the future there will be a client side time slider, which enables each user to travel back in time, via undoing the saved commands. The user name next to each users mouse will be the login name and not an anonymously generated one. Near the time slider there can be a button or checkbox to easily switch the collaboration on and off. These two user interaction elements can be grouped to a preference pane. If one user saves the Wiki page, this could be

[6] http://bloga.jp/ws/jq/wakachi/mecab/wakachi.html

intercepted and every command saved on the server before this event can be deleted.

In the future there have to be also a feature to suppress the synchronization of some events. At the moment the menus from every user are synchronized to every other user, which is not the preferred behavior.

Conflict Management The conflict management could be extended to Operational Transformations, because in our solution it depends on the action taken by the users, which user will win. For example, if two users are positioning the same morph, the last command that arrives at the server will win. This is because the position of a morph is absolute and so the last value will be used. An example in which the first user wins is, if one user deletes a morph before another can change the size of that morph. The deletion will invalidate the resizing command of that morph.

Consecutive projects which will implement a better conflict management are welcome.

Clock synchronization Events which are not originated by a user action, like the tick of a `ClockMorph` are produced by every client nearly at the same time. These multiple events could also collide with each other, if the users are in different time zones. There are multiple solutions to solve these kind of conflicts, one solution is that only one client produces these special events. But the problem with different time zones is not considered with this solution. Not synchronizing these kind of events would be another result to this problem.

References

1. Etherpad time-slider. http://www.youtube.com/watch?v=Endvb81oz80 (2009)
2. Dannert, J.: WebCards - Entwurf und Implementierung eines kollaborativen, graphischen Web-Entwicklungssystems für Endanwender (2009)
3. Hickson, I.: The websocket api. W3C Working Draft 22 December 2009 (2010)
4. Nichols, D.A., Curtis, P., Dixon, M., Lamping, J.: High-latency, low-bandwidth windowing in the jupiter collaboration system. In: UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology (1995)
5. Wang, D., Mah, A., Lassen, S.: Google wave operational transformation. http://wave-protocol.googlecode.com/hg/whitepapers/operational-transform/operational-transform.html (2010)

Bringing TEX's Paragraph Layout Algorithm to the Lively Kernel

Seminar Web-based Development Environments

Tobias Pape

Hasso-Plattner-Institut, Potsdam
{firstname.lastname}@student.hpi.uni-potsdam.de

Abstract. The Lively Kernel as a web-based system uses a simple line breaking algorithm to lay out its text. This class of algorithms is prone to unpleasant output. More sophisticated algorithms exist, e. g., as used in TEX, but are harder to understand. Visualizing the algorithm in a *lively* manner may facilitate its understanding. Integrating the algorithm into the Lively Kernel provides the basis for *lively* visualizing it.

1 Why TEX? A Motivation

The Lively Kernel is a web-based development- and runtime-environment that is self-sustaining and focused on collaboration [4]. Its user interface is an implementation of the Morphic system [7, 8]. The notion of a morph conveys the principle of objects that know how to represent themselves graphically. This also holds for text to be used and displayed in the Lively Kernel.

However, like in most web-based systems, the text layout algorithm used is a rather simple, straight-forward one. Under many circumstances the output is undesirable; unfortunate combinations of words used in the text and the width of the text to be created may lead to visual holes in the text. By the nature of this class of algorithms, avoiding such cases is hard [6]. However, more sophisticated algorithms exist that migitate this problem. E. g., the algorithm used by the TEX typesetting system treats paragraphs as whole entities and avoids jagged lines most of the time. On the contrary, such algorithms are more complex than the straight-forward ones and often are harder to understand.

