Seamlessly Editing the Web

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Computer and Mathematical Sciences with Masters
at the
University of Waikato
by

Brook Novak
University of Waikato
Hamilton
New Zealand
bjn8@cs.waikato.ac.nz

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

University of Waikato
2010

Abstract

The typical process of editing content on the web is strongly moded. Authors are forced to switch between "editing" and "previewing" and "publishing" modes before, during, and after the editing process. This thesis explores a new paradigm of editing content on the web called *seamless editing*. Unlike existing techniques for editing content on the web, seamless editing is modeless, enabling authors to directly edit content on web pages without the need to switch between any modes. The absence of modes reduces the amount of cognitive complexity involved with the editing process. A software framework called *Seaweed* was developed for providing seamlessly editable web pages in any common web browser, and is shown that it can be integrated into any content management system. For the purposes of experimentation, the content management system WordPress was selected, and a plugin using the Seaweed framework developed for it that provided a seamlessly editable environment. Two experiments were conducted. The first study observed users with no or minimal experience with using WordPress, following a set of prescribed tasks, both with and without the plugin. The second study was conducted over a longer time period in a real-world context, where existing WordPress users were naturally observed using the plugin within their own blogs. Analysis of logged interactions and pre-questionnaires and post-questionnaires showed that, in both studies, the participants found the Seaweed software to be intuitive and the new way of editing content to be easily adaptable. Additionally, the analysis showed that the participants found the concept of seamless editing to be useful, and could see it being useful in many other contexts, other than blogs.

Acknowledgements

I would like to begin by thanking my family for supporting me throughout my academic years. You have all supported me in many ways — mentally, spiritually, financially — I am very grateful for this.

A big thanks Dr David Bainbridge. It has been a real privileged to have had you as my supervisor for the last two years. I have learnt so much from our many discussions.

Thank you all at Digital Library Consulting. Although the direction of the work took a different turn, you continued to support me to the end. The meetings, use of your facilities and resources, Grillers, and expertise had really helped me pull this thing together! I am so grateful for your generosity. It was such a pleasure to have met you all, and to have worked along side you.

Thanks to Dr Doris Jung for not only taking the time to revise my work, but all the positive support through each step of the way.

Thanks to Dr Steve Jones for taking interest into my work, and helping me with my write up.

Alice, Pip and Kelly, you all have supported me "heaps as" throughout the year. Thanks for proof reading literature what may at times have seemed from another world.

A shout out to all the people who took the time to participate in my studies — without your input I would have been lost. I am very grateful for your input. Thank you all.

Cheers to the boys for the many surfing missions throughout the year. These times were a much needed sanctuary from my research.

Thank You God for giving the motivation and determination to achieve more than I ever would have achieved for my own sake. This is for You. You have blessed me abundantly through my academia years with magnificent people. Thank You.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the last two decades the world wide web has progressively developed from a read-only environment maintained by IT professionals and computer enthusiasts, into an environment in which anyone can contribute content without technical expertise. The web has experienced three authoring paradigms since its birth, and at each advancement, the authoring process has been marginally simplified. The establishment of the *web 2.0* in 2004 marks the most recent paradigm shift: a revolution for authorship on the web. With the introduction of the AJAX (Asynchronous JavaScript And XML) technology and a widespread support for browser-based WYSIWYG (What You See Is What You Get) editors, the web has matured into an environment for the public to participate in. Today, six years later, the authoring process has hardly changed, enforcing an unnatural distinction between viewing and editing and publishing. This thesis explores a new authoring process for the web called *seamless editing*, which simplifies the editing process. Seamless editing is set apart from other methods of editing as it is entirely modeless, that is, there is no distinction between editing, previewing or publishing modes. Compared with typical editing methods, seamless editing reduces the degree of cognitive complexity involved when carrying out editing tasks on the web.

A framework called Seaweed was developed for supporting seamless editing on the web. The framework provides modeless WYSIWYG editing facilities for manipulating content in real-time for any web page. The framework was integrated into a content management system (CMS) called WordPress: a

system for publishing blog entries on the web. Two separate user studies were conducted exploring the seamless editing concept on the web using WordPress blogs extended with the Seaweed framework. The next section provides an illustration of seamless editing on the web, and is followed by the contributions that came out of this work. The final section of this chapter presents the structure of this thesis.

1.1 An Illustration

Consider the following example of a user editing content on the web: a user is reading an article in Wikipedia. The user reads a sentence, located in the fifth paragraph for the second section, which they feel is too vague. Figure 1.1 presents two sets of steps, that the user could carry out to change the paragraph in the web page. The set of steps on the left presents a moded editing process, reflecting the typical work-flow that users must undergo when editing content, using standard features in Wikipedia. The set of steps on the right presents the work-flow for a seamless editing approach. It is assumed for both methods of editing, that the user is already logged into their Wikipedia account.

Figure 1.1 highlights the way in which seamless editing simplifies this task. Reducing the amount of steps, in effect, reduces the amount of time and cognitive effort needed to achieve a task. The figure also illustrates the range of transitions between modes that the user makes, while performing the standard editing process. At each transition, the user may have to wait due to load times. Once a transition is complete, they must adjust their mindset in order to work with the new mode, and interpret the new perspective, of the content they want to change. The unnecessary steps and the transitions between modes can distract the user, disrupting them from achieving their task. At each transition the user may have to remind themselves of what they were originally trying to achieve.

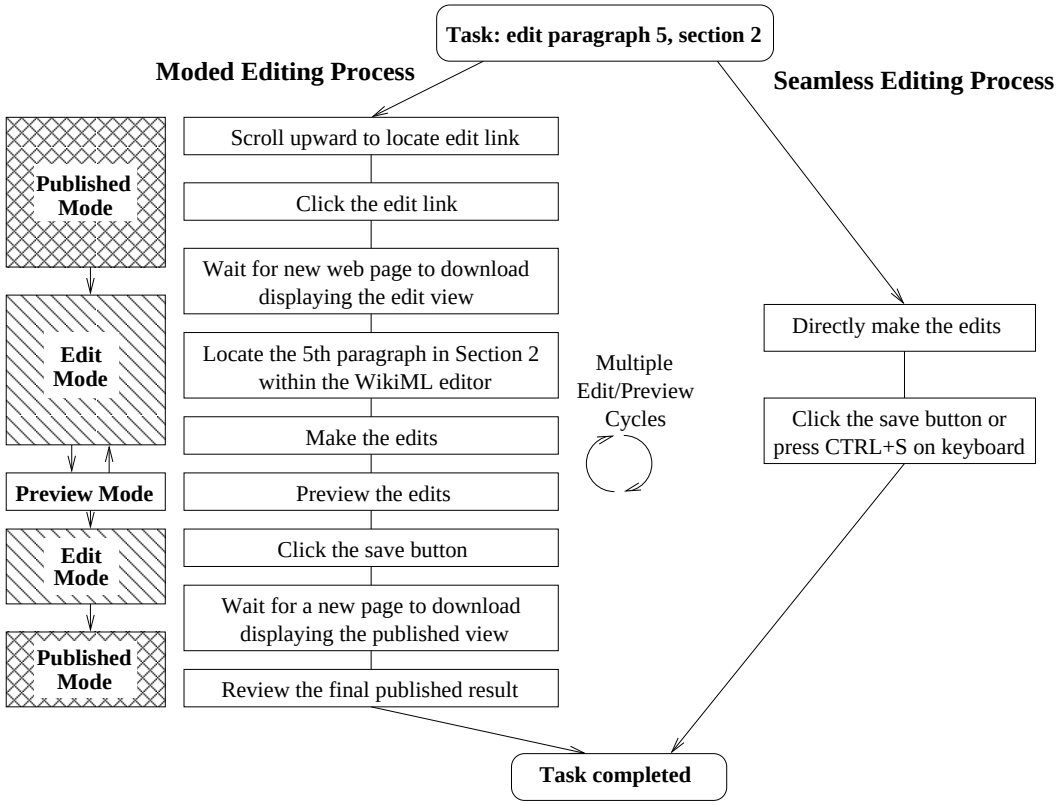


Figure 1.1: Comparison of work-flows for editing content on the web.

1.2 Contributions to Research

This research explores seamless editing on the web, a new paradigm for authoring content. Two observational user studies were conducted using WordPress, extended with a plugin for providing seamless editing. The first study observed participants with no to minimal experience with using WordPress, following a set of given tasks with and without using the plugin, on temporarily assigned blogs. The second study was conducted in a real-world context, where existing WordPress users were naturally observed using the plugin on their own blogs. Both studies collected both observational data, and qualitative feedback from surveys, messages and emails sent to the researcher. Six research questions were established for evaluating the concept of seamless editing and the Seaweed software:

1. What are the situations in which people prefer using Seaweed over using an external WYSIWYG editor? And what are the situations in which they do not? Participants in both studies generally prefered Seaweed

over the WYSIWYG editor in WordPress. In particular, minor sized edits on published content were the most common type of edits, and generally were mostly made using Seaweed.

2. What are the situations in which people prefer using Seaweed over writing raw HTML markup? And what are the situations in which they do not? Some participants prefered HTML editors over visual editors in general, as they liked to have total control over the HTML source themselves. However, in the general case, the participants preferred using Seaweed when making edits on published content.

3. How intuitive is Seaweed? The outcomes from both of the studies showed that the Seaweed plugin was highly intuitive.

4. How well people who have substantial experience with the traditional way of editing in WordPress adapt to seamless editing? The second study found evidence that people who are accustomed to editing their own blogs in their own way, find it easy to adapt to the seamless way of editing.

5. Do people who access the web, like the concept of seamless editing? All participants in both studies liked the concept of seamless editing. Many of the participants gave additional positive feedback toward the concept, such as "the editing-in-place interaction [that Seaweed offers] is something I've wanted forever."

6. What are other contexts where people who access the web, could see seamless editing being helpful (other than blogs)? Seamless editing is seen as a fundamental concept, that people can see being useful in many contexts other than blogs.

The developments for making WordPress a seamlessly editable environment using the Seaweed framework, can be adopted for any CMS. The framework itself was designed to work in any common web browser, and to be used in any type of web page. Thus for future endeavours, the work presented in this

thesis provides a guide for creating seamlessly editable environments in other contexts other than blogs.

1.3 Synopsis

Chapter 2 presents a literature review, discussing the fundamentals of web pages, the core principles that seamless editing is built upon, a history of authoring content on the web, and investigates existing modeless editors for hypermedia environments and the web. The implementation of the Seaweed framework developed for supporting seamlessly editing environments is presented in Chapter 3. The developments made for integrating the Seaweed framework into WordPress — the chosen CMS for evaluating seamless editing — are presented in Chapter 4. Chapter 5 reports on two user studies carried out for evaluating the seamless editing concept, and addresses the six research questions. Lastly, Chapter 6 concludes the work, discusses generalising seamless editing for the web, and details future work in this area of research.

Chapter 2

Literature Review

This chapter begins with establishing two classifications of web pages in terms of their existence and organisation (Section 2.1). Central to the discussion are the HCI concepts and principles of direct manipulation, which provides a language to describe and evaluate systems related to the central work of this thesis and provides a foundation of principles that the seamless editing concept is built upon (Section 2.2). Issues surrounding WYSIWYG editing with HTML documents are then visited in Section 2.3. Section 2.4 looks at two conceptual roles on the web — contributors and consumers — and how they have merged through the simplification of the editing process over time. Section 2.5 investigates a web page editing method called *in-place editing*, and discusses the method in terms of direct manipulation. Finally, Section 2.6 presents and evaluates several systems classified as being *modeless editors*, the ideal candidates for simplifying the editing process. Of all the modeless editors investigated, a system called Mozile required the least amount of cognitive effort to edit content on the web. However, Mozile could not be used for this project because it is an abandoned project that only supports a small set of outdated web browsers.

2.1 The Anatomy of a Web Page

Web pages can be classified as being either static or dynamic. Dynamic web pages can further be classified as being composite or non-composite. We now

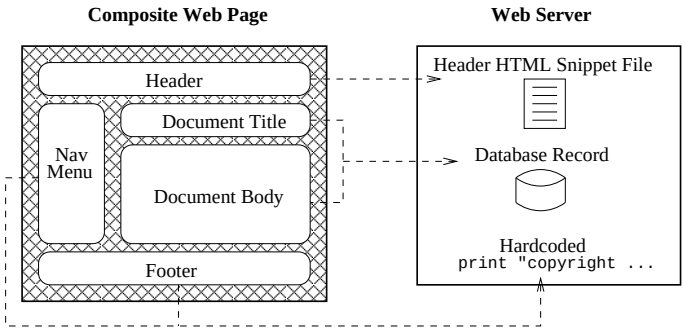


Figure 2.1: A dynamic web page as a composition of various data sources.

discuss these in turn.

2.1.1 Static and Dynamic Web Pages

Static web pages are actual documents stored on a web server's file system. A static page does not change: the document remains the same every time it is visited.

Dynamic web pages are virtual documents, they do not actually exist on a file system but instead are generated by a computer program each time the page is requested. The content and structure of dynamic web pages can be determined by the state of the session. For example, a page might display a *logout* button for authenticated sessions, whereas unauthenticated sessions may display a *login* button.

These definitions of static and dynamic are from the server's point of view, not the visitors. A static web page may have JavaScript to give the document a dynamic aspect on the client-side. For example, animated drop down menus. As with dynamic web pages, a server might generate a web page with the same content every time, appearing as static content on the client-side. Except where otherwise stated, the terms static and dynamic refer to the physical existence of the digital document.

2.1.2 Composite Web Pages

Dynamic web pages are usually generated as a mash-up of a range of data sources. Figure 2.1 displays the layout of a web page containing five sections.

Each section of the page represents a single chunk of information that resides on the web server. For example, the "Document Body" section of the web page (which may contain a new article) came from a database on the web server.

For the purposes of this thesis, a web page comprised of at least one data-source other than hard-coded content is considered to be a *composite web page*.

2.1.3 Separation of Structure and Presentation

There is a wide variety of possible display devices for a web page. For example, print-outs, web browsers on desktop computers or web browsers on small screen devices. To simplify the authoring process of device-independent pages the document only specifies the content and structure, and a markup language called CSS (Cascading Style Sheets) is used to describe the presentation.

2.2 Direct Manipulation Systems

This section discusses the direct manipulation principles, which the concept of seamless editing is built upon. The term *direct manipulation* was coined by Shneiderman in 1983 [30], and is used to refer to systems which have the following properties/principles:

- Continuous representation of the object of interest.
- Physical actions or labelled button presses instead of using a complex syntax.
- Rapid, incremental, reversible operations whose impact on the object of interest is immediately visible.
- Permits usage with minimal knowledge.

Sketchpad was one of the first systems that exhibited properties of a direct manipulation (1963) as discussed in [19]. Sketchpad is a graphical design program, that displayed its editing area as sheets of paper containing graphical objects which could be manipulated via a pointing device.

Shneiderman notes that a key virtue of direct manipulation systems is *intuitiveness*. Users can quickly and effortlessly learn the system's basic/essential functionality. They can learn advanced features later in their own time. As users become familiar with the system, they can predict system responses.

Another key virtue of direct manipulation systems is that they require minimal cognitive effort to complete a task. Users can immediately see if their choice of action produces the desired result. Because actions can be reversed users do not need to worry about making mistakes, thus error messages are rarely needed.

A measurement on how well a system implements the direct manipulation principles can be described as *directness*. The notion of directness — a feeling or impression that results for interacting with an interface — is evaluated in [19]. Directness is broken down into two aspects: distance and engagement. The former is the distance between ones thoughts and the physical requirements of the system under use. The latter describes the level at which the interface makes the users feel as if they are directly manipulating the objects of interest.

2.2.1 Distance of Directness

The distance of directness refers to the relationship between the task that a user has in mind and the way that it can be accomplished via an interface. A short distance means that the translation between a task and actions required to achieve a desired result is simple and straightforward: that thoughts are readily translated into the physical actions required by the system, and that the system output is in a form readily interpreted for verifying that the actions addressed the user's goals of interest. Thus the critical issue for a system to achieve a short distance is to minimise the amount of cognitive effort required to bridge the gulf between the user's goals and the way they must be specified to the system.

