Active Essays on the Web

Takashi Yamamiya, Alessandro Warth, Ted Kaehler

Published in the Proceedings of the Seventh Annual International Conference on Creating, Computing, Connecting and Collaborating through Computing, Kyoto University, Kyoto, Japan, January 2009

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Technical Report TR-2009-002

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Active Essays on the Web

Takashi Yamamiya takashi@vpri.org
Alessandro Warth alex@vpri.org
Ted Kaehler ted@vpri.org

Viewpoints Research Institute
1209 Grand Central Ave., Glendale, CA 91201

Abstract

This paper describes "Active Essays" and their implementation with Internet technology. An Active Essay combines a written essay, program fragments, and the resulting live simulations into a single cohesive narrative [11]. We believe the integration of programming and natural language makes a superior teaching medium for expressing mathematical, scientific, and even literary ideas. It is especially effective when it can be read, run, and authored in a web browser. We review our previous implementations of Active Essays on the Web. Chalkboard [25] is our latest Active Essay framework. We discuss Chalkboard's features, examples, design decisions, and unresolved issues.

1. Introduction

Programming languages were made for giving orders to machines. They have evolved to be somewhat more human-friendly and modular, which makes programs easier to maintain. Now we understand that the single most important aspect of a program is readability, not efficiency. An important additional goal is for a programming language to be a good medium for learning and communication between humans.

A programming language has the notable feature that its meaning is strict and ideas expressed in it actually run on a machine. This property has a great benefit: when combined with a written essay, it allows the reader to see an animated and interactive expression of what is being taught. Neither natural language nor mathematical formulas are as alive or expressive. The best way to rigorously learn a complex idea is to program it. While the program you write may not run well at first—often, it will not run at all—the process of debugging will lead to an elegant expression of the idea that actually runs. The result is often good material for another learner to read.

On the other hand, natural language is better than a programming language to show an overview. Written language works well to express the "Why" and "What" knowledge that are hard to describe in a program. The combination of a natural and a programming language in an Active Essay has all of the advantages. It can be an excellent substitute for a textbook.

The Internet gives people large opportunities to access massive amounts of knowledge. But information on the web tends to keep the form of old media like newspapers, TV, and articles. These do not yet use the real ability of the computer. Our project aims to re-introduce programming languages as a way to teach powerful ideas on the web. We want to harmonize literature and programming into a new medium that is more expressive than each alone.

2. Background

2.1. Active Essays

An "Active Essay" is a written essay mixed with a computer program. The term was coined by Alan Kay. The goal was to provide students with a hands-on experience of mathematical and scientific ideas via dynamic simulations. The same concept, but without the essay is used to create and teach dynamic content in the Etoys authoring environment [3, 4, 20].

Think of a math book that includes several paragraphs and formulas on each page. The reader follows the author's idea not only by reading the text but also trying formulas. In the case of Active Essays, programs are embedded in the page instead of formulas. The biggest advantage of an Active Essays is that the program actually runs and animates

This material is based upon work supported by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

the example. This is more dynamic than a formula. It is really fun to see a few lines of code generating little surprises on the screen. Another advantage would be that you don't have to calculate the numbers yourself – the computer does a faster and better job of "fiddling variables".

2.2. World Wide Web

The Internet has made finding and publishing information incredibly easier, but it's content still tends to mimic traditional media. Even though it is highly dependent on computer technology, the web makes little use of programming languages as communication media.

A programming language is the primary tool for talking between man and machine. It could also be a good way to exchange ideas among humans. If there were enough support tools, it would be natural to write a blog post using a programming language, to chat with source code, and even to write a novel that runs as a program. Especially when the topic is science or mathematics, a programming language is often the most accurate way to present an idea.

It is ironic that you can't play with a LOGO program while you read the Wikipedia article about LOGO.

> However, e.g. while the article on Logo has some good information and examples, none on them can be run, dynamically changed and tried, etc. To me this is outrageous given that the browser was done some years after HyperCard, longer after the Apple][, and long long after the prior art of the 60s and 70s. – Alan Kay [12].