To make complex algorithms more conceivable, algorithm visualization may be employed. Yet, most visualization techniques do not cope for the *process* nature of algorithms and try to visualize dynamic facts statically, e. g., with flow charts [9]. With the Lively Kernel, however a system exists that provides means to build more *lively* applications, visualizations, in this case. That said, the most natural way to convey facts or processes in the Lively Kernel is to actually implement them in the Lively Kernel and present them interactively. Thus, the aim of this work is to provide the Lively Kernel an implementation of TEX paragraph layout algorithm and then present its mechanism using the algorithm itself.

1.1 Notes on Synonyms

In order to describe paragraph layouting or, more precise, line breaking, two viewpoints exist that may introduce confusion. A *line* of text can either denote a *physical* line, i. e., a line with a certain physical width, a product of line breaking; or a *logical* line, i. e., a conceptual line of text to be broken into parts. Conversely, two viewpoints on *paragraphs* exist: the *physical* paragraph is a collection of physical lines—line breaking already has been applied; the *logical* paragraph correlates with the *logical* line: a text string not yet broken into physical lines.

To avoid confusion, the word "line" will denote the *physical* line unless stated otherwise.

Paper Organization

Following this section, section 2 covers the concept of paragraph layouting via line breaking with special respect to the algorithms of the Lively Kernel and TEX. Then in section 3, the base implementation of the newly used algorithm is described as well as its integration into the Lively Kernel. Then, in section 4 related work is given followed by a summary and an outlook in section 5.

2 Laying Out Paragraphs, Conceptually

The process of paragraph layouting mainly comprises line breaking, i. e., at which points in a logical paragraph to introduce the start of a new physical line.[1] As pointed out earlier, different approaches exists to achieve this. In the following, first the current approach used in the Lively Kernel is described. Then, the approach used by TEX, the Knuth-Plass algorithm (KPA), is outlined briefly.
N.B.: Like most line breaking algorithms, both algorithms use the *text width* or *composition width* as dominant restriction: no single line determined by the algorithm shall be wider than the composition width specified by the user.

2.1 Text in the Lively Kernel

The main facility to show text in the Lively Kernel is the `TextMorph`. This morph is a simple rectangular morph that holds a string[2] to be rendered. Upon rendering of the morph, line breaking is carried out by walking all *text words* of the string and collecting them in a *text line*. The walking mechanism is supported by a so called *chunk stream* that yields consecutive text words.

Text words Lively Kernel text words are wrappers for non-space text strings that can be rendered to a Lively Kernel canvas, e. g., SVG. Text words are created lazily by a chunk stream during parsing of the input string. Also, text words

[1] This has changed during the introduction of computers into typesetting, where the former predominant task of *spacing out* has diminished in importance [6].
[2] Text morphs may hold rich text, however, this is not covered here.

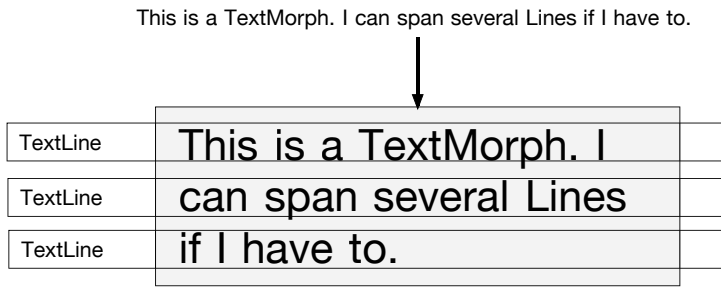

Fig. 1: Text rendering in the Lively Kernel: A string is split into text words that are arranged in text lines of a maximum width.

are responsible for determining their own width by querying the Lively Kernel font for the width of the word's characters. A text word has an associated 'raw' element that can be directly drawn by the underlying rendering mechanism, e. g., SVG as above. As an example, for the string in Figure 1 13 text words would be created, each knowing about its width and an associcated 'raw' SVG element.

The chunk stream For line breaking, the text words of a string belonging to a text morph are processed sequentially. To do that the text string has to be broken up into text words, and these have to be returned one after another. The operations are carried out by the chunk stream:

1. Collect characters form the current position in the text string until
2. a space-denoting character is read.
3. Then form a text word form the collected characters,
4. determine its width and
5. return the text word.
6. Store the new position in the text string for the next iteration.