Whenever we interact with a device, we are using an interface language. That is, we must use a language to describe to the device the nature of the actions we wish to have performed. Two dialects are involved in an interface

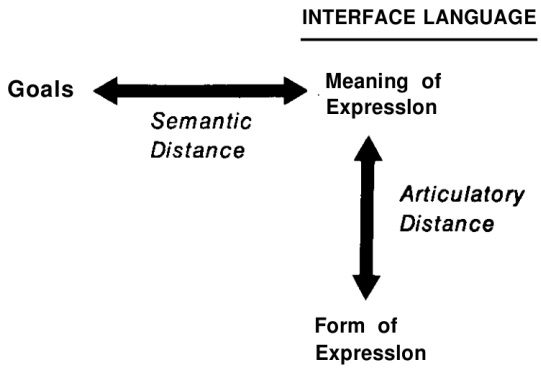


Figure 2.2: Relationship of semantic and articulatory distances with the interface language (reproduced from [19]).

language: the input interface language, spoken by the user, and the output interface language, feedback spoken by the system. Hutchins *et al.* identify two properties in interface languages: *semantic distance* and *articulatory distance*.

Semantic Distance

Figure 2.2 illustrates how the semantic distance concerns the relation of the meaning of an expression in the interface language to what the user wants to say (their goals). Two important questions about semantic distance are:

- Is it possible to say what one wants to say in this language? That is, does the language support the user's conception of the task domain?
- Can the goal of interest be said concisely? Can the user say what is wanted in a straightforward fashion, or must the user construct a complicated expression to do what appears in the user's thoughts as a conceptually simple task?

Articulatory Distance

Articulatory distance concerns the relation of the meaning of an expression and their physical form, as illustrated in Figure 2.2. On the input side, the form may be a sequence of character-selecting key presses for a command language interface, the movement of a mouse and the associated "mouse clicks" in a pointing device interface, or a phonetic string in a speech interface. On the

output side, the form might be a string of characters, a change in an iconic representation, or variation in an auditory signal.

Gulfs of Execution and Evaluation

Both semantic and articulatory distances are involved during the two phases of carrying out a task: execution and evaluation. The execution phase refers to the process of the translation of users' goals to the actions required to achieve those goals (the input interface language). The evaluation phase refers to the process of the users verifying whether the goals have been achieved via the output interface language. The two gaps comprised of semantic and articulatory distances during these phases are referred to as *gulfs*.

Figure 2.3 shows the relationships among the semantic distance, articulatory distance and the gulfs of execution and evaluation. The semantic distance in the gulf of execution refers to the formulation of a goal that the user wishes to carry out using the system. The articulatory distance in the gulf of execution refers to the input expression(s) used to meet the user's goal. Once the system interprets the user's input, the response is output to the screen ready for the user to interpret. The process of the user interpreting the system output spans the articulatory distance in the gulf of evaluation. Finally, the semantic distance in the gulf of evaluation refers to the process of the user accessing whether the final result — the output expression — achieved their goals.

2.2.2 Direct Engagement

The degree to which a system makes the user's experience feel as if they are directly interacting with objects of interest can be described in terms of *direct engagement*.

A system with a high level of direct engagement uses a *model-world metaphor* [2]: providing the user with a world of objects which they interact with. The input and output interface languages have a close relationship in such systems, where the output language is the very artifacts which the users refer to during

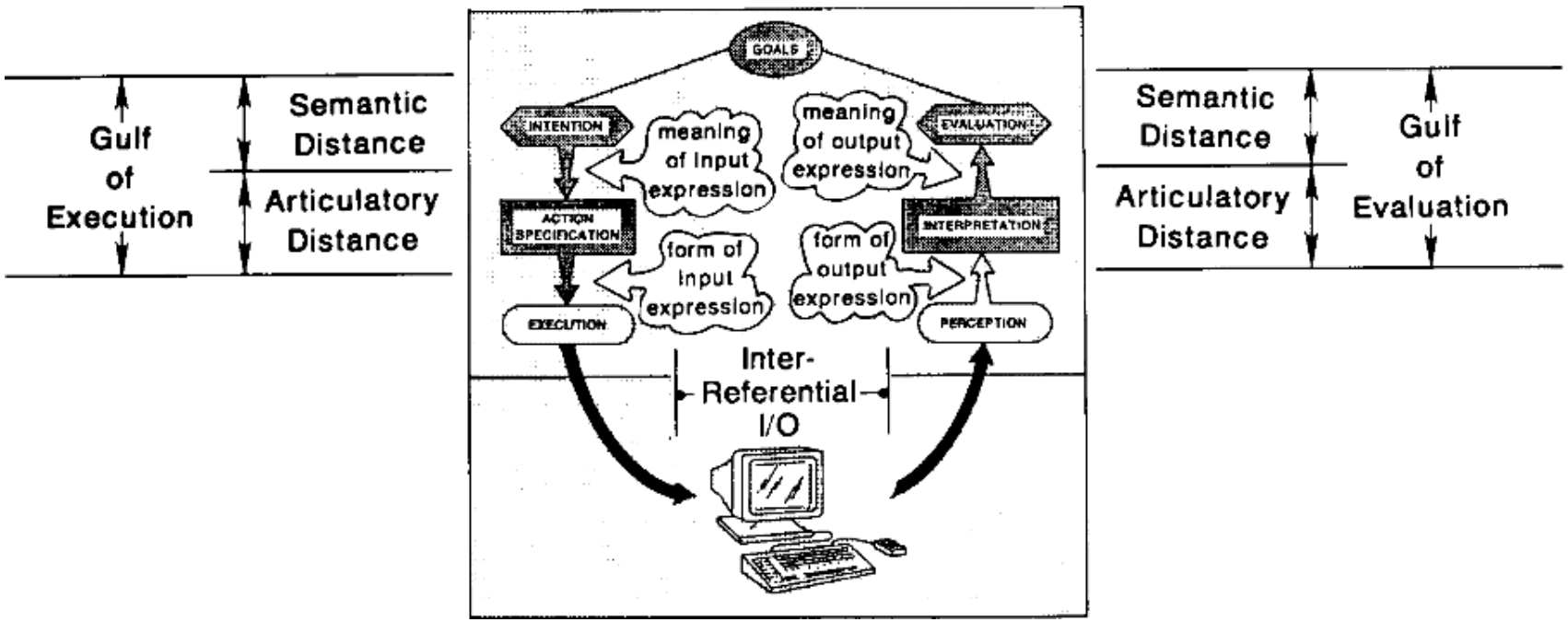


Figure 2.3: The gulfs of execution and evaluation (reproduced from [19]).

interactions, and can be used as part of the input language (which is said to be inter-referential). For example, the drawing program known as Microsoft Paint[1] uses the model-world metaphor in that users interact with a canvas using a set of artistic tools. To create a stick man figure, the user selects a paint brush object, and draws the figure directly on the canvas object. As the user draws the picture the system renders the lines on the canvas. Here the input-language — the mouse gestures to create the lines — is mimicked by the output language: the displaying of the painted lines directly in-place of where the gestures took place. After the user strokes the first lines for the arms, legs and torso, they can interact directly with the output language. For example once the arms are drawn the user may then shorten the arm lengths using a eraser tool to make the stick man figure's limbs proportionate to one another.

Conversely, a system with a low level of direct engagement uses a *conversion metaphor*: users must interact with an intermediary to a hidden world, denying the user from direct engagement. Returning to the stick man figure example: imagine that the user of a graphics system where the creation of lines is controlled by typing stroke commands in a console, and a preview is rendered after executing each command. Here the input language is physically separate from the output language because the relationship is not inter-referential. The user is obstructed from engaging directly with the "canvas" because it is only a preview image. In other words, the user must deal with an intermediary interface.

Minimal Requirements to Achieve Direct Engagement

Hutchins establishes four requirements a system should meet, as a minimum, in order to achieve a feeling of direct engagement:

1. Execution and evaluation should exhibit both semantic and articulatory directness.

2. Input and output languages of the interface should be inter-referential,

[1] See http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/mspaint_overview.mspx for more information.

allowing an input expression to incorporate or make use of a previous output expression. This is crucial for creating the illusion that one is directly manipulating the objects of concern.

3. The system should be responsive, with no delays between execution and the results, except where those delays are appropriate for the knowledge domain itself.

4. The interface should be unobtrusive, not interfering or intruding. If the interface itself is noticed, then it stands in a third-person relationship to the objects of interest, and detracts from the directness of the engagement.

2.3 WYSIWYG Document Editing

A WYSIWYG editor provides an environment where the view displaying the document being edited (what you see) very closely matches the target output (what you get). WYSIWYG editors are a breed of direct manipulation systems, they have minimal semantic distance on the output side [19].

The first WYSIWYG editor is considered to be Bravo [25]. Bravo was an experimental document editing program developed by Xerox for a system called Alto. Later, Xerox adopted the WYSIWYG idea into their commercial system called the Star Workstation [31]. The Star Workstation was built upon a set of design principles. Xerox considered "What You See is What You Get" as a design principle in itself — acknowledging that it greatly simplifies the editing process.

Before the time of Star Workstation's release, document editors used a markup language to specify formatting and layout of a document. The markup was written by users in a plain text editor environment. These documents were then compiled into the target output. This way of editing required the users to remember the commands/markup language in order to achieve the desired formatting for their documents. Such system were strongly based on the conversation metaphor.

While other WYSIWYG editors were eventually developed after Star Workstation, they generally did not simplify the document publishing process as well as Star Workstation [21]. For example, Star Workstation had the ability to render mathematical formulae directly within the editor as is would appear when printed. Other editors would only display mathematical formulae in a markup language, for example, $sqrt(sigma(1, n, (x * 3)/2)))$ to represent the formula that would eventually be printed, requiring several print-edit cycles to get the presentation right.

2.3.1 Office Documents

WYSIWYG editors for office documents, like Microsoft Word,[2] have come a long way. Today, it is relatively effortless to create and print a document containing rich formatting. The edit views of the documents are so close to the target output, that in the general case a user does not have to go through a series of print-edit cycles to get the presentation right.

The intended output of an office document is not always as a hard-copy. Increasingly people tend to share documents in digital form. In these situations there is no notion of a "published" view of a document because the document is always displayed in the same view: the edit view.[3]

2.3.2 HTML Documents

HTML (Hyper Text Markup Language) documents on the web will always have a published and edited view of a document. Unlike office documents, the target output of an HTML document is not locked down to a single program, but instead is broadcast to a range of hardware devices, operating systems and web browsers. For example, a document can be downloaded on a small screen device using Opera Mini version 4 to render it, or it can be downloaded on a PC with a high resolution screen using Microsoft's Internet Explorer version 7 to view it. The two representations will typically be quite different.

[2]See http://www.microsoft.com/word for more information.

[3]Unless a document is imported in a different WYSIWYG editing program and formatting is lost during the conversion.

HTML layout engines used in web browsers render web pages slightly differ from each other. Each engine is riddled with quirks causing violations of the W3C standards. Some engines can completely miss the mark when adhering to the W3C standards, where in some cases the presentation of a web page can look significantly different when viewed across different web browsers. For example, even though Internet Explorer 6 was developed to support the CSS 2 W3C standards, it does not support the CSS 2 fixed positioning scheme. In every other web browser that supports CSS 2, fixed elements would render at a position relative to the web browsers window, whereas Internet Explorer 6 would render them using static positioning.

To add to the complexity of producing cross-browser web documents, for each version of a web browser, both the release versions and operating system versions, it is common that new bugs/quirks are introduced and/or old ones fixed.

Full Page Editors

Many web page editors, like the popular Adobe Dreamweaver,[4] attempt to automatically address cross-browser issues for the users, but do not guarantee consistency amongst all web browsers. Because of the large number of combinations of web browsers, versions and target OS platforms, WYSIWYG editors do not guarantee that the view which the users see while editing their documents will match what everyone will see on the web.

Editors like Dreamweaver are useful for creating static web pages and designing the presentation aspects of a dynamic web page, but they are cumbersome to use when editing content within a composite web page. For example, when a user is browsing their website and discovers an article they wish to edit, they may have to perform the following tasks:

1. Login to the web server's file system that is hosting their website using a FTP (File Transfer Protocol) program or web service.

2. Find the file in the web server's file system which contains the article.

[4]See http://www.adobe.com/products/dreamweaver for product web site.

3. Download a local copy of the file containing the article.

4. Open the file in Dreamweaver.

5. Locate the place in the article of where to make the edit.

6. Make the edits.

7. Save the changes of their local copy.

8. Upload the new version of the article file via FTP.

The CMS in the example above stores article content as self-contained files. The process becomes more complex when using images in the article content, since the user must upload the images to the web server's file system and work out the relative URLs needed to link them. Clearly this process of editing is not efficient, nor is it user friendly because not only does it take a large amount of steps to simply update content, but it also requires the users to posses a range of technical skills.

Composite Page Editors

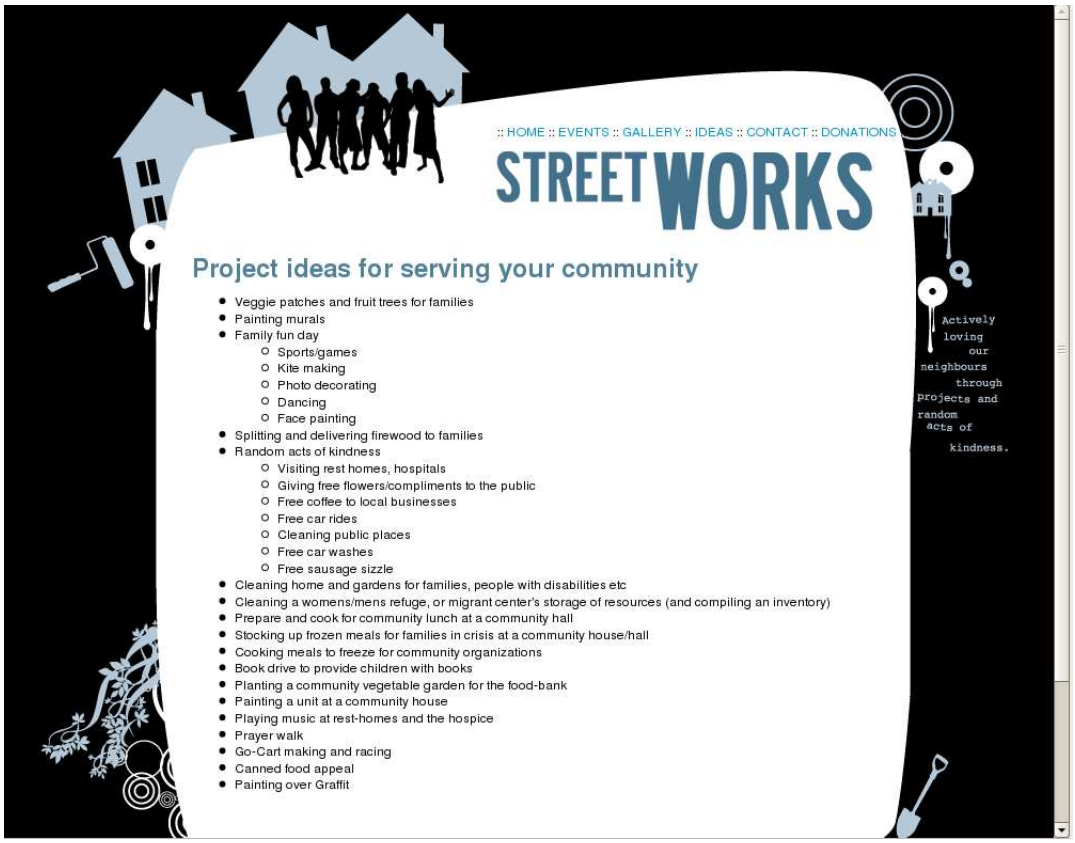
Today there are a vast range of well supported WYSIWYG editors which operate in most web browsers (for example TinyMCE[5]). Many content management systems[8] use these WYSIWYG editors as an internal editor. Unlike external editors, internal editors are accessible directly from a web browser — providing the functionality to easily create and update content.