What kind of tool could support running examples in an article? The abundant introductory material for science written in Etoys is a good place to start. The history of Literate Programming shows another interesting aspect of the topic. These were our starting points.

2.3. Literate Programming

Active Essays share ideas in common with Don Knuth's Literate Programming. Active Essays emphasize dynamic experimentation, while Literate Programming focuses on the readability of system-level programs. But, both of these media try to combine literature and programming. Advantages of Literate Programming can also be applied to Active Essays.

Don Knuth said that even though more time is required for writing, the total time including writing and debugging in Literate Programming is not greater than normal programming. This is because it takes less time to debug a well-understood program [13]. Although it is possible that his words arise from too much enthusiasm, integrated documentation makes it very much easier for a third person to understand a program.

The original idea of Literate Programming was implemented as the WEB system for Pascal. The WEB generates program source code and documentation from single WEB source file. But, to be compatible with the inflexible nature of Pascal, the syntax of WEB is rather complicated. Attempts using the Smalltalk language [17] reduced this difficulty.

3. Previous attempts

In 1994 Ted Kaehler built an active essay in HyperCard that explained and ran Richard Dawkins' example of evolving the sentence "methinks it is like a weasel" [10]. The HyperTalk scripts for the example were in fields on cards, and could be modified by users. The stack kept old versions of scripts, so that the user could revert after trying out a change to a script.

In 1995, Kaehler built two active essays in web pages using Glyphic Codeworks as plugin to run the code [9]. Each of Kaehler's essays were awkward; the first was not on the web, and the others required a plugin that was difficult to install. These efforts were tantalizing, and caused us to search for an easy and capable active essay format in a web page.

Because there are many design choices when designing an Active Essay framework on the web, we introduce our recent attempts in historical order. These are helpful in examining possible designs and understanding our latest decisions. We have made five experimental implementations over the last few years. Those projects are described with their features, server and client platforms, screen layouts, and data formats. All projects use HTTP for sharing content.

3.1. Scamper Workspace

A Scamper Workspace (2004) [27] by Takashi Yamamiya was an extension of Scamper, a web browser written in Squeak. Scamper Workspace allows the reader to execute any Smalltalk code that is present on a web page. An author can include illustrative examples in Smalltalk on a web page.

People often write a small amount of source code in a blog. And it is natural that a reader may want to run this code without any effort. In the case of Squeak, you have to copy and paste from a web browser to Squeak. However, Squeak contains a web browser named "Scamper", and a web page is just text the same as other Squeak text objects. Scamper is very limited as a web browser, but it has enough power to let the user read blogs.

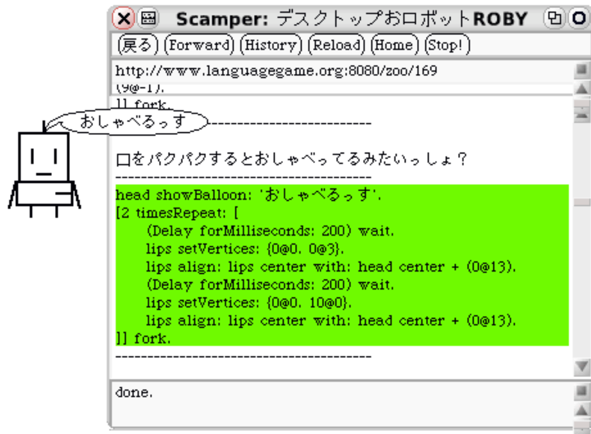

Figure 1. Scamper Workspace.

"No Application" is Squeak's motto. Squeak consists as a number of objects that each have different tiny functions, and those are connected together in a single world of objects. In that sense, there is no need for an "Application" because an application is just an artificial boundary. So, it seemed natural to allow Scamper to have the commands that execute text as code.

Thoru Yamamoto created a lot of content for the Scamper Workspace. Figure 1 explains how to draw a robot by Squeak functions. This is a typical one which consists of a short story and a couple of lines of code. A reader would execute the code while reading a story.