Text lines and line breaking Just before rendering the text morph, the aforementioned chunk stream is processed and text lines are formed. Each text line is responsible for the rendering of the text word it contains.

The actual line breaking happens when the text lines are to be filled with text words. As mentioned above, a certain limit for the text lines width exists: the composition width. A text line is formed through the following process:

1. The chunk stream is queried for the next text word.
2. The new word's width is added to the current line width.
3. If the composition width is overshot, end the line and put the word at the beginning of a new line;
4. If not, add the text word to the current line and add the width for a space.
5. Do this until the chunk stream is empty.

Note that the process listed under *Chunk stream* is actually performed here during the first step, hence, this sequence essentially makes up the arrow in Figure 1.

After these steps, a text morph contains an array of text lines comprising renderable text words. The algorithm's straight-forward manner accounts for its *first fit* classification, according to Knuth [5, p. 78]. This contrasts the TEX paragraph layout algorithm which is described next.

2.2 The Knuth-Plass algorithm

The TEX typesetting system employs an algorithm for line breaking conceived by Knuth and Plass in 1982 [6] which treats the paragraph it operates on as a whole. Like the Lively Kernel line breaking algorithm, the KPA does not operate directly on text strings but *nodes* which have similarities to Lively Kernel text words. However, unlike in Lively Kernel the KPA nodes may stand for more than *text not containing white spaces*. These nodes are generated by the TEX language parser, yet, this process is not of interest here.

The nodes that resemble Lively Kernel text words best are *boxes*; they have a certain width and *material*, mostly pure text content. The implicit spacing in the Lively Kernel line breaking process contrasts with explicit nodes that carry space information in the KPA: *glues* have a certain width as well as information in how far this width may changed (called *shrink- and stretchability*). Last, an invisible node in the KPA has no correspondant in the Lively Kernel: the *penalty* node conveys the additional cost of a line break at a certain point in the text string.

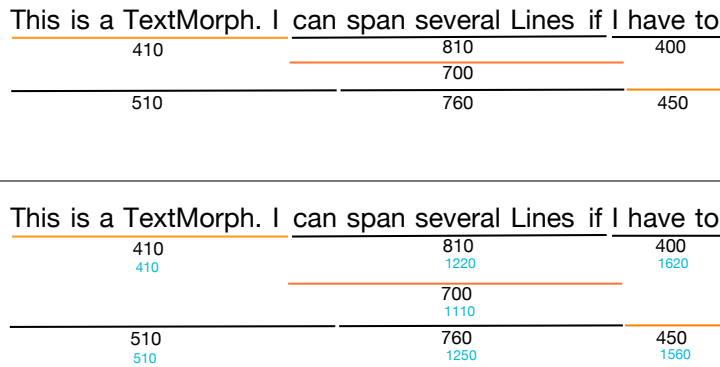

Fig. 2: Simplified intermediate result of the KPA. Above: Five possible breakpoints were found. The cost of the lines each breakpoint creates is given. Below: Additional to the previous, the *accumulated cost* from the beginning of the paragraph is given. The highlit sequence denotes the sequence of chosen breakpoints.

Line breaking with the KPA Given a list of pre-arranged nodes, the KPA tries to find the best sequence of line breaks by maximizing the 'pleasantness' of the line breaks. To achieve this, the KPA uses a quantification of how bad a line break may look: the *badness*. Considering the upper part of Figure 2, there are two possible breakpoints for the line that would start with "can span several. . . ;" one before and one right after the "I." However, the former breakpoint would cause the white space in that line so much to be shrunken to fit the desired width that a *badness value* of 810 is allocated to it. For the latter one, not so much white space has to be shrunken, hence, the value is lower: 700.