Figure 2.4(a) is a web page which is generated from a CMS called Joomla.[6] Figure 2.4(b) shows the internal editor used for editing the main content of the web page. To edit the web page shown in Figure 2.4(a), a user would have to perform the following steps:

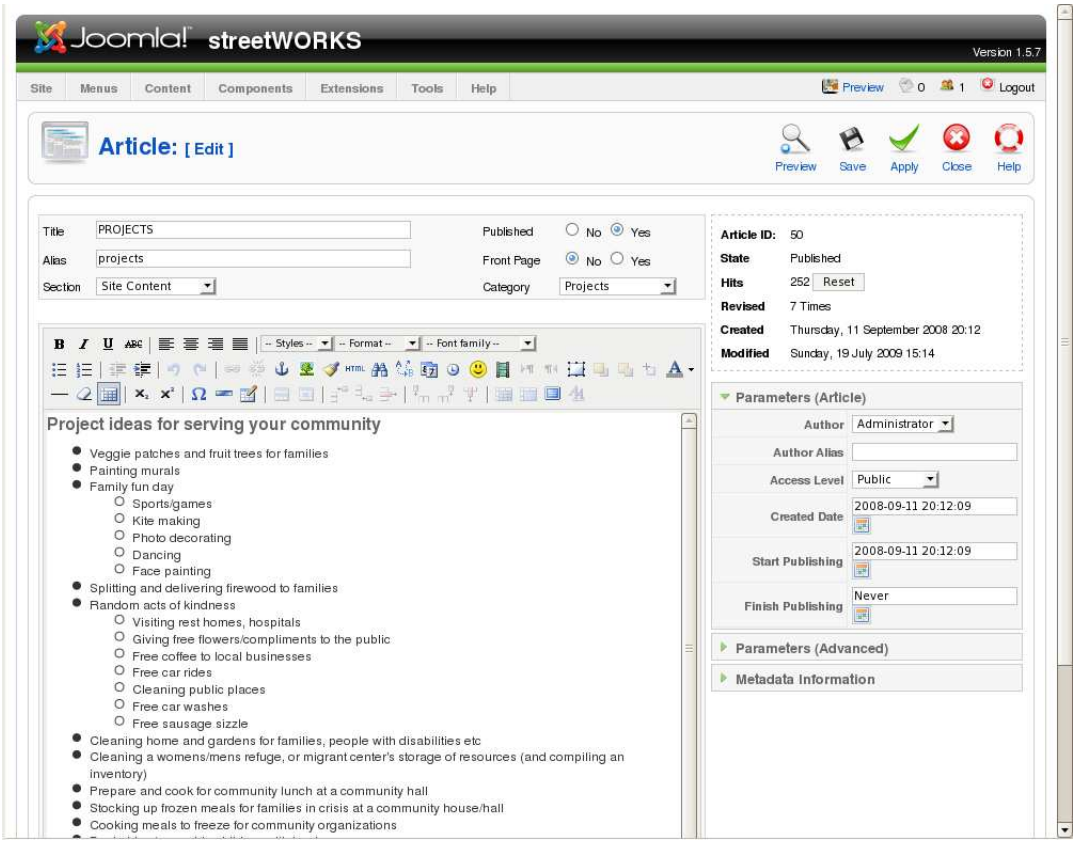
1. Login to Joomla's administrator site.

2. Locate the article called "Project ideas" in the *article manager*.

[5]See http://tinymce.moxiecode.com for project web site.
[6]See http://www.joomla.org for project website.



(a) Published view.



(b) Internal WYSIWYG editor view.

Figure 2.4: Editing article content with Joomla.

3. Click the edit button next to the article.

4. Locate the place in the article (presented in Figure 2.4(b)) of where to make the edit.

5. Make the edits.

6. Click the "save" button.

This editing process is more efficient than using an external editor to edit content since it require less steps. Also, internal editors require less skills/knowledge to edit content.

2.4 Authoring the Web

Authorship on the web has continuously evolved as web site editing technologies have advanced. This section establishes three authoring paradigms of the web, bringing to light the original visions for the web, and how the conceptual roles of the contributor and consumer on the web have blurred over time.

2.4.1 Web Authoring Paradigms

1990 - Late 1990's: The Contributor or Consumer Web

The Birth of the World Wide Web Tim Berners-Lee is the creator of the web [17], a distributed hypertext system over the Internet. The design of the web was influenced by Vannevar Bush's Memex [10], the first design of a hypertext system. The web shared fundamental ideas with Ted Nelson's Xanadu[27], a hypothetical global hypertext system which was independently designed.

From 1980 Berners-Lee began pursuing ideas of the web. In 1980 he joined CERN. At this time the Internet and hypertext had matured and Berners-Lee married them together, forming the World Wide Web [5]. By 1990 the web was released, and the digital information age began.

The World Wide Web's First Steps The web in the early 90's was a formative period of development with the advancements focused on the improving the underlying supporting network (the Internet), developments of the HTTP protocol, the HTML standard and web browsers. The aim during this period was to make the web globally accessible, fast, and easy to browse. The issue of global authorship for non-IT professions was largely ignored.

The roles of contributors and consumers of the web were clearly distinct when the web began. Authors of the web required knowledge of the HTML language and a set of technical skills which allowed them to set up their web sites and create/edit web pages.

Late 1990's - 2004: Contributor and Consumer Web

The Rise of Wikis In 1994 a computer programmer named Ward Cunningham developed a system called WikiWikiWeb (shortened to Wiki, the Hawaiian word fast). A Wiki is a collaborative system making it easy for people to contribute content into a central repository. The central features of a Wiki allow a user to create, edit and delete pages all within a web browser via a highly accessible editor [12].

Subsequently, Cunningham and Leuf establish six types of Wikis to describe how open a Wiki is for people to edit [23]. The most exclusive type starts at "personal", where access is restricted to a private computer or network. The most inclusive type is *fully open*, where anyone can both read and edit any article.

Fully open Wikis have blurred the lines between contributors and consumers: anyone reading information on a Wiki can instantly become an author since they can easily edit the very content they are reading.

Soon after the first publicly available Wiki in 1995 [14], Wikis became popular and revolutionised the web. Wikipedia[7] is by far the

[7]See http://en.wikipedia.org/wiki/Wikipedia for more information.

best known example, an online encyclopedia open to everyone for contributing information. It began in 2001 [1] and gained almost instant success. The large influx of participants in the Wikipedia project has played a significant part in increasing the number of contributors on the web [22].

The Rise of blogs Another significant player in blurring the lines between contributors and consumers was the emergence of *blogs*, also referred to as *weblogs*.

A blog is a website containing a list of dated posts submitted and maintained by one or more authors, referred to as *bloggers*. Posts may be in the forms of a combination of text, images or video — where the content typically includes current events or personal commentaries.

It is difficult to pinpoint when blogs actually began. In [4] it notes that technically they began from day one, when Berners-Lee used his website (on the very first web server) to keep people abreast of the other websites and servers within their research institution.

The origins of the term "weblog" can be traced back to 1997, when Jorn Barger used the term to describe his website: one which "logged" his Internet wanderings [38]. The concept of a weblog became more clearly established throughout 1998 [7].

It was not until 1999 that a "big bang of sorts occurred" [4] when the first set of blog tools were publicly released. Blogs quickly became popular, transitioning from a world of blogs owned and maintained by Internet technology and programmer professionals/enthusiasts, to a large community including non computer network-savvy bloggers. The global blog community is referred to as the *blogisphere*.

The first tools for blogs released were Pitas[8] and Blogger.[9] These tools automated the creation and updating process of web pages and links whenever a user wanted to create/edit a blog entry. This

[8]See http://www.pitas.com for more information.

[9]See http://www.blogger.com for more information.

significant simplification of owning a blog meant that users did not have to understand how to manage files on web servers, nor did they need to understand HTML syntax.

Blog systems abolished the technical obstacles posed on non web-savvy consumers to become dedicated authors of the web, playing a significant part in blurring the lines between contributors and consumers of the web [24].

2004 - present: Social Web

Transitioning from web 1.0 to 2.0 The idea of the web entering a new version, from 1.0 to 2.0, came from a conference involving O'Reilly and MediaLive International in 2004 [28]. The term web 2.0, coined by Tim O'Reilly, is used to mark an era of advancement for the web. It is difficult to define what the web 2.0 encompasses. Generally, it refers to a collection of design principles and technologies for the web which were absent in the 1990's, and embraced from 2004 onwards.

In [37], it describes the arrival of web 2.0 as a merging of three streams of development: the *applications stream*, which had brought along a number of web services anybody could use on the Internet and the web. The *technology stream*, which had fed the transition to web 2.0 with fast moving/comprehensive advances in networking and hardware technology and quite a bit of progress regarding software (notably AJAX). And the *user perception and participation stream* which has changed the way in which users, both private and professional ones, perceive the web, interact with it, and publish their own information on it.

The latter of these development streams, also commonly referred as the *socialisation stream*, was a huge step toward making the web an editable environment. From the success of Wikis, blogs and social networking, a need for users to contribute to the web was realised.

As a result, a set of design principles were defined to make the web a more user-friendly environment in which everyone could participate.

The Rise of Online Social Communities In 2003, social networking websites became mainstream [15]. Core features of a social network include the ability to construct a public or semi-public profile within a bounded system, articulate a list of other users with whom they share a connection, and view and traverse their list of connections and those made by others within the system.

Many social networks give users the ability to author content on the web. For example, Facebook,[10] a popular social network launched in 2004, gives its users the ability to customise their profile which can be viewed by other friends. Users can also broadcast small status updates to friend networks and create/participate in ongoing discussions.

A key success factor of Facebook is that it is simple to author content on the web. Once logged in, an empty text box is displayed, inviting users to submit a status posting. All users have to do is type what is on their mind then press enter (or click a nearby submit button) to publish their thoughts on the web.

Over the last two decades the web has matured from a consumer orientated world, were authorship was exclusive to people armed with technical knowledge and skills, into an open platform that invites everyone to participate in authorship. The growth of online communities: the blogisphere, Wikis, and social networks, is an indication that there is a need for people to author content on the web. A key factor which has helped the two roles converge is the simplification of the editing/publishing process. The central work of this thesis focuses on simplifying the editing/publishing process even further than the systems discussed in this section.

[10] See http://www.facebook.com for more information.

Roles	Examples
Total separation	HTML Editors HTML Editors and converters Dynamic pages by server-side scripts Professional web tools
Separation with facilities for the authors	Drag and drop Stand-alone content pages Content Management System
Separation with external collaboration	Annotations External linking
Overlap of roles	Weblogs Wikis Browsers Editors

Table 2.1: Web authoring examples (reproduced from [20]).

2.4.2 A Taxonomy of Web Authorship

Vitali has established a taxonomy of the contributor and consumer roles on the web [20], relating them to web authoring scenarios, as shown in Table 2.1.

The first category reflects a clear distinction between the contributor and consumer roles. In this category, a lot of technical knowledge is required to author content on the web. Vitali describes this category as a *web reading environment* rather than a *web publishing environment*.

The second category involves all the situations where the two roles still have no overlap, but many functionalities are provided to simplify the authoring process. One example scenario would be a CMS: a tool used to simplify the editing process of web pages but exclusive to the website owners. By Vitali's definition, single user weblogs can fall into this category as they do not permit anyone to add or edit content to the blog (this relates to document content, not to annotations).

The third category includes readers in the authoring process, permitting them to enhance the content, however the customisation of the content is only partial. There still exists a clear separation of roles: readers cannot edit the actual documents hosted on such systems (unlike the authors, who can change

such content), but merely provide a commentary on the central material being viewed.

In the last category, Vitali notes that in some scenarios all users can be at the same time readers, authors and reviewers of documents. A Wiki is a prime example (and multi-user weblogs to some extent), where members can edit other authors documents, but non-member readers cannot. These systems provide a publishing environment for anyone to contribute to.

Vitali developed a system called IsaWiki that provides a global publishing environment. It is discussed in more detail in Section 2.5.3. One of the key factors in creating its sharable and customisable environments for the general public is the simplification of the editing process.

2.5 In-Place Editing

A new paradigm of web editing is slowly emerging: an editing model that allows users to edit content within the web pages they wish to change. This editing model has been termed as *in-place editing*, *live site editing* and *in-context editing* [29]. The in-place editing model is a marginal advancement towards simplifying the editing process on the web. However, as discussed in Section 2.5.4, it has room for improvement. This thesis focuses on *modeless editing*, which is based on a different editing model to in-place editors. Modeless editors share similar benefits to in-place editors, but do not suffer from problems caused by the reliance on switching between edit and published modes.

The antonym to in-place editing is *back-end editing*. Where users must interact with a separate interface, a back-end, providing the editing facilities to update content on a website. A Wiki is one example of a back-end editing system, where in order to edit an article the user is directed to a new web page (the back-end) providing a text editor such as a plain text box containing WikiML.

In-place editing shares the benefits of systems like Wikipedia. When a user is reading an article and discovers content they wish to edit, they do not need

to manually locate the specific document in a back-end system, since an "Edit" link is embedded next to the very content which they may wish to edit. When the edit link is clicked the user is automatically directed to the document of interest for editing. The term *surf-to-edit* has been used to describe this design principle.

In-place editing shortens both the gulfs of execution and evaluation compared to back-end editing, creating a high feeling of direct engagement. The following three sections explores three web-based systems that feature in-place editing, where the of the level of directness of each system is discussed.

2.5.1 Sparrow

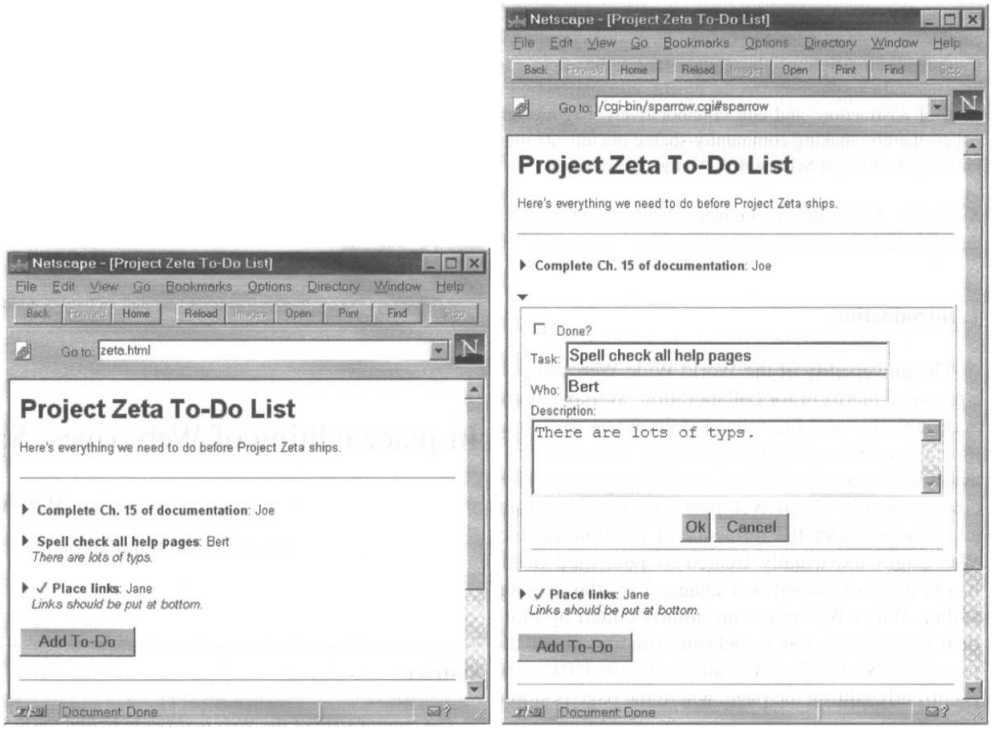
During the rise of Wikis and Blogs (before the web 2.0), a system called Sparrow was developed which hosts community driven websites [11]. Its key feature is a light-weight editing model, which as far as we know is the first in-place editing system for the web.

Sparrow is designed to facilitate community shared web pages. It shares similar community-based philosophies of a Wiki, but is distinct from a Wiki in that at the time of its development Wikis required contributors to have knowledge of HTML, where Sparrow does not.[11] Furthermore, Sparrow has a finer level of granularity of editing: where users can edit parts of a document as opposed to editing over a whole page as in a Wiki. Although Wikis have edit buttons at a section level of an article, users are directed to an editor containing the article's full content scrolled to the section that they want to edit.