3.2. StackWiki

StackWiki (2005) by Takashi Yamamiya was a WYSIWYG authoring tool written in Squeak. A document was saved in original binary format onto a Swiki (Squeak Wiki) server.

StackWiki was inspired by Zest and Marmalade [19] by Benjamin Schroeder and John Pierce. Although Zest only used a local disk for saving data, the idea was similar to a Wiki. You could easily make a link to another page, and make a new page just by specifying a nonexistent name. Additionally, Zest could include multimedia content written in a Smalltalk-like language.

Compared to Zest, StackWiki was closer to the original idea of HyperCard. A StackWiki project consisted of one or more ordered pages, and relationships among pages were defined by the page order and hyperlinks.

By using a background, selected objects can appear on all pages. StackWiki only allows one background, though HyperCard could handle many. The background was implemented as a special "background" page. If you add something to that page, it is shown in the same place behind other elements on all pages.

Figure 2. A screen shot of the StackWiki.

A background can be seen as a special version of macro or template. A macro is a technique to share a common structure in documents. It is useful to reduce redundancy and to improve maintainability. However, it is easy to make a macro be too complicated by including another macro in it. The background is a better compromise for the end-user.

Each StackWiki page is stored in binary format in the Swiki server. The Swiki is used only as a file uploading server and web page server. StackWiki does not use any web standard except HTTP to transport saved data. StackWiki can only read pages from a Swiki server. StackWiki is simple and uses many built-in Squeak features. It took only three days to implement StackWiki.

3.3. Tinlizzie WysiWiki

Tinlizzie WysiWiki (2006) [15] by Takashi Yamamiya et al. was a wiki written in Tweak [16]. It uses OpenDocument Format (ODF) [6] as its data format, and WebDAV as the server.

While the data format in StackWiki was a Squeak-specific binary, in the Tinlizzie WysiWiki, an existing standard format is used. The reason is to make the user's data available in other applications besides Tinlizzie WysiWiki. A user can confidently put large amounts of important data into this system, knowing that it can be viewed and used in other OpenDocument application programs.

An ODF file is just a zip archive which includes XML text and multimedia binary files. It is easy to extract one of the multimedia binary files from the archive using an external tool. Both internal and external resources can be refer-

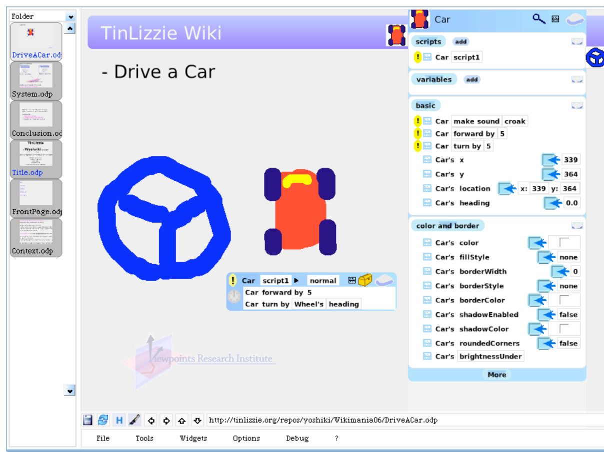

Figure 3. TinLizzie WysiWiki.

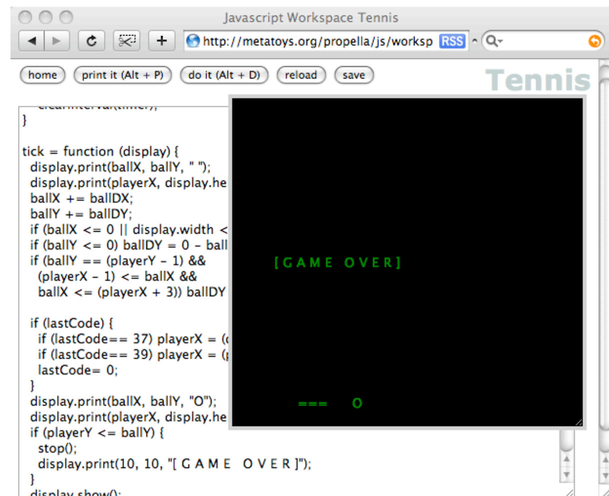

Figure 4. JavaScript Workspace

enced using normal URLs. And if necessary, a new XML tag for a Tweak-specific object can be defined. For example, a project which includes fully dynamic behavior written as Tweak objects can be viewed by an OpenOffice application, although any dynamic features in it would be disabled.