To generalize, given two possible breakpoints in a text string and a composition width. Then the *badness of a line* formed by these breakpoints is obtained from the ratio all the white space in that line has to be shrunken or stretched to make the line width equal to the composition width. For more in-depth badness calculation formulæ and how the value of penalty nodes contribute to it, refer to the original algorithm paper [6].

The main method of the KPA is to (a) find all important *possible* line breaks, (b) calculate their badness and, (c) choose that sequence of breakpoints among the possible ones that has the least overall badness values. To ease this process, the KPA employs a mathematical technique called *dynamic programming* [2]. Essentially, this conveys the constant accumulation of badness values from the start of the paragraph up to the current possible breakpoint. In Figure 2 these accumulated values are given in the lower part beneath the individual badness values. Once these values and their accumulation have been calculated, it is possible to *walk back* the path of least value. In the example, this means starting from the end of the paragraph and choosing the path to the breakpoints before or after the "I." As the breakpoint after the "I" has an accumulated value of 1560, which is less than 1620 for the other one, the breakpoint before the "I" would be chosen. This continues until the start of the paragraph is reached. The found sequence of breakpoints now results in actual lines and line breaks.

⁂

Comparing the Lively Kernel- and the KPA-approach, it is evident that the former is a *line centric*, the latter a *paragraph centric* algorithm. Additionally, the latter seems to inhibit more complexity and, hence, may need more time to be understood and to be consequently implemented.

3 Implementation in the Lively Kernel

The KPA is more complex than the default Lively Kernel line breaking algorithm and visualizing it may ease understanding it. The Lively Kernel provides a programming environment that fosters interactivity and *liveliness*. However, to provide these attributes to a visualization of the KPA, the algorithm itself has to be implemented in the Lively Kernel. As the base language in the Lively Kernel is JavaScript, the TypeSet implementation of the KPA has used as foundation; it is described in the following. Then, its integration into the Lively Kernel and the actual visualization of the KPA are shown.

3.1 The KPA in the Web: TypeSet

TypeSet by Bram Stein is an application of the KPA to the HTML5 canvas using JavaScript [10]. It is intended as a first step towards a library of various line breaking algorithms for the HTML5 canvas element. The implementation covers the whole algorithm as in [6], nevertheless, the simplified version mentioned there was not used and the as well mentioned extensions are yet to be implemented except one: A proof of concept for integration of hyphenation exist.[3]

Written in JavaScript, TypeSet makes use of a few extensions to the JavaScript standard library, namely extensions to `Object` and `Array` and an implementation of a linked list. In order to interact with the HTML page TypeSet is used in, it uses the jQuery library[4]. The data structures needed by the KPA are implemented as simple constructor functions returning slim objects. This especially includes boxes, glues, and penalties but also holds for breakpoints.

As pointed out in subsection 2.2, the KPA does not operate directly on text strings but rather on a list of nodes. To provide the algorithm with this list, TypeSet implements a set of *formatters* that turn a text string into a list of nodes according to the desired formatting; centered, justified, and flush-left formatters are included. To perform the transformation, the text string is split at white space characters and all remaining sub-strings are turned into box-nodes, unconditionally interleaved by node-sequences that denote a space in the respective formatter. Hence, line break characters, tabulator characters and others are treated all the same. After transformation, the TypeSet implementation works like the TEX implementation.

3.2 Integration with the Lively Kernel

Whilst a full implementation of the KPA in JavaScript—TypeSet—is available, integrating it into the Lively Kernel requires adoption.

The Class system The Lively Kernel provides a class system similar to that of Smalltalk [3], contrasting the prototype style of pure JavaScript. In order to consistently use TypeSet in the Lively Kernel, its object construtor style is to be transformed into the class style of the Lively Kernel. Additionally, as no linked list implementation was available in the Lively Kernel, a port of the TypeSet linked list version is used. It is needed to maintain the list of breakpoints in the main part of the KPA.