Figure 2.5(a) displays a web page in Sparrow which implements a list for a hypothetical group project named Project Zeta. The Project Zeta To-Do List looks like a regular web page, except with added functionality. Clicking on a black triangle symbol causes the item to open into a dialog-box-like region to allow editing of the item (Figure 2.5(b)).

The opening of an item into an editable item occurs by a new page being

[11] Now there are Wikis which support WYSIWYG editors or a simplified markup language called WikiML.



(a) A Sparrow document.

(b) The same sparrow document with edit forms exposed.

Figure 2.5: Example of editing content with Sparrow.

sent to the browser. These new pages are not a back-end view of the web document however, and Sparrow helps maintain continuity by placing the form in the context of the rest of the web page, where it supplants the original item. When the new web page is displayed the scroll state of the browser is maintained via anchors. The contributor makes changes to the item by using the <*input*> text control (a standard web form element). Once changes are made, the user clicks the "Ok" button, Sparrow makes the change to the web page and redirects the user's browser back to the original URL, which then shows the newly altered page, scrolled to the appropriate location.

Discussion

There are two conceptual authorship roles in the Sparrow system: *page authors* are users who setup new Sparrow pages, and *contributors* who interact with the light-weight editing model. Although Sparrow simplifies the editing process, a lot of effort needs to be invested in setting up a new Sparrow web page.

Page authors must have knowledge of HTML and come to grips with Sparrow's markup language encoded within HTML comments to create *templates*. Templates are skeletal web pages that provide the structure and presentation of a document ready to be evolved by contributions from the Sparrow community.

The key design-principles of Sparrows light-weight editing model are:

Editing directly in the web browser. Contributors do not need to change to a different application and find the place in a file-system where the page is stored. Simply clicking on the page that is in view brings the editing facilities directly to the user (this concept was previously described as surf-to-edit).

Editing one item at a time. Contributors add/edit one item at a time. This finer level of granularity makes it easier for users to perform small changes. Sparrow is designed for occasional, incremental changes rather than large changes or many additions at one time.

In-place editing. During editing, the context of the rest of the page is retained. Edit regions are clearly associated with items being edited: an edit region is placed in a close proximity to the item being edited and is pointed to by the downward turn of a triangle graphic used to expose it. The page content surrounding the editing region remains unchanged and visible, so users can continue to browse the rest of the page even while editing.

No need to know (or see) any HTML. Contributors fill out text forms rather than see the HTML used to format the page content. Page authors pre-specify the formatting for Sparrow items.

Sparrow's light-weight editing model is a strong method for simplifying the editing process, but the user experience is hindered by its dependence on URL redirects. For every edit, users need to wait for the page to reload with the editing controls temporarily exposed, and then wait again for the page to reload once they submit their changes. The marginal response times are enough to make the editing experience cumbersome. Sparrow was developed

in the late 1990s, before AJAX technology existed and browser support for web standards was notoriously piecemeal at that time [40] so basing the implementation around <*iframe*> elements for communication would have been problematic at best or more likely impossible for supporting all browsers at the time.[12]

A re-implementation of Sparrow was later developed and renamed to SparrowWeb [6]. The time of its development was just before the establishment of web 2.0, so each edit still required a round-trip communication between the browser and the web server (involving URL redirects).

2.5.2 DirectEdit

Recently (2009) a CMS designed for small websites and small businesses called DirectEdit[13] was developed which features WYSIWYG in-place editing facilities to simplify the editing process [13]. Other content management systems that exhibit in-place editing features similar to DirectEdit are Adobe's InContext Editing[14] and concrete5.[15]

A DirectEdit document is broken down into sections which can be manipulated via a web browser. These sections are called DirectEdit elements for which there are five different types: Zones, Boxes, Fields, Images and Links. Figure 2.6 shows a screen-shot of a DirectEdit web page in administrator mode, exhibiting several types of DirectEdit elements. Fields are editable text areas which can contain formatting. Boxes are a combination of fields, images and links specified as an HTML template. The structure and formatting is defined by these templates to adhere to the CSS design principle of separation of presentation and content/structure. Boxes can be used as reusable blocks containing a common design that is repeated in a single web page.

Figure 2.7 shows an action sequence of editing textual content with DirectEdit. To edit textual content shown in Figure 2.7(a), the user must hover

[12]Before AJAX <*iframe*> elements were used as a hack to achieve client-server communication without URL redirects.

[13]See http://www.directedit.co.nz for project web site.

[14]See http://www.adobe.com/products/incontextediting for more information.

[15]See http://www.concrete5.org for project web site.

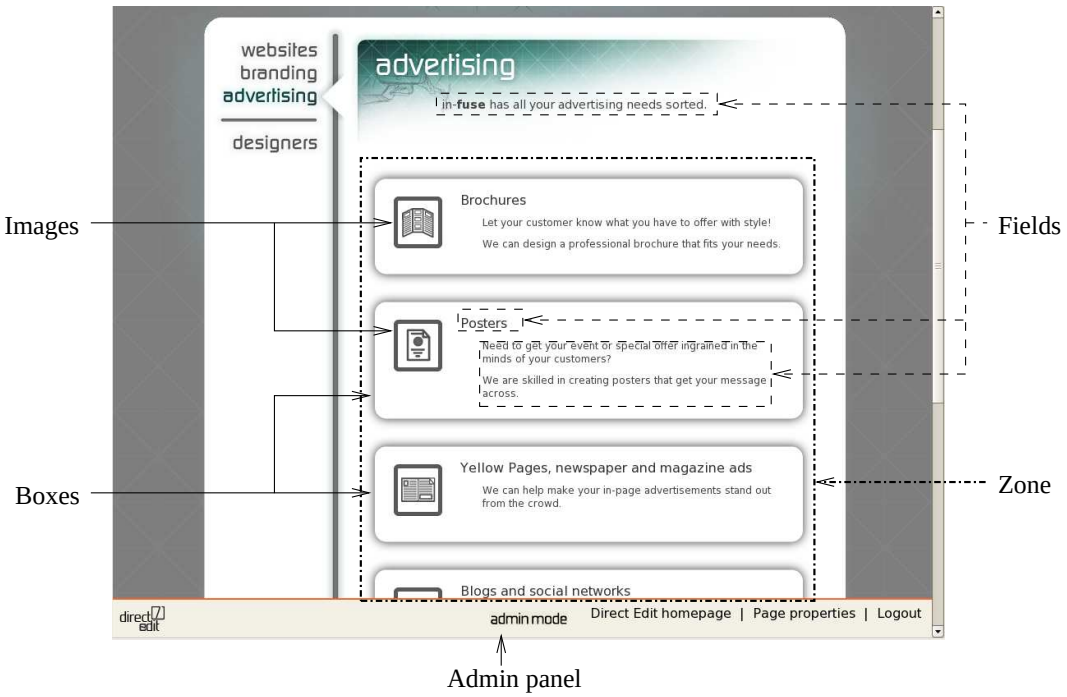
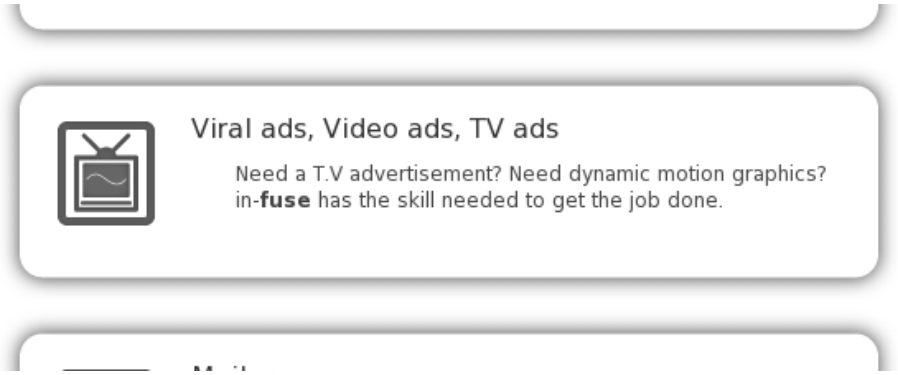


Figure 2.6: A DirectEdit web page in administrator mode.

the mouse over content marked as a DirectEdit field. After a small time period elapses a popup menu appears as shown in Figure 2.7(b). The user then clicks the "edit text" menu item, and the editable section is replaced by a WYSIWYG editor as shown in Figure 2.7(c). Once the user make their changes, they click the save button to accept changes. DirectEdit then removes the WYSIWYG editor and the new content is displayed as standard HTML, while the changes are submitted via AJAX.

A unique feature of DirectEdit is the concept of zones. A zone can contain a combination of boxes and other zones. Zones can let the authors re-arrange, create and delete inner-boxes. For example, Figure 2.6 contains a Zone (labelled) containing a vertical list of boxes. Each box can be deleted via a delete icon on the top-right position of the zone (that appears when the mouse hovers over the box, refer to Figure 2.7(b)). The boxes can be re-arranged by clicking and dragging them with the mouse, and then dropping them in the desired location within the zone. New instances of boxes following the same structure can be added by mouse to the end of the list of boxes then clicking a popup menu item called "add box".



(a) Step 1: identified text to edit.

(b) Step 2: invoke popup menu.

(c) Step 3: load in-place WYSIWYG editor.

Figure 2.7: Editing content with DirectEdit.

Discussion

Like Sparrow, DirectEdit's central feature is its light-weight editing model, but the creation of new documents is cumbersome. As with Sparrow, DirectEdit must be specifically tailored for each website via a template system, where authors of new pages must require HTML knowledge and/or experience with an external HTML editor. DirectEdit's use of AJAX for asynchronously saving content avoids issues with relying on URL redirects as discussed in the previous section.

2.5.3 IsaWiki

In light of the clear division between consumer and contributor roles on the web, a tool called ISA (Immediate Site Activator) was developed by Vitali to simplify the web authoring process [33].

The main idea of ISA is to exploit standard desktop tools for the creation of content and layout, and to employ a server-side application for the delivery of the final web pages. ISA is strongly moded due to its reliance on external editors such as Microsoft Word. It features a template system for marking editable sections of a web page similar to Sparrow and DirectEdit.

In a pursuit to revive the visions of Ted Nelson's Xanadu project, bringing global edibility to the web, a system called IsaWiki was developed [36, 35, 20]. ISAWiki's design stemmed from both ISA and a hypermedia system called XanaWord [34]. The template system used in ISA became automated via a system called elISA, which used heuristic methods to identify document content and design elements (such as navigational menu items of a web page).

Inspirations drawn from XanaWord were the ability to change content in a hypermedia system no matter who the original author might be and the support and management of versioned documents. ISAWiki was designed to co-exist with the web: users would install a plugin for Internet Explorer 6 or Firefox (older versions now only supported) which would provide a sidebar shown in Figure 2.8. When a user requests a new web page, the plugin interrogates an ISAWiki server to check if personal modified versions created by the

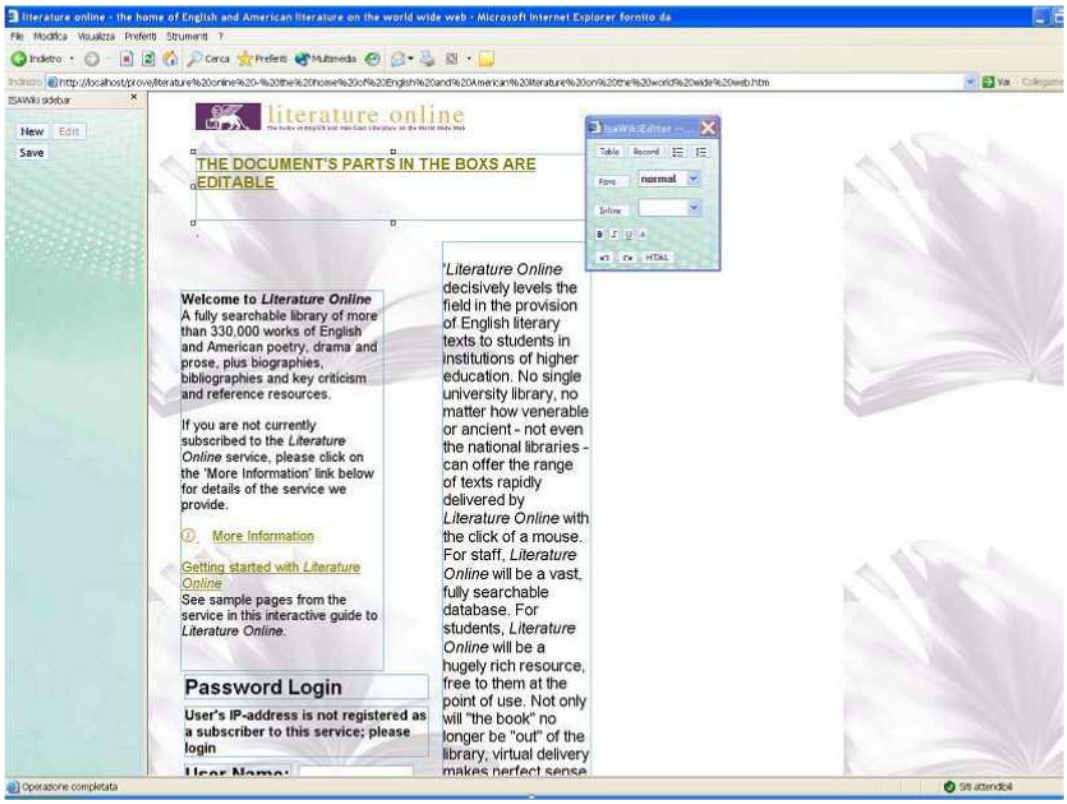


Figure 2.8: A screen-shot of an ISAWiki web page in edit mode.

individual of the requested URL exists. The server returns a list of modified versions, and the plugin displays the latest version available. The user can view other versions via a list on the sidebar which is displayed while the user is in view-mode.

To create a new version of a web page, that is, edit and create a personal version of any web page on the web, the user must click an edit button in the sidebar (shown in a selected-state on the top left of Figure 2.8). The browser then enters an edit mode, where a WYSIWYG editing toolbar appears as shown in Figure 2.8, and the document content (identified via the elISA sub-system) becomes editable. The in-place WYSIWYG editors maintain the exact CSS styles and layout. Once the user makes their change, they click a save button to save the new version.

2.5.4 Issues with In-Place Editing

Sparrow, DirectEdit and ISAWiki feature the idea of in-place editing, a method that is one step closer to simplifying the editing process. However in-place editing is not the pinnacle of simplifying the editing process for web pages: it is plagued with issues that detract from a direct manipulation experience as users switch to and from edit and view modes.

The following sub-sections identify three factors of in-place editing that attribute to the disorientation of users while they edit, described as *disconnects*. Disconnects interrupt continuity when both initiating the edit process and evaluating the results of the changes.

The Appearance Disconnect

The difference in the appearance between edit and view modes invoke cognitive effort to translate the content of interest between the two modes and can be confusing for the users.

Even though systems like DirectEdit use a WYSIWYG editor for in-place editing, the editing process does not benefit from the WYSIWYG principles, but instead uses the editors as an intermediary visual/rich editor. When the user invokes a WYSIWYG editor in DirectEdit, the CSS within the editor does not match the CSS of the published view of the content. The transition results in new text wrapping positions within the content they wish to edit, due to a change of size of the section containing the content, and the change of fonts. The combination of the change of text wrapping and fonts/colours makes it difficult for the users to make a connection between the two-modes. When a user switches to edit mode they have to perform another search within the new view of the content to relocate where they wish to make the edit.

In terms of direct manipulation, the appearance disconnect expands the semantic distance on both input and output sides of the interface language. In a scenario to illustrate this, consider a user using in-place editing to edit content in a web page containing many levels of heading elements. The user wishes to structure the content using headings in the same convention as sim-

ilar content surrounding the content that they are editing. The user is given traditional WYSIWYG interface with a list of formatted headings to choose from. However, because the heading CSS is different in the editor to the CSS of the surrounding content, the user does not know what levels to use. It might be that the first level is used for the main title/heading of the website, or an article heading. At this point, there is a clear gap in the semantic distance in the gulf of execution as the task of structuring the content is possible but cannot be described in a straight-forward fashion in the given interface language of the in-place WYSIWYG editor. The instant feedback given by the WYSIWYG editor is only temporal, the published view of the edits are only viewed after the changes are made and accepted. Thus extra cognitive process is required by the user to determine whether the published result matches what they intended.