Sometimes the entire state of an object does not need to be save. For example, when text is saved during editing, should the position of the cursor should be saved or not? There are two strategies. One is to save everything with a few things excepted (deep copy), and the other is to save only the specified object types. Tinlizzie WysiWiki took the latter course, although Tweak's native mechanisms were the former.

Saving only a specific state has two disadvantages. a) A user might expect to save everything including minor state because combining arbitrary objects in any peculiar way is possible in Tweak. b) Each new widget needs to be implemented with its own custom exporter. But the "saving everything by default" strategy has a compatibility problem with future version of the system. Changing the name of just one instance variable can make reading an older project impossible. This strategy was not taken by the project.

3.4. JavaScript Workspace

The JavaScript Workspace (2007) [26] by Takashi Yamamiya is a simple web application. It uses a normal web browser and JavaScript on the client side, and Ruby CGI on the server side. It behaves like a Smalltalk Workspace, and content is managed in the same manner as a Wiki.

Let us review the workspace again. A workspace is a text editor with two additional commands, "do it" and "print it". The "do it" command executes the source code selected by the user, and "print it" inserts the result in text after the cursor position. It is similar to a Read Eval Print Loop (REPL) in a dynamic language, but the use is slightly different. A typical usage of a workspace is as documentation for the program. An author often writes example source code inside the workspace, so that a user can try the actual functions while reading the text. A REPL is a two-way dialog between machine and human, while a workspace is a three way conversation among machine, author, and user.

The workspace is an indispensable tool in Smalltalk, so it makes sense that it would be useful in other systems. It would be nice if there were a workspace for JavaScript. This was the initial motivation for JavaScript Workspace. And then it was a natural consequence to use a Wiki to save the content.

During development, however, we realized that it was more than just a workspace. It was an entire web application. JavaScript Workspace has a simple user interface, which includes a couple of buttons and one big text area. The text area does not allow hyperlinks or emphasized text. But, a variety of things can be made from such minimal configuration. It is possible to make hyperlinks by assigning into the `location` property of the `window` object and rich text could be shown by modifying the DOM tree. Even games could be created by setting up an event hander and a timer (Figure 4). A piece of source code can do almost anything.

Just one text box on a web page is a radical idea. It is in the complete opposite direction of the current trend toward rich Internet application pages. Web applications consist of a number of hidden functions these days, but JavaScript Workspace does not have any invisible information. Everything on the screen can be seen as source code. JavaScript Workspace may appear dangerous since it can run any JavaScript code, but in fact, it is quite a safe system.