The Lively Kernel provides a namespacing system into which the TypeSet implementation has been integrated; the `projects.TeX` namespace is reserved for it while under development.

Hooking into text morphs The Lively Kernel provides an *ecosystem* of objects that can all be interacted with at runtime. Lively Kernel Worlds, normally

[3] `http://www.bramstein.com/projects/typeset/flatland/`
[4] `http://jquery.com/`

```
createLayer("TeXLayer");
layerClass(TeXLayer, TextMorph, {
  assureNet: function() {
   this.net = new projects.TeX.TeX.NodeNet(this.textString);
  },
  composeLines: function(proceed, initialStartIndex, initalTopLeft,
      compositionWidth, font, testEarlyEnd) {
    this.assureNet();
    return this.net.composeLines(initialStartIndex, initalTopLeft,
        compositionWidth, font, testEarlyEnd);
  },
  renderText: function(proceed, t, w) {
    var result;
    withLayers([TeXLayer], function(){ result = proceed(t,w); });
    return result;
  },
  resetRendering: function(proceed) {
    delete this.net;
    proceed();
  },
})
```

Listing 5.1: Slightly simplified form of the ContextJS layer used to enable TypeSet line breaking in the Lively Kernel

HTML files, are such ecosystems—combining the kernel of Lively with specific applications. For end user applications it would be natural to bootstrap a Lively Kernel World and extend it with application code. Nevertheless, introducing a different paragraph layout algorithm into the Lively Kernel involves changes within the core parts, the Lively Kernel itself, which text morphs are part of. Now, deliberately changing core logic of any system may result in unexpected behavior, hence, isolation mechanisms are required. It would be possible to bootstrap a Lively Kernel World and work on a copy of the core part without affecting other Worlds, still, reintegrating these changes would be troublesome.

To overcome these issues, a facility for context oriented programming in the Lively Kernel, ContextJS, is used [1]. That way, using the KPA instead of the default Lively Kernel line breaking algorithm can be seen as a behavioral deviation which can be layered over the default implementation. In this case, the context to activate the layer of KPA behavior is *explicit user request*; Text morphs that shall use KPA line breaking have to have the layer activated for themselves.

When text is to be rendered in the Lively Kernel, a text morph is responsible for its text string to be mapped to renderable objects (cf. subsection 2.1). Hence, the behavioral deviation just mentioned has to affect the `TextMorph` class. The key entry point for line breaking is the function `composeLines()` of `TextMorph`. Replacing this function by the TypeSet implementation plus the initialization of

the node list (currently called *net*) is nearly everything needed to integrate the new behavior. Find a simple version of this integration code in Listing 5.1. Note that this listing includes two more functions to ensure that the layer containing the required behavior is activated.

Using this layer, the TypeSet implementation of the KPA can easily be used in any text morph throughout the Lively Kernel without affecting other text morphs using the default implementation.

Text morph API The Lively Kernel is a highly interactive system. Therefore, the text morph of the Lively Kernel includes numerous functions providing *editing* behavior as well as positioning functions to help the Lively Kernel interpret user events such as mouse moves or mouse clicks. Incidentally, all these API functions are implemented in the class `TextLine` that implement the functionality described in section 2.1. To recapitulate: the Lively Kernel line breaking algorithm turns a text string into *text lines* of renderable objects. While producing renderable object with the TypeSet implementation would be possible without the notion of a text line, it seemed reasonable to introduce a text line class specific to the KPA implementation to provide the required API functions: `projects.TeX.TeX.Line`. Except for the `render()` function, all functions implement the API needed for interaction.

⁂

Given the implementation just introduced it is possible to have text morphs in the Lively Kernel use the KPA for line breaking. However, *visualizing* the algorithm is missing at this point. Note that whilst all displaying capabilities necessary are provided, editing of the text is not currently possible.