In Sparrow there is less of an appearance disconnect than for DirectEdit because formatting is purely defined by page authors. Page authors specify the formatting for the Sparrow documents, and the in-place editors are placed directly next to the content of interest. The edits are at such a fine level of granularity that minimum cognitive effort is required to mentally make a connection between the editable content and the published content.

During the transition, while the in-place editors are being loaded, users are distracted by a sequence of loading states, and the final placement/area that the editors occupy does not map exactly to what the view displayed during the loading sequences. In Sparrow a blank document is observed when switching between modes due to a URL redirect. Sparrow then displays the new content with the exposed controls and scrolls to the editable content. In DirectEdit, the WYSIWYG editor has a different size to the area that the actual content occupies (the DirectEdit field elements). When switching to edit mode the WYSIWYG editor is initially rendered at a larger size than the content being replaced and the CSS of the formatting of the editable content briefly changes to the web browsers default styles. For example, the text may become larger since the web browsers default text size styles happens to be large than the web page's current style. Once the CSS for the WYSIWYG

editor is downloaded (and parsed), the editable content changes yet again. Soon after the GUI within the WYSIWYG editor area is loaded, the editable area within the editor becomes smaller. In some cases scroll-bars appear if the content is larger than the in-place editor. This issue is more apparent on both slower Internet connections and slower hardware/web browsers since the loading periods are drawn out. The temporary exposure of these changes of appearance further expands the semantic distance of directness.

ISAWiki avoids the appearance disconnect by maintaining the exact CSS styles and position of editable content when switching to edit mode.

The Interaction Disconnect

When the user locates where they wish to make an edit, whether it is fixing an error, adding new content, or removing existing content, in-place editors do not allow the user to directly edit the content right away. For example, they cannot place a blinking cursor by directly clicking in the desired location, instead they must first switch to an edit mode via an interface before they can carry out their task.

In DirectEdit the user must first replace the content with the WYSIWYG editor via a popup menu, then relocate the desired place they wish to edit (in the editor). Similarly, in Sparrow the user must click the arrow first to expose the editable areas then click the part they wish to edit. In ISAWiki, the user must click an edit button in the sidebar.

The input interface language requires a series of actions to switch to edit mode before allowing the users to carry out an edit task. Here the input semantic distance is expanded: the user must formulate "mode switching" actions as part of the input interface language to achieve an edit related task. The input articulate distance is expanded: the user must move, hover and click the mouse in order to switch to edit mode to achieve the task. The reliance on pop-up menus, and/or edit-mode buttons, violates the fourth requirement of achieving the feeling of directness of engagement (established in Section 2.2.2) since the intermediary interface elements are intrusive. For example, in the case of DirectEdit the user must interact with a popup menu to invoke an

editor.

Documents containing editable content within substantially sized sections can be troublesome for users. To edit content near the end of a large editable section, users may have to scroll back near the top of the section to find a button that switches to edit mode. Furthermore, once transitioned to edit mode, the user may have to scroll back down to where they wish to edit. The introduction of scrolling spans both articulatory and semantic distances.

The interaction disconnect becomes more problematic when performing multiple edits in different editable sections on a web page. For example, if a user decides to increase the size of the first occurring letter for each of the fields (except the titles) shown in Figure 2.6, the user must invoke the edit menu, switch to edit mode, make the edit then accept the changes, and do this for each field one at a time. The feeling of engagement is lost since the interface elements used to switch between edit and view modes are intruding multiple times when the user wishes to achieve a single goal over multiple editable sections. This violates the fourth requirement of direct engagement.

The Delay Disconnect

The Sparrow system suffers from poor response times due to using URL redirects for transitioning to and from edit and view modes. This violates the third requirement of direct engagement specified in Section 2.2.2, where there should be no delays during execution and results while carrying out tasks. Responsiveness is an important factor for creating and maintaining the illusion of direct engagement with a world of objects (in this case, pieces of content on a web page).

DirectEdit's response times are satisfactory thanks to its use of AJAX for asynchronous uploading. The feeling of direct engagement is maintained since saving changes processes in the background. However, the time needed to invest in the ceremony involved in switching to edit mode (the mouse hover to access the menu and selection of the menu item) can be enough to disrupt a user's line of thought.

2.6 Modeless Editing in Hypermedia Environments

The less distinction between edit mode and view mode, the simpler the edit process becomes. The disconnects identified for in-place editing in the previous section are all consequences of switching to and from edit and view modes. This section investigates hypermedia systems that have eliminated the edit/view mode distinction — the fundamental design principle upon which Seaweed is based.

2.6.1 Pyxi

Pyxi [39], a graphical browser and editor for the revived Xanadu project called Udanax, supports editing of content directly in a browser without switching to an edit mode. To distinguish between editing and navigational actions with the mouse, links can be followed by holding down the Alt key while clicking — if the Alt key is not down while clicking a blinking cursor is placed instead.

2.6.2 KMS

KMS (Knowledge Management System) [3] is a distributed hypermedia system viewed and constructed via a graphical browser. The browser has a first-person view of the underlying hypertext structure, displaying the actual nodes containing spatial content called *frames*. Frames are always editable via the keyboard and mouse. KMS exploits a three-button mouse to support a range of editing actions as well as navigational actions. For example, items are deleted by clicking the middle and right buttons together while hovering over them, and links can be followed by clicking on them using the left button. Thus the user never has to press any button or key to temporarily switch to edit mode — the environment is truly modeless.

KMS has no notion of a draft mode. Changes to frames are saved automatically in the background when a user navigates to a new frame. Of all the well-known hypermedia systems, KMS reaches the pinnacle of modelessness.

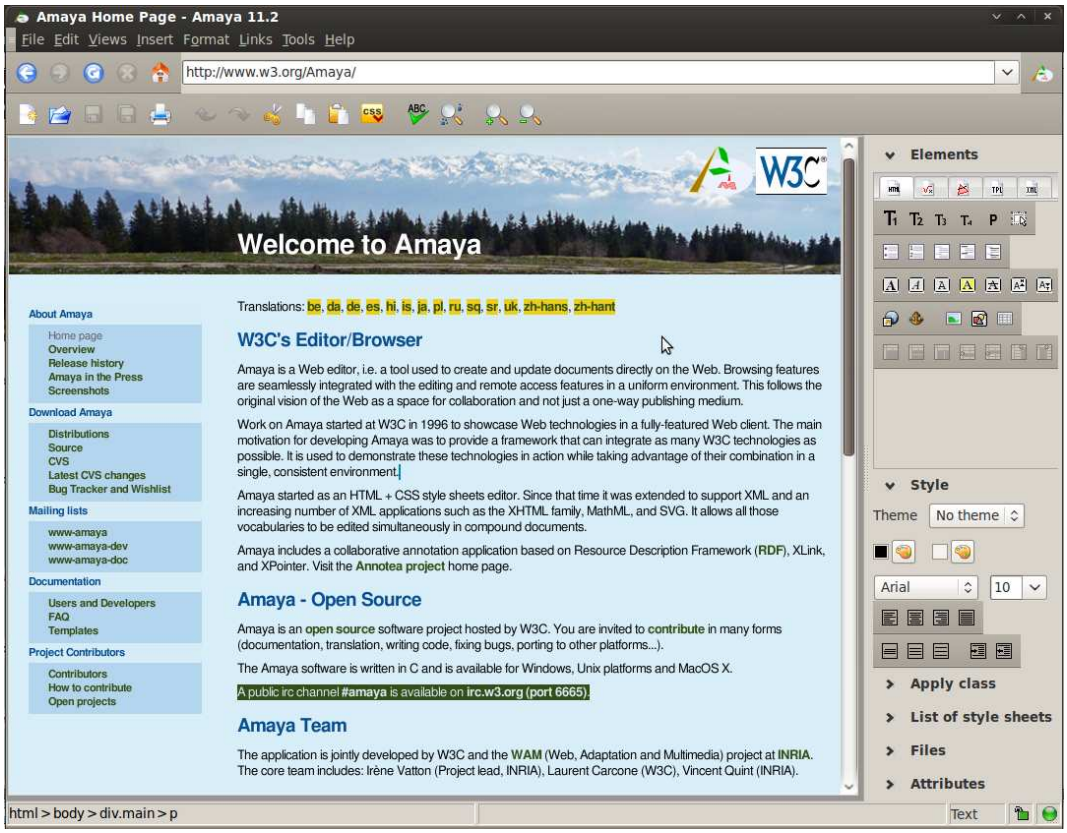


Figure 2.9: A screen-shot of Amaya.

2.6.3 Modeless Editing on the World Wide Web

There are many systems which strive for modeless editing, such as Platypus[16] and context-sensitive editors such as Lime and Bitflux (see [9] for a comprehensive list). However they are either poorly supported by common web browsers or are only partially modeless. The following two sections evaluate two fully modeless editing systems for the web.

Amaya

Amaya is a tool developed by W3C used to create and update documents directly on the web, where "browsing features are seamlessly integrated with the editing and remote access features in a uniform environment."[17]

Figure 2.9 is a screen-shot of Amaya. The interface contains features found in a typical web browser: the centre-stage is the rendering of HTML docu-

[16] See http://platypus.mozdev.org for project web site.

[17] Quoted from http://www.w3.org/Amaya, the project website.

ments, and a address bar is available at the top of the window. A WYSIWYG editing GUI is displayed on a panel on the right of the window. To edit a page, the user interacts with the document as if it were open in a document processor like Microsoft Word. In Amaya, the edit-interactions take precedence over navigational actions: to navigate to a web page via an anchor element users must double click the links.

Amaya edits pages at a full-page level. That is, it does not restrict the editing of certain parts of a document (such as navigational menus), allowing users to edit any part of the web page. Amaya is therefore geared for editing static pages, but is not practical for editing dynamic/composite pages.

Amaya supports the saving of live content. If the web server is setup correctly, and the user has authority to remotely save web pages via WebDav, Amaya will attempt to save an edited document to the web server that hosts the web page whenever the user actions to save their changes. If a remote save attempt fails, then a local copy is saved instead.

Amaya reaches the pinnacle of seamless editing for static HTML pages. One disadvantage is that it requires users to download a new web browser, an impractical solution for most community shared websites.

Mozile

The closest system related to Seaweed is Mozile a context-sensitive XHTML editor for web browsers.[18] Section 3.1.2 in the next chapter covers the technical similarities and differences between Mozile and Seaweed.

Initially Mozile was developed as a plugin for the Mozilla web browser. A CMS using Mozile would mark elements as being editable via a non-standard HTML attribute called *contentEditable* (see Section 3.1 for a detailed discussion of this attribute). Today there are no content management systems that use Mozile due to lack of cross-browser support. ISAWiki however used Mozile for supporting the seamless WYSIWYG editing in the Firefox ports of the plugin. Web browsers render these editable elements no different from non-editable content. When a user clicks on the element marked as *contentEditable*

[18] See http://mozile.mozdev.org for project web site.

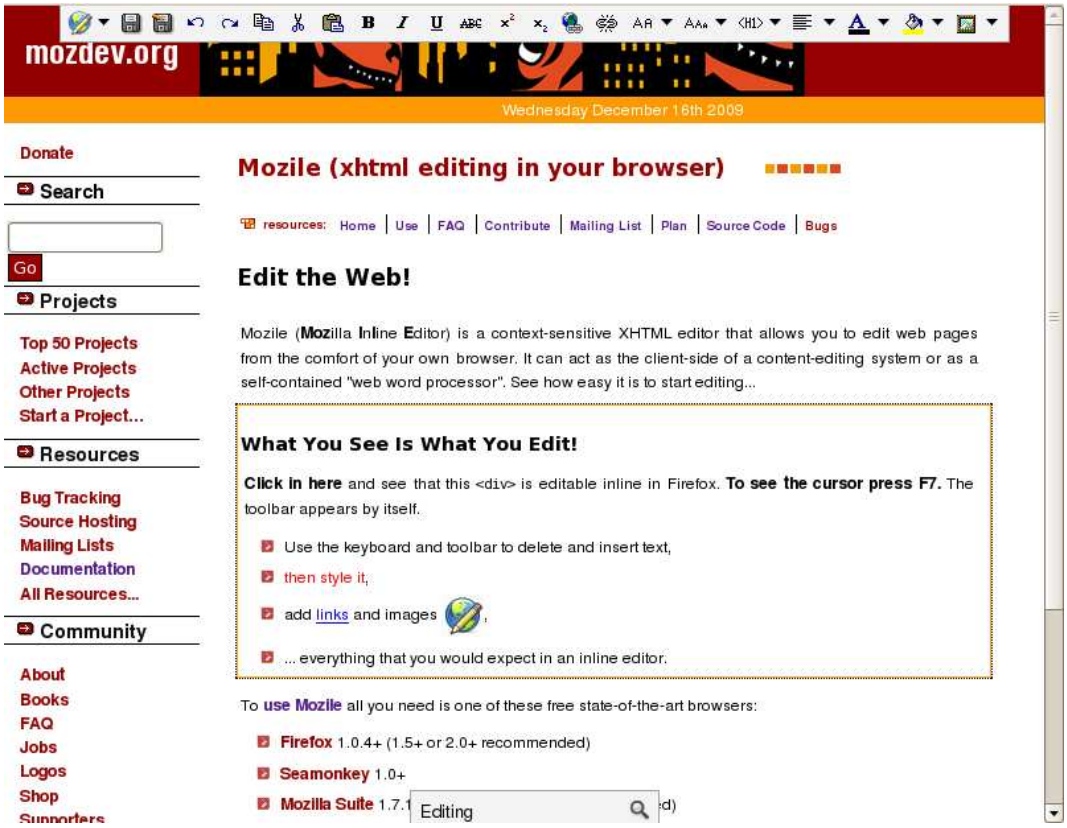


Figure 2.10: A screen-shot of a demo of the Mozile editor.

a blinking cursor appears accompanied with GUI as shown at the top and bottom of the web page in Figure 2.10. Users can begin editing content directly without having to switch to edit mode. The demo shown in the figure provides saving feature via a save button (displayed as disk icons on the top left). Once this button is clicked, a dialog appears showing the raw HTML source of the edited content, and gives an option to email the source. The mind does not have to stretch far to image a more seamless approach to the saving of data — AJAX would be an obvious technique to avoid delay disconnects identified in Section 2.5.4.

Mozile avoids the appearance and interaction disconnects by exploiting features shared amongst some browsers. It has various bugs and partial implementations, and is only supported on a small subset of web browsers available today. The project development on Mozile has now stopped.

2.7 Summary

This chapter began by visiting the fundamental concepts of a web page, a medium that is difficult for the public to modify. The principles surrounding direct manipulation systems were visited, which are the foundation for creating intuitive editing systems. The benefits and pitfalls of WYSIWYG editors, a type of direct manipulation system, were then discussed. A chronological and categorical view of the authoring processes for the web was presented, showing that the concept of seamless editing has yet to take its place in the consortium of authoring techniques for the web. The benefits and pitfalls of in-place editors was reviewed, evaluating three examples. Finally the concept of modeless editing was discussed, visiting two examples for hypermedia environments, and two examples for the web. There are no existing modeless editors for the web that are supported by all common web browsers today. The next chapter presents a system developed to support modeless editing on the web for all common web browsers.

Chapter 3

Seaweed Framework Implementation

This chapter walks through the design and development of the essential components that make up the Seaweed framework. Seaweed (Seamless web editor) is designed to work with any web-based system that features editing or creation of content, such as a CMS. It is a light-weight, unobtrusive JavaScript framework which provides seamless editing on any web page via any common web browser (Section 3.2.1 identifies the supported web browsers). The chapter begins an evaluation of native WYSIWYG facilities readily available in web browsers (Section 3.1), identifying the problems that lead to a need for developing the Seaweed framework. Section 3.2 presents a list of requirements which the framework had to support in order to provide modeless editing facilities, and is followed by a high-level overview of the framework. The remaining sections discuss implementation details for meeting the requirements established for the framework.