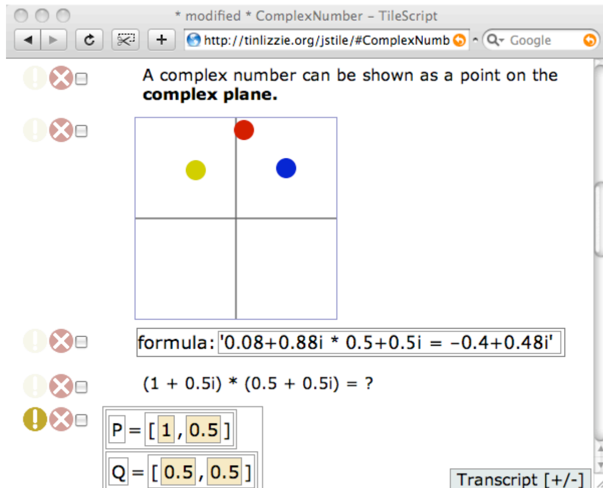

Figure 5. TileScript

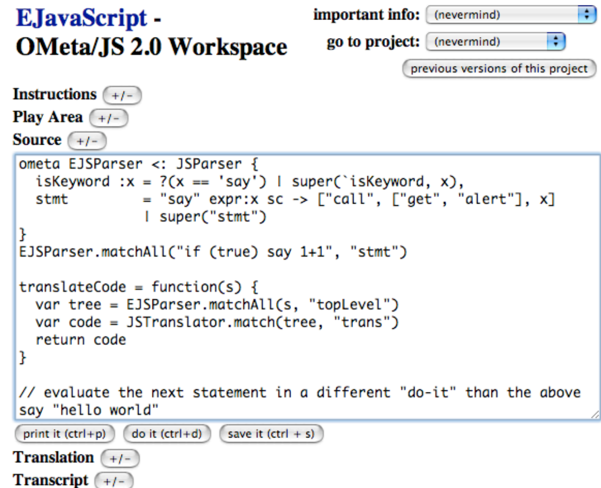

Figure 6. The OMeta/JS Workspace

The user interface of the JavaScript Workspace was adopted by OMeta/JS Workspace (see Section 3.6).

3.5. TileScript

TileScript (2008) [24] by Takashi Yamamiya et al. uses Scriptaculous [2] as its GUI library and WebDAV for server storage. Its data format was JSON.

A TileScript document consists of one or more paragraphs, and a paragraph is either JavaScript code, a "tile script", or an HTML expression. A tile script is a set of draggable tile object, where each tile represents a syntactic element in the programming language. You can combine tiles to construct a program using drag and drop. This is an easy way to make a program and avoid all syntax errors. JavaScript can represent more complex programs than TileScript. HTML is used for annotation and explanation. TileScript is a richer version of JavaScript Workspace.

The motivation of TileScript was to investigate remaking Etoys as a web application. The initial idea was to make tiles available in a web browser. After tiles worked, the next step was to hook up the Etoys environment itself, which includes event handling, scheduling and bitmap animation. But those issues seemed too difficult for the current nature of a web document.

Flow layout, in which the actual positions of the elements on a page are dynamically determined by the reader's browser, is a significant feature of a web document. You don't have to specify the concrete position of elements, but rather just the logical structure.

On the other hand, Etoys allows free layout, where the size and position of elements are fixed by the author. It assumes a particular screen size. Free layout works well for graphical application like Etoys, but if the reader's screen size is smaller than the original, clumsy operations like zooming and horizontal scrolling are required.

Because the goal of TileScript was not simply reproducing Etoys, but investigating new possibilities, flow layout was adopted in TileScript. Position was supported in the form of objects embedded HTML. TileScript provided for variable watchers like Etoys, but those widgets were also laid out with text flow.

3.6. OMeta/JS Workspace

The OMeta/JS Workspace (2007) [21] by Alex Warth extends its predecessor, the JavaScript Workspace, with support for arbitrary programming languages. This is done via a user-defined function called `translateCode`, which is implicitly called by the Workspace in order to translate the code that the user wishes to execute into JavaScript, "the assembly language of the Internet" [8].

The benefits of this almost trivial extension are twofold. First, it makes it possible for the user to express his ideas in the language that is best suited for the job, which is especially important for active essays. For example, when discussing differential geometry, it makes much more sense to write programs in a language like Logo than in JavaScript. In this case, the user can define `translateCode` to be a function that translates Logo to JavaScript.* (Such translators are fairly straightforward to implement using OMeta [23].) This enables the user to get out from under the limitations of JavaScript, effectively putting him in charge of his own software destiny.

Second, the flexibility provided by `translateCode` turns the web browser into a convenient platform in which to experiment with new programming language ideas. In

*See `http://jarrett.cs.ucla.edu/ometa-js/#Logo`

the past year, the OMeta/JS Workspace has been used as the user interface of a number of experimental programming languages, e.g., Etude (a language for describing music)†, Worlds/JS (a language that enables programmers to control the scope of side effects) [22], and even OMeta/JS itself. Users are able experiment with these new languages inside the web browser, without having to download any additional software, which makes our research much more accessible.