3.3 Visualizing the KPA

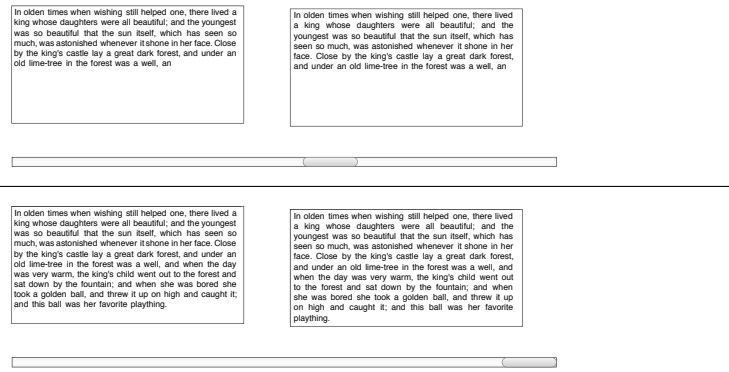

Fig. 3: The slider application: dragging the slider changes the amount of text to be processed. Left: the KPA, right: the default algorithm.

Line breaking is a process where, mostly, elements of a text string are processed in an ordered manner. Is is thus possible to emulate the stream of text that is processed by varying the text string's length. A demo application has been implemented that allows to choose the length of the text using a slider (see Figure 3). From the upper to the lower part, the slider has been dragged to the right, effectively augmenting the text to be rendered. That way, it is possible to monitor the behavior of both algorithms available. Moreover, one key property of the KPA can directly experienced: adding words to the end of a text can affect the line breaking of *previous* lines. This will never happen in traditional algorithms. This application forms the presentation basis for the implemented visualizations.

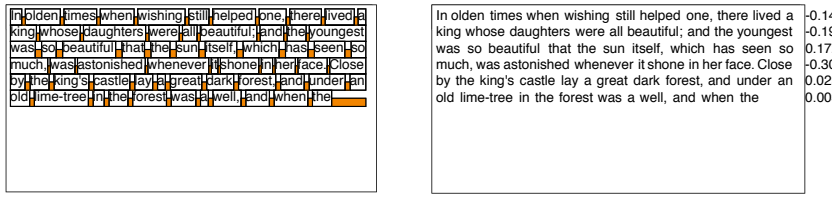

Fig. 4: Example visualization of aspects of the KPA. Left: *box* nodes are highlit by black rectangles around the text; *glue* nodes are denotet as orange rectangles. Right: The number next to each line denotes the *stretch ratio* applied to that line.

Visualizing the KPA is a temporal application, i. e., not every time the KPA is used for a text morph its visualization is desirable. Additionally, there are different aspects of the algorithm to be visualized that are not of the same interest every time. Seeing that, several ContextJS layers have been implemented, each visualizing a specific algorithm aspect individually. For example, in Figure 4, the left part visualized the *box* nodes (black rectangles around the text) and the *glue* nodes (orange rectangles) of the text portion processed by the KPA. The right part, however, visualized the *stretch ratio* applied to each line (cf. [6]).

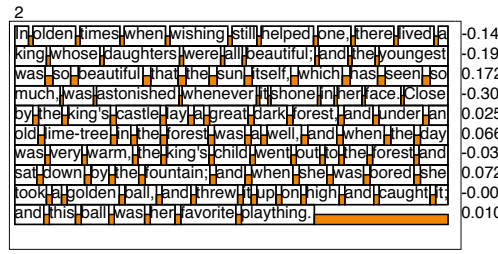

Fig. 5: All visualization layers impemented have been applied to the text; different aspects of the KPA are, thus, visualized simultaneously.