Before delving into implementation details, the following scenario presents a typical example of seamless editing using the Seaweed framework, showcasing its central features: a user visits the home page on their web site they created for car enthusiasts. The user identifies that a paragraph of text detailing the current status of their custom built car project is out of date. The user clicks into the paragraph containing the outdated text to get a blinking cursor. They begin updating the content as if it were open in a typical WYSIWYG editor,

so that it specifies the current status of their car project. They spend a small amount of time formatting the new content until they are satisfied with the overall style. Once they have made their changes, they press the `CTRL` and `S` key combination on their keyboard to save their changes. This example assumes the web sites CMS is enhanced to work with the Seaweed framework. The Seaweed framework only provides seamless editing features on the client-side (that is, the ability to directly manipulate content in a web page), integrating the Seaweed framework into a CMS is discussed in the following chapter.

3.1 Native WYSIWYG Editing Facilities

There are two HTML attributes supported by modern web browsers that enable users to edit web pages: *designMode* and *contentEditable*.

- The *designMode* attribute can only be set via JavaScript on the document element. When set to a value of "on" the entire web page becomes editable, and the user can place a cursor anywhere in the page and begin typing.

- The *contentEditable* attribute is like *designMode* but at a finer granularity, where certain HTML elements can become editable. Setting *contentEditable* to a boolean true value on a document's <*body*> element has the same effect as turning *designMode* on.

WYSIWYG Editors like TinyMCE and FckEditor use a combination of the *designMode* and *contentEditable* attributes to support WYSIWYG editing across all major web browsers. <*iframe*> elements are used with *designMode* turned on, and their <*body*> element with *contentEditable* set to *true*, so that the editors can be positioned within a web page.

In an editable web page, the web browser's native code handles all the editing functionality. However, the GUIs used for WYSIWYG editors are not native: they are developed as HTML elements and are managed by a JavaScript framework. To execute native WYSIWYG commands, JavaScript must use a native method in the DOM called *execCommand()*. For example,

when the user clicks a button to apply bold formatting to selected content, the framework must call the native bold command via the *execCommand()* method.

3.1.1 The Problem with Native WYSIWYG

The use of an <*iframe*> is the culprit behind the disconnects identified in Section 2.5.4. These elements do not inherit the CSS of the parent document. When in-place editors replace editable content with the <*iframe*> elements, the content's styling is changed, giving rise to the appearance disconnect.

In-place editors only display the <*iframe*> elements when the user requests to begin editing the content. If the <*iframe*> elements are in place all the time, the edit mode of a web page would look different to the actual published view. Furthermore, in cases with multiple editable sections of a web page, WYSIWYG editor frameworks do not provide a single GUI to manage multiple editors within one page. The requirement of requesting to edit content before actually editing the content increases the interaction disconnect.

Ideally, the *contentEditable* attribute could be used to achieve modeless editing. Internet Explorer 5.5 was the browser that first supported the *contentEditable* attribute. Nowadays most popular modern browsers support this attribute (except for mobile platforms). However, the attributes *designMode* and *contentEditable* are not part of any HTML specifications, only popular web browsers support one or both of these attributes. Because of the lack of specifications, there are countless discrepancies for the *execCommand()* method between web browsers.[1]

The HTML 5.0 specification drafts include the *designMode* and *contentEditable* attributes.[2] In the future all web browsers will strive to reliably and consistently support *contentEditable*. This thesis explores seamless editing by conducting experiments in real-world settings: where subjects use their own web browsers during the experiments. Consequently due to lack of support

[1]For example, see http://www.quirksmode.org/dom/execCommand.html for the *execCommand()* compatibility tables.

[2]See http://dev.w3.org/html5/spec/Overview.html for specification drafts.

for the *contentEditable* attribute, there was a need to write a framework to support modeless editing facilities on all popular web browsers.

In the future — once the HTML 5.0 specifications are established and fully supported by web browsers — the Seaweed framework can be used for legacy support. Furthermore, a JavaScript framework would still have to be developed to support an HTML 5.0 based equivalent of Seaweed since it would have to handle discrepancies/missing features between web browsers.

3.1.2 Technical Evaluation of Mozile

Section 2.6.3 describes a web-based modeless editor called Mozile. Initially Mozile was a web browser extension only supported by browsers based on the Mozilla layout engine. The last release switched to a JavaScript framework which also supports Internet Explorer 6. Seaweed strives for the same functionality as Mozile with support for all web browsers.

Mozile was developed before the Mozilla layout engine began supporting *contentEditable* (which only began support in 2008). For legacy support, and an attempt towards supporting other browsers, Mozile implemented all editing commands in pure JavaScript. In the last release of Mozile the *contentEditable* and *designMode* attributes are used to discover cursor positions and content selection, but not editing. Mozile's actions are buggy. For example, it has serious issues with handling white-spaces. Its functionality is also very limited and in the end it only reliably supported Mozilla-based browsers.

Seaweed follows a similar path to Mozile: avoiding cross-browser issues and supporting more web browsers by implementing a pure JavaScript framework for executing editing actions.

3.2 Framework Requirements and Overview

The following list specifies the set of functional requirements that the Seaweed framework had to fulfil in order to support seamlessly editable environments on the web:

- Define higher level entities to represent page content/elements.
- Manage input events from the mouse and keyboard.
- Manage placement of the editing cursor.
- Support selection.
- Support all common WYSIWYG editing actions.
- Support undo and redo.
- Support cut/copy/paste to/from the system clipboard.

The remaining sections of this chapter (after the overview) are organised such that the implementation details for meeting the requirements listed above are discussed in the respective order that they are presented in. Before delving into specific implementation details, the following section presents an overview of the Seaweed framework.

3.2.1 Web Browser Support

An essential non-functional requirement for the Seaweed framework was the ability to support all common web browsers that were used in the web during the time of development of the framework. It was important to support common web browsers since participants were observed using the framework in their own web browsers for evaluating the concept of seamless editing (covered in Chapter 5).

According to global web browser usage statistics from W3C and Net Applications,[3] the web browsers that were commonly used in the web during the time of the Seaweed framework's development were: Internet Explorer (version 6 and above), Firefox (version 2 and above), Safari (version 3 and above), Chrome (version 2 and above) and Opera (version 9 and above). A cross-browser JavaScript unit testing framework was written for testing the Seaweed

[3]See both http://www.w3schools.com/browsers/browsers_stats.asp and http://marketshare.hitslink.com/browser-market-share.aspx for archived statistics during June to August 2009.

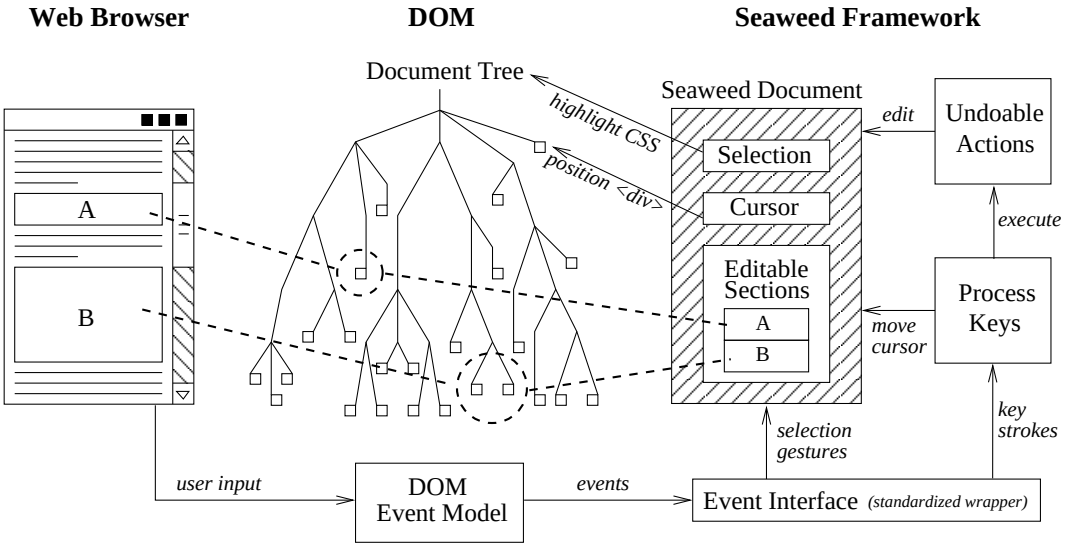


Figure 3.1: A high-level view of the Seaweed framework in relation to the DOM.

framework in order to support all of these commonly used web browsers. Appendix A presents the list of web browsers that the Seaweed framework ended up supporting, showing that not only all of these common web browsers were supported, but legacy versions as well.

3.2.2 Framework Overview

Seaweed is a client-side framework written entirely in JavaScript. It exploits the DOM to create an illusion that parts of a web page are directly editable just like a WYSIWYG editor. Figure 3.1 displays a high-level view of the Seaweed framework and how it interacts with the DOM. The figure shows three views of a web page: the rendered display in the web browser seen and manipulated by the user (on the left), the underlying DOM tree which is manipulated via JavaScript (in the middle), and an abstract editable view seen by the Seaweed framework (on the right).

A Seaweed document is broken down into a set of *editable sections*. Figure 3.1 depicts these sections in the page labelled as “A” and “B”, also showing how they are part of the DOM tree and maintained by the Seaweed framework.

Seaweed listens for all mouse and keyboard input events via an event interface: a sub-system that provides cross-browser event listening facilities. When

```
<h1 class="editable-title">
  New products arriving this summer!
</h1>
```

Figure 3.2: HTML Markup for statically declaring editable sections.

the user clicks into editable content, a mouse DOM event is fired, normalised by the event interface and eventually, if needed, the Seaweed document model's selection changes. When the user presses the "a" key, Seaweed interprets the key stroke and in effect will insert the letter "a" in the HTML document by changing the DOM — but only if the selection is within an editable section.

3.3 Seaweed Elements

There are three types of HTML elements which Seaweed uses for controlling edibility in a web page: *editable sections*, *place-holders* and *packaged elements*.

3.3.1 Editable Sections

As pointed out in Section 3.2, a web page can be broken down into editable sections. Users can seamlessly edit the inner contents of these sections. To declare an editable section, the standard *class* attribute of an HTML element are prefixed with "editable". Figure 3.2 gives an example of declaring an editable section for a heading within the HTML markup. The method of declaring editable sections via classes was used rather than inventing a custom attribute to avoid the need to use a custom DTD (Document Type Definition) or otherwise produce invalid markup.

As with the *contentEditable* attribute, Seaweed supports making a web page fully editable by assigning an editable class name to the document <*body*> element. Elements can be made editable "on-the-fly" via JavaScript as an alternative to the static declaration approach.

name	description
readableName	A human-readable name that describes the editable section.
phMarkup	HTML Markup used for editable section's placeholder.
singleLine	A boolean: true if the editable section should be a single line, false to be multi-lined.
actionFilter	A string which defines editable actions that can/cannot be executed on the editable section.

Table 3.1: Properties for editable sections.

```
seaweed.declarePropertySet("title",
  {
    readableName: "Title of the article",
    phMarkup: "<em>[Enter title]</em>",
    actionFilter: "SpellCorrect,SpellUnmark,SpellMark",
    singleLine : true
  }
);
```

Figure 3.3: Example of declaring a property set.

Property Sets

Every editable section belongs to a *property set*. A property set is a collection of properties that control the behaviour of the editable section. For example, whether or not the editable section can have multiple lines.

Each property set is referred to by a case-insensitive name, and can contain a selection of properties listed in Table 3.1. Property sets are declared via JavaScript as shown in Figure 3.3. To assign a property set to an editable section, the trailing part of the *class name* attribute after the "editable" prefix. For example, the editable section declared in Figure 3.2 would be assigned to the property set named "title" (hyphens are ignored).

Initially Seaweed contains a default property set which all editable sections use if (i) they do not have an assigned name, (ii) the property-set does not exist and (iii) the property within the assigned property set does not exist.

(a) Editable section with content.

(b) Empty editable section.

Figure 3.4: An editable section place-holder in action.

Figure 3.3 presents an example of declaring a property set named "title" via the Seaweed framework. The framework stores the property set for later reference when handling events for editable sections that the property set belongs to. An editable section using this property set would be single lined. The action filter only allows spelling actions (but in principle could be extended to allow other actions noted in Section 3.8.1). Standard text editing actions such as inserting text are implicitly allowed. Whenever an editable section using the property set becomes empty, a special place-holder containing the content "*[Enter title]*" would appear.

3.3.2 Place-Holders

There are two types of place-holder elements: *editable section* and *modifiable node* place-holders.

Editable Section Place-Holders

Editable section place-holders are elements that appear when an editable section becomes empty. An editable section may begin in an empty state, or later become empty due to the user deleting all the contents during a session. Editable section place-holders serve two purposes:

1. To prevent the HTML layout engines from rendering the empty editable sections either a smaller size, or not at all.

2. To be a clear marker for users to identify an empty editable section.

As listed in Table 3.1, editable sections can be assigned specific markup to represent as the place-holder. Figure 3.4 displays screen-shots of an editable

section place-holder in action. Figure 3.4(a) displays an editable section containing the text "Joe Blogs", which when deleted, the place-holder is revealed as shown in Figure 3.4(b). In this example, the *phMarkup* property contains CSS for changing the background colour as well as an <*em*> tag to italicise the editable section's place-holder.

Modifiable Node Place-Holders

HTML block-level elements within an editable section, such as a <*p*> element, can become empty. To ensure the HTML layout engine renders empty block-level elements, a modifiable node place-holder is inserted. These place-holders are <*span*> elements containing a single non-breaking white-space.

3.3.3 Packaged Elements

Multiple DOM nodes can be packaged into a single unit so that the user may not edit the inner contents. For example, an editable section may contain buttons which consist of a <*div*> and <*img*> element, as well as a text node. Encapsulating the three node tuple in a packaged element prevents the users from "tearing" them apart. Packaged elements are created by assigning the name "sw-packaged" to their *class* attribute.

3.3.4 Protected Elements

Some elements within editable sections should not be editable. For example, GUI elements used for aiding the editing process, and special hidden elements within a web page that are fully editable, should not be able to be edited or removed by the user. To avoid editing such elements, they can either be marked as being protected, or wrapped in a protected element. Elements are marked as being protected by assigning the name "sw-protected" to their *class* attribute.

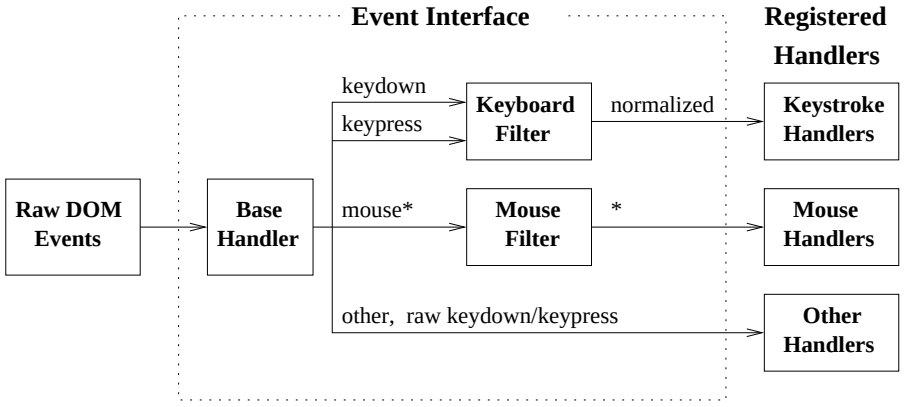


Figure 3.5: The event interface.

3.4 The Event Interface

There are many inconsistencies with the DOM event models amongst web browsers today. An event interface was written to overcome these inconsistencies. Many open-source JavaScript API's only provide cross browser facilities for registering to events (such as the Prototype API[4]), but do not provide enough information about the user input-related events.

Figure 3.5 displays a high level view of how DOM events are forwarded to registered event handlers. Filters are used for mouse and keyboard events to provide rich/reliable input data to event handlers. There is a base event handler which all events arrive at. The base handler takes care of passing event information and the consumption of events (if requested) in a cross browser fashion. The following sections explain the roles of the mouse and keyboard filters.

3.4.1 The Mouse Filter

All mouse events are passed to a mouse filter. In some web browsers events are raised on the web browser's scroll-bars, whereas others do not. The mouse event filter discards cases where events are raised on scroll-bars since the scroll-bars are not part of the editable document.