4. Chalkboard

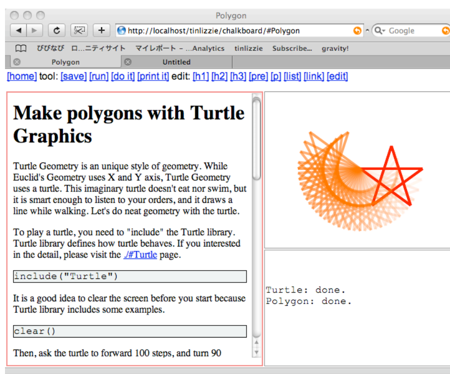

Figure 7. Chalkboard

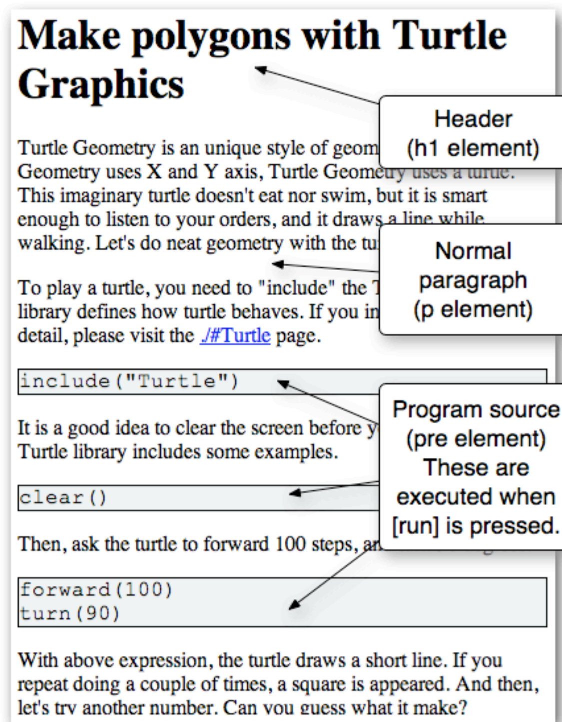

Figure 8. Chalkboard Document

Chalkboard by Takashi Yamamiya is the latest attempt of an authoring tool for Active Essays on the Web. In this project, some design decisions were based on past work, these decisions are described at section 4.3, 4.4 and 4.5. In short, because the aim of Chalkboard is to provide a simple and robust platform for Active Essays, the simpler way was always chosen.

4.1. Document structure

Figure 8 shows a typical Chalkboard project. You can use standard HTML formats like normal paragraph, header, list, hyper text and source code (`<pre>` element). Source code is shown in a fixed pitch font and gray background.

You can execute any text in the document, but the run command and `include` function only work with code inside a `<pre>` element.

To make a new project, you simply access a nonexistent URL on the Chalkboard, or make a new link as `./#NewProject` with the link command. When the project is saved, a new project is created.

There are two ways of running a program in a project. The primary case is using a "do it" or "print it" command on each fragment of code in order. A reader invokes this command on program text, and experiments with the content step by step. Another way is to use the "run" command to execute all of the code.

The `include` function provides a way to use a project as a library. When as the `include` is called, the specified project its executed in the same way as the "run" command, except there are no visual effects. It allows an author to hide unessential parts of the program.

4.2. Chalkboard's user interface

The screen of Chalkboard is made up of the following parts:

- Tools: There are command buttons in the top-most area. Every function of Chalkboard can be invoked from this tool bar.

  home Link to the home page.

  save Save the page to the server.

  run Evaluate all JavaScript expressions within the source code areas. Each result is printed in the transcript area. If an error happens, the border surrounding the source code becomes red and the editor area is scrolled to show it. Once the error is fixed, the border becomes black.

†See `http://jarrett.cs.ucla.edu/ometa-js/#Etude`

do it Evaluate the selected expression or the selected line if there is no selection. The resulting value is shown in the transcript.

print it The same as do it, but print the result just to the right of the expression.

h1, h2, h3 Make selected lines into a header. These generate `<h1>, <h2>, <h3>` HTML elements respectively.

pre Make the selected lines into a source code area.

p Make selected lines into a paragraph.

list Toggle to a list item or normal paragraph.

link Assumes selected string is a URL, and it makes it into a hyperlink.

edit Toggle browse and edit mode. This is unnecessary in most browsers. But, some web browsers disable the event handler in editor mode, and hyperlinks work only in browse mode.