These layers can be combined easily. Hence, a general visualization as in Figure 5 can be built up. In the figure mentioned, the following visualization layers are used:

1. `TeXBoxVisualizationLayer` Highlighting boxes and glues, cf. the left part of Figure 4.
2. `TeXRatioVisualizationLayer` Putting the stretch ratio of each line beside it, cf. the right part of Figure 4.
3. `TeXToleranceVisualizationLayer` Displaying the currently used *tolerance value* at the top left of the text. In Figure 5, it is the 2 in the upper left. Note that the tolerance value determines what stretch ratio is considered acceptable for lines at all.

All these layers can be activated, individually or all at once, while using the slider application mentioned above. That way, the change of the individual status and values used or produced during the line breaking algorithm can be monitored easily by just dragging the slider—a *lively* visualization of the KPA.

4 Related Work

Line breaking has to be done in nearly every text-displaying application. However, especially text related programs have to deal with line breaking. Incidentally, non-trivial algorithms like the KPA are employed in several applications. TEX and all related systems make use of the KPA or extensions thereof, e. g., by [11]—whether with or without accompanying hyphenation algorithm. A related algorithm is used in the (unpublished) *hz-program* by Herman Zapf. The DTP program InDesign by Adobe is known to be loosely based on the KPA and makes use of the *hz-program*.

Besides the reference implementation for the KPA, TEX, other implementations exist. TypeSet [10] was used as basis for the Lively Kernel implementation presented here. The algorithm is also implemented by the Apache FOP (XML-Formatting Objects processor).

5 Wrap Up: Outlook and Summary

While directly editing text using the KPA is not yet possible; the roundtrip 'use the default algorithm for editing and the KPA for displaying' seems natural—the KPA was invented with only displaying text in mind, not editing. Tool support automating this roundtrip is to be implemented. For the visualization, more aspects of the algorithm can be made visual, e. g., the stretch and shrink values of certain nodes or how the possible breakpoints are acutally connected.

By means of the work presented, the goal set has been achieved. Using TypeSet, an implementation of the paragraph layout algorithm used by TEX, the Knuth-Plass algorithm, has been integrated into the Lively Kernel and can optionally be used everywhere replacing the default algorithm. Also, three basic

visualizations for different aspects of the algorithm are provided. Together with the simple slider application, this makes a good starting point for explaining or exploring the KPA—while actually using the algorithm itself.

References

1. Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., Perscheid, M.: A Comparison of Context-oriented Programming Languages. In: COP '09: International Workshop on Context-Oriented Programming. pp. 1–6. ACM, New York, NY, USA (2009)
2. Bellman, R.: Dynamic Programming. Princeton University Press (1957)
3. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
4. Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A., Mikkonen, T.: The Lively Kernel A Self-supporting System on a Web Page. In: Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008 Revised Selected Papers, Lecture Notes in Computer Science, vol. 5146/2008, pp. 31–50. Springer-Verlag, Berlin, Heidelberg (2008)
5. Knuth, D.E.: Digital typography. No. 78 in CLSI Lecture Notes, Center for the Study of Language and Information, Stanford, CA, USA (1999)
6. Knuth, D.E., Plass, M.F.: Breaking paragraphs into lines. Software—Practice and Experience 11(11), 1119–1184 (Nov 1981)
7. Maloney, J.H.: Morphic: The Self User Interface Framework. Sun Microsystems, Inc. (1995), ftp://ftp.squeak.org/docs/Self-4.0-UI-Framework.pdf
8. Maloney, J.H., Smith, R.B.: Directness and Liveness in the Morphic User Interface Construction Environment. In: UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology. pp. 21–28. ACM, New York, NY, USA (1995)
9. Nassi, I., Shneiderman, B.: Flowchart techniques for structured programming. SIGPLAN Not. 8(8), 12–26 (1973)
10. Strein, B.: TEX line breaking algorithm in JavaScript (Nov 7th 2009), http://www.bramstein.com/projects/typeset/, last accessed: Jul. 30th 2010, 11:00
11. Thành, H.T.: Micro-typographic extensions to the TEX typesetting system. Ph.D. thesis, Masaryk University Brno (Faculty of Informatics) (Oct 2000)