Mouse button states cannot be determined outside of a mouse event, for

[4]See http://www.prototypejs.org for project website.

example, whether the left mouse button is down or not. Mouse button states are tracked by analysing button-state information from every mouse event object passed through the filter. This makes mouse button state information available in any context via querying the mouse filter (that is, in events other than mouse related events).

3.4.2 The Keyboard Filter

Web browsers are known to have a large amount of inconsistencies regarding keyboard events. Every browser has its own way of identifying a key being pressed on a keyboard. Furthermore, some keys are raised on either or both of the *keydown* or *keypress* events.

The closest open source API to identifying keys for all popular web browsers is Qooxdoo API,[5] however it is incomplete. For example, Opera keys are not identifiable via this API. The Qooxdoo API's keyboard event handler was used as a starting point for Seaweed's keyboard filter. A custom event called *keystroke* was devised to unify the *keypress* and *keydown* events. For each *keystroke* event handler, the keyboard filter uses browser-specific maps which translate browser-specific numerical key identifier information (extracted from the event objects) to a human readable string. For example, if the user presses the delete key on a keyboard, a normalised event containing a key value of "delete" would be passed to all registered keystroke event handlers. The filter would also discard key events to avoid the keystroke listeners from receiving two events per key stroke (as depicted in Figure 3.5).

3.5 The Cursor

The cursor, sometimes referred to as the caret, is the cornerstone of the Seaweed framework. The cursor is the basis of the selection model. It provides points of references for knowing where to edit content, and it is the key element that creates the illusion that users are directly editing the content on the web page.

[5]See http://qooxdoo.org for project website.

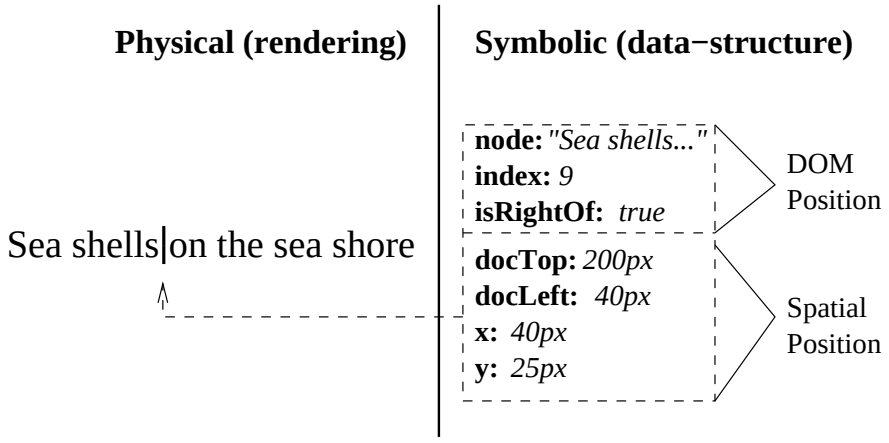


Figure 3.6: The physical and symbolic representation of the cursor.

3.5.1 Physical and Symbolic Representation

There can only be a single instance of a cursor in a Seaweed document at any given time. The cursor has two representations: its physical appearance and the symbolic information about its position in the document.

On the left of Figure 3.6, part of a web page is displayed with a cursor placed right after the word "shells". The data-structure shown on the right is the symbolic representation of the cursor. The data-structure contains information for both rendering the cursor (spatial position) and performing editing operations on the DOM (DOM position).

The physical representation of the cursor is rendered as a <*div*> element. Figure 3.1 indicates how it resides in the DOM: directly in the document's body, separate from the content. The <*div*> uses absolute positioning to be placed at any position over the editable content. By default the *z-index* CSS property is set to a higher index than the editable content to ensure that the <*div*> is rendered over the editable content. The colour of the cursor is set via the *backgroundColor* CSS property, and is set to the same colour as the foreground text which the cursor is placed in. By setting the <*div*> as the same colour as the foreground colour, as long as the text is clearly visible the cursor will also be visible.

The *visibility* CSS property of the <*div*> element is toggled between "visible" and "hidden" values periodically over time, achieving a blinking effect.

This creates a closer feeling of a standard document editor.

The <*div*> element is marked as a protected element to prevent cases such as removing the cursor <*div*> when a user selects all of an editable document and hits the backspace to delete all the contents.

3.5.2 Spatial Placement Algorithm

Cursor placement is restricted to editable sections to help users distinguish between the editable and non-editable regions of a web page. When a user clicks into an editable section, the cursor must be placed at the closest text cursor position to the mouse pointer.

Unfortunately the DOM does not provide any methods to determine cursor positions (unless *designMode* or *contentEditable* is used, which is not an option as described in Section 3.1.1). Thus, an algorithm was devised to determine cursor placements via mouse clicks.

Building Blocks

Microsoft's Internet Explorer 4 (1997) introduced attributes for all element nodes in the DHTML model that describes the physical dimensions of an element in a web page. These attributes are *offsetParent*; *offsetWidth*; *offsetHeight*; *offsetTop* and *offsetLeft*. Most, if not all, web browsers followed Internet Explorer's footsteps and the attributes are well supported in contemporary browsers.[6] The W3C drafts for the upcoming CSSOM (CSS Object Model) specification have included these properties[7] and thus must be supported by all modern web browsers in the future.

Figure 3.7 displays the top part of a web page that contains a <*p*> element inside a <*div*> element. The *offsetWidth* and *offsetHeight* attributes refer to the dimensions of an element in pixels (excluding margins). The *offsetTop* and *offsetLeft* attributes are the distances from the *offsetParent* in pixels. The *offsetParent* is the closest positioned containing element. For example,

[6]See http://www.quirksmode.org/dom/w3c_cssom.html for compatibility tables for the offset attributes.

[7]See http://www.w3.org/TR/cssom-view/#offset-attributes for specification drafts.

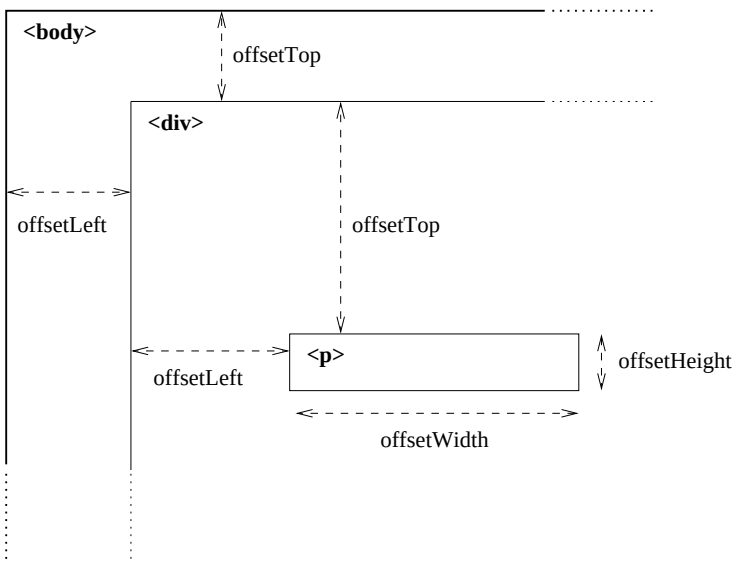


Figure 3.7: DHTML/CSSOM offset properties.

in Figure 3.7 the *offsetParent* of the <*p*> element is the containing <*div*> element. Note that the hierarchical parent in the DOM tree is not always the same as *offsetParent* due to CSS positioning schemes such as relative positioning.

The offset attributes can be used to locate the position of an element relative to the document. This is achieved by summing the *offsetTop* and *offsetLeft* attributes of all the *offsetParent* ancestry up to the document's <*body*> element. In some cases, depending on the browser, and whether it is in *quirks-mode* or not, the document's CSS border must be manually added (if one exists). To determine the position of an element relative to the window, the document's horizontal and vertical scroll-bar positions are subtracted from the calculated x and y positions of the element relative to the document respectively.

Another widely supported method is *elementFromPoint()*, which is currently included in the CCSOM specification drafts. This method returns an element in a document at the given (x, y) pixel coordinates. These coordinates are relative to the browser window for Gecko and Trident based browsers, where other browsers use coordinates relative to the document. This method is not as well supported as the offset properties (for example Firefox versions

below 3 do not support this), so an alternative JavaScript implementation was written for legacy browser support.

Total Isolation Approach

One solution for determining a cursor position from a mouse click is to encapsulate every character within an editable section in a <*span*> element. Thus, when a user clicks in an editable section, the *elementFromPoint()* method can be used to directly discover the clicked character. Since the characters are isolated in dedicated <*span*> elements, the position and size of the characters can also be determined using methods described in the previous section. <*span*> elements are chosen because they are the only element that can reside in any element where there is text (according to HTML 4.01 and XHTML 1.0 specifications). As simple as this approach appears, it has too many pitfalls to be regarded as a practical solution:

- The pre-processing step to isolate every character with <*span*> elements takes noticeably long periods of computation time. During this period, the browser becomes unresponsive.[8]

  The pre-processing phase must occur either when a page loads or on the first cursor placement. The former approach would thwart bursty/rapid navigation through seamlessly editable web pages. The latter approach would lose the illusion of direct manipulation since the response time for the first edit would be too long.

- The amount of memory used is bloated because of the large amount of DOM tree nodes required.

- The large amount of <*span*> elements create a large DOM tree. In general, the larger the DOM tree, the slower the performance of the browser. Manipulating the DOM for performing editing operations becomes more expensive.

[8]Except for Opera which runs JavaScript on a separate thread to the window event thread.

- The *elementFromPoint()* method becomes slower because there is a larger amount of nodes to search.

- The approach increases complexity in the rest of the framework's implementation. All operations which manipulate the DOM must ensure that all text nodes always have exactly one character that is encapsulated in a dedicated <*span*> element.

Four cursor placement algorithms that overcome the pitfalls outlined above were developed and tested. The goal was to discover the fastest algorithm, in order to satisfy the third requirement of direct engagement (see Section 2.2.2) so that, in the general case, users do not have to wait for a cursor to appear every time they click into an editable section.

The Search Space

When a user clicks on a web page, the (x, y) coordinates of the mouse pointer are supplied by the DOM in a mouse event object. The *elementFromPoint()* method is used to get the element which the mouse cursor is pointing at (if not already supplied by the mouse event object). A list of all the text nodes within the element is collected by traversing the element's DOM tree in-order. Text nodes where cursor placements cannot occur are excluded. For example, text nodes within <*script*> or <*style*> elements, or text nodes consisting purely of white-space for HTML source-code formatting purposes.

In a common case, the element which a user clicks on is a type of container element that is not fully visible by the user. These container elements are only partially visible due to the web browser's scroll-bar state only revealing part of the element, or the container element happens to be larger than the web browser's window. An extreme, but typical scenario where this occurs is when a user clicks near the edges of a web page just outside of a <*p*> element. The element actually clicked is the document <*body*> element, and therefore includes every text node in the entire web page as part of the search space. To minimise the search space to yield faster performance when searching for the nearest cursor position, the search space is capped to include only nodes that

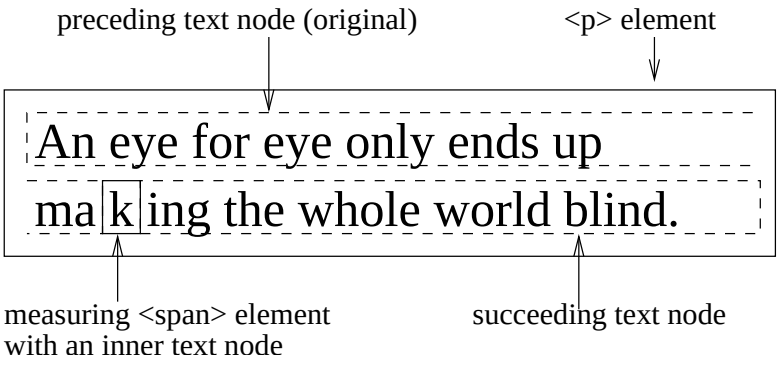


Figure 3.8: Nodes involved in measuring characters spatial properties.

are visible to the user.

All four cursor algorithms use the same method for collecting the search space. Each algorithm will now be described in turn.

Sequential Search

The first approach performs a naive sequential search over the entire search space. Every character in each text node within the search space is measured and compared.

Figure 3.8 conveys the logic used to measure a single character using a simple example. The example shows a <*p*> element that originally had a single text node, which has been broken down into four nodes used to measure the letter "k". A <*span*> element is inserted to the right of the text node (labelled as the original text node) where the letter to measure resides. Another text node (labelled as the succeeding text node) is inserted to the right of the measuring <*span*>. The original text node's content is then distributed amongst the nodes. The content up to the letter "k" remains in the original text node (which becomes the preceding text node). The letter "k" is stored in an inner text node within the measuring <*span*>. Lastly, the remaining content is stored in the succeeding text node. Thus the letter "k" becomes wrapped with an element from which spatial properties can be extracted using the methods previously described.

It is necessary to have a succeeding text node in order to maintain the text wrapping state. For example, if the remaining text after the letter "k" in

Figure 3.8 is not included, then the HTML layout engine would likely move the measuring <*span*> up into the first line since the content would be able to fully fit within the <*p*> element's bounding box.

The first measurement is considered as the *current closest* cursor position. Successive measurements are compared to the *current closest* measurement. If a measurement is spatially closer to the target (x, y) coordinate (which usually would be a certain mouse click position), then the *current closest* is set to the closer measurement. A measurement is considered closer to another if it is on a line closer to the target y coordinate. If the two measurements being compared are on the same line, then the measurement with an x coordinate (to the left or right of the measured character) closest to the target x coordinate is considered to be closer.

The nodes used for measuring are only inserted once per text node. For each character within a text node, the contents of the measuring nodes are simply shifted to isolate each character. This cuts down on the amount of DOM manipulations needed to measure each character.

The performance of this algorithm is considerably poor, to the point of being unusable. Manipulating the DOM is the most expensive operation because the layout engine must re-examine the DOM tree and perform various maintenance routines, such as re-evaluating the document's CSS based on the new DOM structure. Because the JavaScript threading model on all browsers, except Opera, executes scripts synchronously to the browser's thread for handling user input and rendering pages, the manipulations are not seen by the naked eye. For Opera — where rendering occurs on a separate thread — the DOM manipulations can be briefly observed by the user (in slow cases). The sequential search algorithm served as a base case to improve upon: subsequent algorithms cut down the amount of DOM manipulations to yield best performance.

Pin-point Search

The pin-point algorithm performs an informed sequential search that does not need to measure every character in the search space to discover the closest

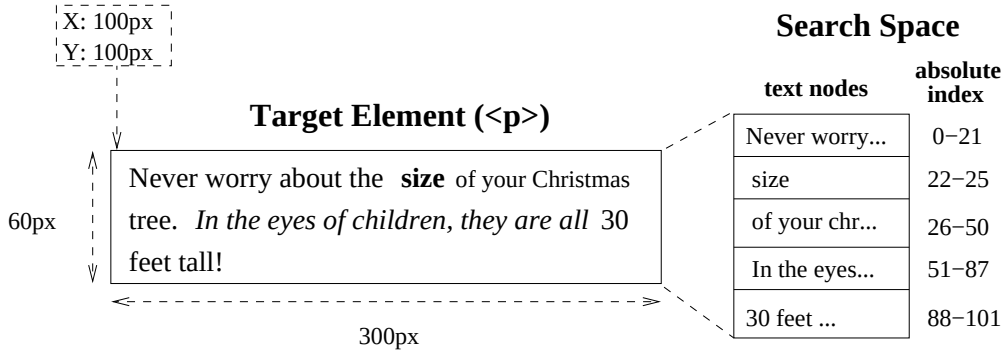


Figure 3.9: The pin-point algorithm's perspective of the search space.

cursor position.

Each character in the search space has a relative and an absolute index. The relative index is the character index within the character's text node. The absolute index is the index within the whole search space. For example, the character "s" in the word "size" shown in Figure 3.9 has a relative index of 0 since it is the first character within the text node where it resides, and an absolute index of 22 since it is the twenty second character within the whole search space.

The search begins by starting at the text node and character index that is estimated to be the closest to the target coordinate. If a user clicks at (x, y) coordinate $(130, 130)$, just before the period symbol (.) after the word "tree" displayed in Figure 3.9, the calculations presented in Figure 3.10 are used to estimate the starting point. Working through the calculations:

1. The "!" character is isolated and measured. The character is 16 pixels in height, 6 pixels in width and is at (x, y) coordinates $(154, 144)$.