- Editor: The largest area on screen is a text editor where you can write descriptive text and JavaScript code.
- Play Area: The top right area is referenced from the global variable PlayArea. It is typically used to show a graphical object with a Canvas element. The source code in the editor draws on this Canvas.
- Transcript: Results of "do it" and "run" commands are printed on the right bottom area. It is also used to show general output via the `display` function.

4.3. Client and Server platform

A web application requires both client and server programs. To keep the development process simple, the Apache web server and WebDAV protocols are used as the server logic of TileScript, OMeta/JS, and Chalkboard. With WebDAV, the remote server can be used in the same manner as local storage. In addition, version control is provided by a Subversion module.

Each project is specified by a URL with the name of the project after a hash mark (#). Normally, a hash element is used to point to a place within a document, but we treat it as a project name. An advantage of this trick is that a web browser doesn't wipe and reload the JavaScript when you change projects. A disadvantage is that Internet Explorer doesn't handle these URLs properly in the history list. This method is derived from the way GMail [1] works.

4.4. Screen layout

There are two options for screen layout. One is free layout, which allows the user to place an object in any position on the screen. Squeak-based projects, Etoys, StackWiki and Tinlizzie WysiWiki are designed in this style. Another option is flow layout, in which objects in a page are aligned in paragraphs, and the position on the screen is adjusted based on the screen width. TileScript is designed with this style. Flow layout was chosen for Chalkboard because of usability on the web, as explained in section 3.5.

4.5. Data format

Powerful file formats and interoperability are in a trade off relationship. Squeak exports the internal memory as a project file. This is powerful and easy for rapid development. However, it is difficult to maintain compatibility because it depends on all of Squeak. Chalkboard uses HTML as its data format. The main part of the file is the explanatory text, and the program is extracted from the PRE elements.

There is a special simplicity in Chalkboard's UI. Only logical format commands for a paragraph are supported. There are no emphasis or font size commands. This can help an author avoid strange looking layouts depending on the web browser.

5. Related Work

Emergence [18] is an active essay based on Conway's Game of Life. It is implemented as a series of web pages that, using a combination of text and Java applets, explain the rules of the system and enable the reader to interact with a live simulation. Unfortunately, the system is exposed as a black box that cannot be inspected or modified by the reader.

The LogoWiki [7] was a Javascript-based implementation of the Logo programming language that enabled users to run, modify, and share programs in the web browser.

SophieScript [14] is a scripting language that makes it possible for active assays to be built on top of the Sophie multimedia editing environment.

The Lively Kernel [5] is a Smalltalk-like programming environment that operates inside the web browser. Its mechanism for enabling users to share their projects was inspired by our JavaScript Workspace implementation.

6. Conclusion and Future Work

We have briefly discussed the concept of Active Essays, and their implementation as Internet applications. Quite a few prototypes have been developed over the past years. Based on our experiments, we have developed Chalkboard as an Active Essay web application using a normal web browser and a WebDAV server.

To allow collaboration between users, we are considering more sophisticated extensions to Chalkboard:

- **Versioning**

  The most effective way to learn from content in Chalkboard is not only following the author's intention, but also modifying minor parts of the project to see how it effects the result. Sometimes a reader might want to save for future use a project that is halfway modified, but not want to break the original project. In this case, keeping on the server versions for each user independently is useful.

- **Guidance**

  Some projects in Chalkboard offer the reader a chance to follow step by step instruction, as a in tutorial. In this case, it is desirable to keep track of the reader's behavior and record an error when some problem occurs.

7. Acknowledgments

We owe Alan Kay thanks for his encouragement. The layout design of Chalkboard is based on his idea of how to teach students the basic ideas of computer science. Scott Wallace and Yoshiki Ohshima often have joined our discussions. And we would like to express thanks for the valuable insights that Kim Rose, Ian Piumarta, Bert Freudenberg, Hesam Samimi, and other colleagues have given us.

References

[1] Gmail. `http://mail.google.com`.
[2] Scriptaculous website. `http://script.aculo.us`.
[3] Squeak Etoys website. `http://squeakland.org`.
[4] B. J. Allen-Conn and K. Rose. *Powerful Ideas in the Classroom*. Viewpoints Research Institute, Inc., August 2003.
[5] D. I. Antero Taivalsaari, Tommi Mikkonen and K. Palacz. Web browser as an application platform: The lively kernel experience. SML Technical Report TR-2008-175, 2008.
[6] M. Brauer, P. Durusau, G. Edwards, D. Faure, T. Magliery, B. Radius, and D. Vogelheim. Open Document Format for Office Applications (OpenDocument) v1.0. `http://docs.oasis-open.org/office/v1.0`.
[7] A. Bryant. Logowiki. `http://web.archive.org/web/20060708094224/http://www.logowiki.net/`.
[8] D. Ingalls. Message on squeak-dev mailing list. `http://lists.squeakfoundation.org/pipermail/squeak-dev/2008-September/131243.html`, September 2008.
[9] T. Kaehler. Active Essays. `http://web.archive.org/web/19970104044631/http://www.research.apple.com/research/proj/learning_concepts/evolution_active_essay/active_essay.html`, 1995.
[10] T. Kaehler. An Active Essay on Evolution. Apple Research Note RN-95-51, 1995.
[11] A. Kay. Active essays. `http://web.archive.org/web/20060710213801/http://www.squeakland.org/whatis/a_essays.html`.
[12] A. Kay. A "little demo". `http://www.redhat.com/archives/olpc-software/2006-April/msg00035.html`.
[13] D. E. Knuth. *Literate programming*. Center for the Study of Language and Information, Stanford, CA, USA, 1992.
[14] J. Lincke, R. Hirschfeld, M. Rüger, and M. Masuch. Sophiescript - active content in multimedia documents. In *C5 '08: Proceedings of the Sixth International Conference on Creating, Connecting and Collaborating through Computing (c5 2008)*, pages 21–28, Washington, DC, USA, 2008. IEEE Computer Society.
[15] Y. Ohshima, T. Yamamiya, S. Wallace, and A. Raab. Tinlizziewysiwiki 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*, pages 36–46, Washington, DC, USA, 2007. IEEE Computer Society.
[16] A. Raab. Tweak. `http://tweakproject.org/`.
[17] T. Reenskaug and A. L. Skaar. An environment for literate smalltalk programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 337–345, New York, NY, USA, 1989. ACM.
[18] M. Resnick and B. Silverman. Exploring emergence. `http://llk.media.mit.edu/projects/emergence/`, 1996.
[19] B. Schroeder and J. Pierce. Zest and marmalade. `http://114.csail.mit.edu/slides/zest.tgz`.
[20] J. Steinmetz. *Squeak: Open Personal Computing and Multimedia*, chapter Computers and Squeak as Environments for Learning, pages 471–476. Prentice Hall PTR, 2001.
[21] A. Warth. OMeta/JS Workspace. `http://jarrett.cs.ucla.edu/ometa-js/`.
[22] A. Warth and A. Kay. Worlds: Controlling the Scope of Side Effects. VPRI Research Note RN-2008-003, `http://www.vpri.org/pdf/rn2008001_worlds.pdf`, 2008.
[23] A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM.
[24] A. Warth, T. Yamamiya, Y. Ohshima, and S. Wallace. Toward a more scalable end-user scripting language. In *C5 '08: Proceedings of the Sixth International Conference on Creating, Connecting and Collaborating through Computing (c5 2008)*, pages 172–178, Washington, DC, USA, 2008. IEEE Computer Society.
[25] T. Yamamiya. Chalkboard. `http://tinlizzie.org/chalkboard/`.
[26] T. Yamamiya. JavaScript Workspace. `http://metatoys.org/propella/js/workspace.cgi`.
[27] T. Yamamiya. Scamper Workspace. `http://languagegame.org:8080/propella/91`.