2. Floor of 60 divided by 16 = 3 lines.

3. (154 + 6) - 100 = 60 pixels.

4. 2 (the second line of the 3 estimated lines).

5. ((3 - 1) * 300) + 60 = 660 pixels.

6. ((2 - 1) * 300) + (130 - 100) = 330 pixels.

```
1. Measure the last character in the search space.

2. Estimated line count = floor(target element height
  / last character height).

3. Last line width =
   (last character x position + last character width)
   - target element x coordinate.

4. Closest line number = line with smallest difference in vertical
   position to target y coordinate.

5. Unwrapped search space width = ((estimated line count - 1)
   * target element width) + last line width.

6. Unwrapped target x position = ((closest line number - 1)
   * target element width) + (target x coordinate
   - target element x coordinate).

7. Estimated absolute character index = ceiling(
     unwrapped target x position /
    (unwrapped search space width / search space size)
   ).

8. Estimate node/index = get-rel-dom-position(absolute index).
```

Figure 3.10: Calculations for estimating starting point.

7. Ceiling of 330 / (660 / 102) = 51.

8. Relative node/index of 51 = the 4th Text Node (the italicised text), at character index 0 (the letter "I").

The estimate of the starting point in Figure 3.9 is very close to the target coordinate. Estimations become less accurate as the search space increases in size because there is more chance that the amount of characters per line are not evenly distributed. An uneven distribution of characters per line is usually due to jagged text wrapping, or the variety of fonts and sizes of characters within the search space.

Once the starting point is discovered, the pin-point algorithm enters a search comprised of two passes. The first pass sequentially measures each character in the search space starting from the estimated start point toward the

left. As with the sequential search algorithm, a *current closest* measurement is updated at each new measurement. If a measurement is found to be directly at the target coordinate, then the search is completed (and there is no need to enter the second pass). Otherwise, the search continues until the start of the search space is reached or a measurement is found to be on a new line that happens to be at a further distance from the target y coordinate.

When the left search is finished, the second pass is entered. The second pass switches the direction of the search, beginning after the estimated starting point and measuring characters towards the right. The second phase follows the same logic as the first.

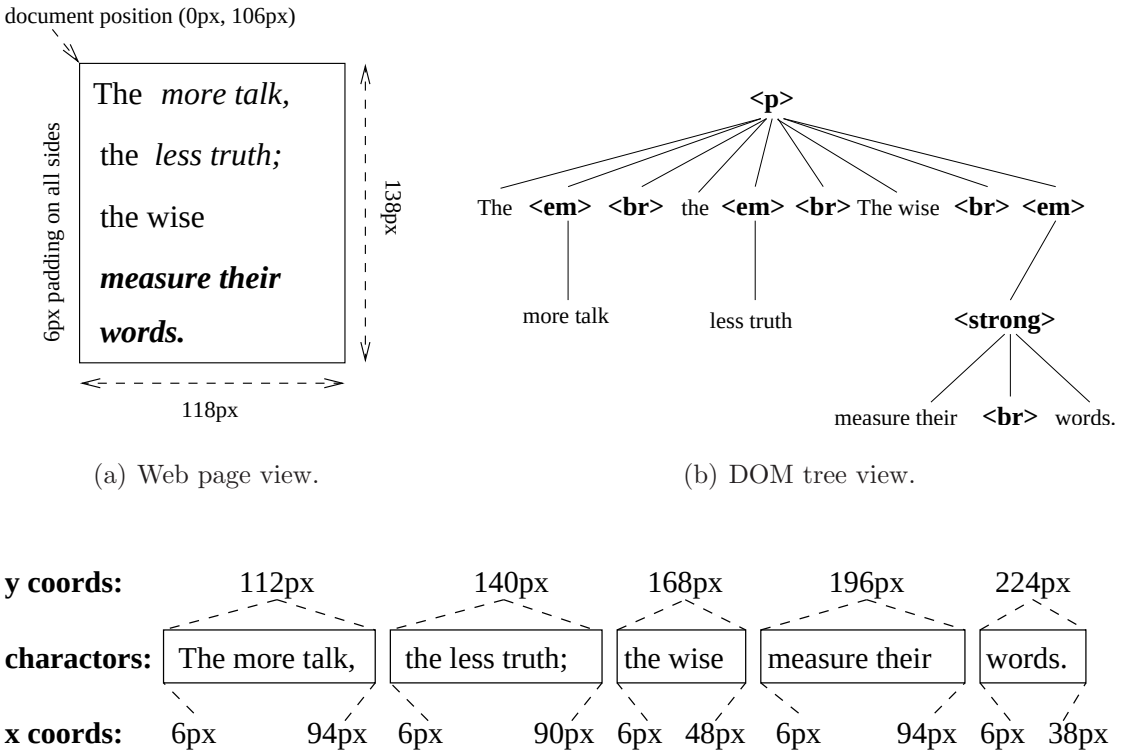
Returning to the example above in Figure 3.9, the first pass would begin by measuring the "I" character (the estimated starting point), becoming the *current closest* measurement. Moving to the left: the preceding white-space character, followed by the period symbol (.) would be measured. Each of these two measurements progressively gets closer to the target coordinate, thus setting the *current closest* measurement in each instance. If the mouse click was close enough to the period symbol, the search would instantly abort since the target coordinate lies directly within the bounds of the period symbol.

In a case with poor performance, if the estimate starting point is far off from where the target coordinate lies, the algorithm may have to measure several full lines of text in order to determine the closest cursor position.

Dual Binary Search

The binary search algorithm is an efficient algorithm for locating an item in a sorted collection. It turns out that a two-pass binary search can be used against the search space to discover the closest cursor position.

In the general case when performing an in-order traversal on a DOM tree, the nodes are visited such that each node is either visually below or to the right of the previously visited node. The search space is collected using an in-order traversal, thus forming a spatially sorted collection. Search spaces are readily organised in ascending order by the y coordinate in the web page, therefore the characters in the search space are grouped by the line where they reside.



(a) Web page view.

(b) DOM tree view.

(c) Search space view.

Figure 3.11: The dual binary algorithm's perspective of the search space.

Figure 3.11 conveys this grouping by surrounding characters on the same line with a box in the search space view of a paragraph of text. Furthermore, the x coordinates of the characters are in ascending order within each line-group. For example, the first character in the last group (the right-most group with text "words.") is at an x coordinate of 6 pixels, and each successive character's x coordinate increases by roughly 6 pixels all the way up to the last character on the line at an x coordinate of 38 pixels.

Two separate binary searches are used to quickly locate the nearest cursor position to the target coordinate. The first search performs a vertical position based binary search over the whole (y-ordered) search space to locate the line group which the target coordinate resides. The second search performs a horizontal position based binary search over the (x-ordered) line group to locate the nearest character to the target coordinate.

Figures 3.12, 3.13 and 3.14 present the binary search pseudo code used

```
find-cursor-at(target-coordinate)
{
  search-space = get-search-space(target-coordinate);

  start = measure(first character in the search-space);
  end   = measure(last character in the search-space);

  sample-set = [start,end];

  current-closest = closest(start, end, target-coordinate);

  if (target-coordinate within best.bounds) {
    return current-closest;
  }

  # First pass
  y-search(start,end);

  (lower, upper) = select-bounds(sample-set);

  # Second pass
  x-search(lower, upper);

  return current-closest;
}
```

Figure 3.12: Dual binary search pseudo code.

for the dual binary search. The search begins by measuring the first and last characters in the search space (Figure 3.12), becoming the lower and upper bounds of the search space respectively. If one of the initial measurements is displayed directly at the target coordinate, the search is finished. The first binary search is then entered (Figure 3.13).

Once the first binary search, the *y-search*, finds a measurement on the same line as the target coordinate, the search ends. The upper and lower bounds are selected from measurements stored in a *sample-set* (gathered during the *y-search*) for the second search, the *x-search*. The upper and lower bounds for the second search are determined as follows: if the *current closest* measurement is visually positioned to the left of the target x coordinate, then it becomes the lower bound and the upper bound is set to the closest sample in the *sample-set* with a larger absolute index to the lower bound. Conversely, if the

```
y-search(lower, upper)
{
  if (lower.abs-index) == (upper.abs-index - 1) {
    # Upper and lower bounds have met, no more characters to search
    return;
  }

  # Half way point between upper and lower samples
  current.abs-index = (lower-sample.abs-index + upper.abs-index)/2;

  # Get node and relative index from absolute index
  (current.node,current.rel-index) =
    get-rel-dom-position(current.abs-index);

  # Measure the sample
  (current.x, current.y, current.width, current.height) =
    measure(current.node, current.rel-index);

  # Store it in the sample set if doing a line search
  sample-set.add(current);

  if (current is closer to target than current-closest) {
    current-closest = current;
  }

  if (current is same line as the target y) {
    # Goto second pass: the x binary search
    return;
  } else if (current.y > target-y) {
    upper = current;
  } else {
    lower = current;
  }

  y-search(lower, upper);
}
```

Figure 3.13: First pass pseudo code: line binary search.

```
x-search(upper,lower)
{
  if (lower-sample.abs-index) == (upper.abs-index - 1) {
    # Upper and lower bounds have met, no more characters to search
    return;
  }

  # Half way point between upper and lower samples
  current.abs-index = (lower-sample.abs-index + upper.abs-index)/2

  # Determine node and relative index from absolute index
  (current.node,current.rel-index) =
    get-rel-dom-position(current.abs-index);

  # Measure the sample
  (current.x, current.y, current.width, current.height) =
    measure(current.node, current.rel-index);

  if (current is closer to target than current-closest) {
    current-closest = current;
  }

  if(current on line above current-closest) {
    lower = current;
  }
  else if (current on line below current-closest) {
    upper = current
  }
  else {
    # Current is on same line as current-closest
    if (current.x > target.x) {
      upper = current;
    }
    else {
      lower = current;
    }
 }

 x-search(lower, upper);
}
```

Figure 3.14: Second pass pseudo code: character binary search.

current closest measurement lies to the right of the target x coordinate, then it becomes the upper bound and the lower bound is set to the closest sample in the *sample-set* with a smaller absolute index to the upper bound.

Even though the upper and lower bounds selected for the second pass may contain characters on different lines, the search space is still sorted by distance. Figure 3.14 shows how samples found to be on different lines to the target y coordinate during the *x-search* are considered to be more distant regardless of their x coordinates compared to samples that lie on the same line at the target y coordinate.

Some white-space characters are not rendered due to the HTML layout engine collapsing the white-space. When a character is found to be not rendered (where the offset attributes are zero or non-existent), the algorithm sequentially searches in the left direction to discover the nearest occurring rendered character. If the search reaches the lower bound, then the sequential search switches direction and locates the nearest rendered character to the right of the sample. If no other characters are found that are rendered within the upper and lower search space bounds then the search is complete.

The dual binary search algorithm requires the search space to be sorted. There are exceptions to the in-order traversal method for creating a spatially sorted collection of DOM nodes. For example, CSS positioning allows elements to be manually positioned in a web page regardless of where they reside in the DOM tree. CSS Floats are another example that can lead to unordered collections. However positioning styles are usually used for the presentation and structure of a web page rather than actual content. Furthermore the algorithm works within manually positioned elements/floats since an in-order traversal within these elements is spatially ordered.

Homing Dual Binary Search

The homing dual binary search is the same as the binary search, except instead of halving the search space for each new sample, the search space is narrowed down to an estimation of the closest absolute index.

The estimation step follows the same logic as the pin-point search for se-

lecting the start point (outlined in Figure 3.10). The unwrapped width of the search space is the distance between the lower and upper bounds. No extra DOM manipulations are made in order to calculate the estimate absolute index.

Algorithm Performance Analysis

To determine the best suited cursor placement algorithm, a statistical analysis of the algorithm performances was conducted. A statistical approach was necessary because the performance of each algorithm is dependant on the state of the search spaces. For example, the pin point algorithm may perform best in situations where the text in the search space is wrapped evenly, and the text size and font is the same. The homing dual binary search might perform better than the dual binary search if the state of the text wraps give accurate estimations of where to best narrow the search space.

A benchmark package was written to measure the performances for each of the algorithms on a range of different web browsers. The benchmark ran each algorithm over sampled (x, y) coordinates in a web page. The web page contained a variety of HTML elements and CSS styles to provide representative test data for general use, including common layouts used in WordPress blogs — the environment that the Seaweed framework was utilised for evaluating seamless editing. Samples were taken at even x and y spacings in the page, auto-scrolling through the document until the end is reached.

The benchmark collected the search space at each sampled (x, y) coordinate. For each search space collected, the benchmark ran all four algorithms and recorded the times taken for each algorithm to present an answer. Using the same search space for each algorithm produced comparable results between them.

The benchmark could run in any common web browser except for Internet Explorer — which could only run the benchmark on smaller scale tests. During large tests (which typically take about 30 minutes to run on other browsers), all versions of Internet Explorer would freeze without any warning/error mes-

sages.[9] The small scale test results for Internet Explorer resembles the same performance trends discovered in the full analysis as described below (see Appendix C for an example of raw test results for Internet Explorer). These results were excluded from the full analysis because they did not provide enough data to be fairly averaged with the full-scale results.

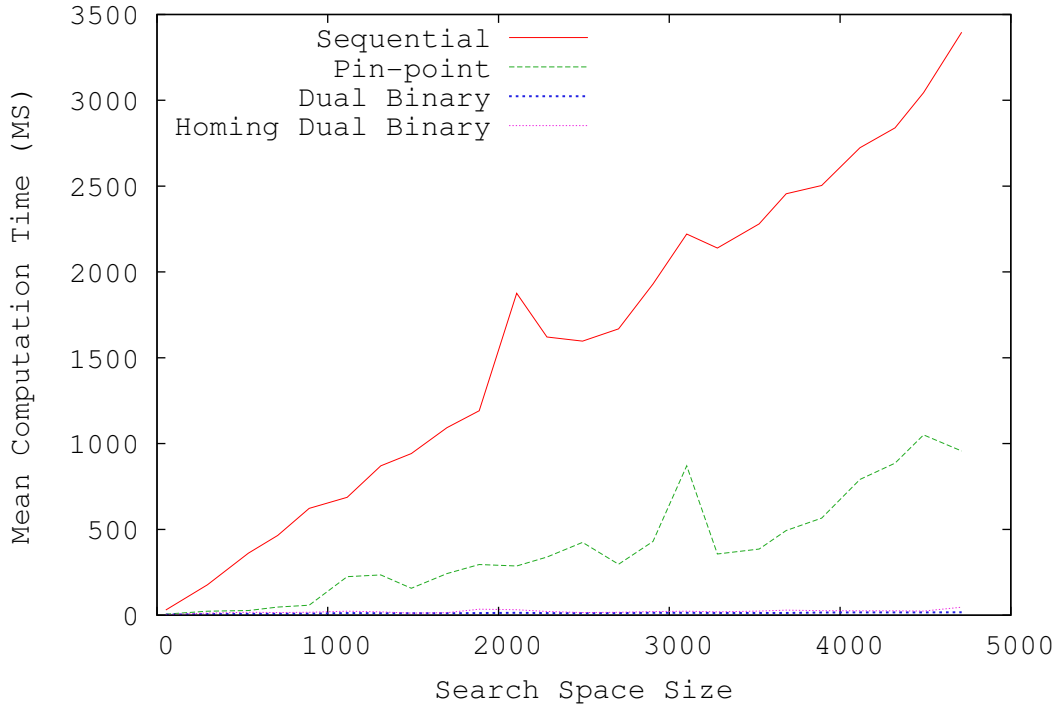
Potential biases effecting computation times by systems beginning or ending background processes during a benchmark session were minimised by interleaving the four algorithms per sample as opposed to recording results one algorithm at a time. For example, if a system update process occurs while the benchmark is only half way through benchmark, succeeding computation times recorded for all algorithms would slightly increase instead of just for one or two remaining algorithms.

A total of eight benchmarks were performed, where each benchmark was on a unique platform (web browser/version and operating system) on a range of hardware devices and screen resolutions. A total of 66,410 samples evenly distributed across the eight benchmarks were recorded for the full scale analysis.

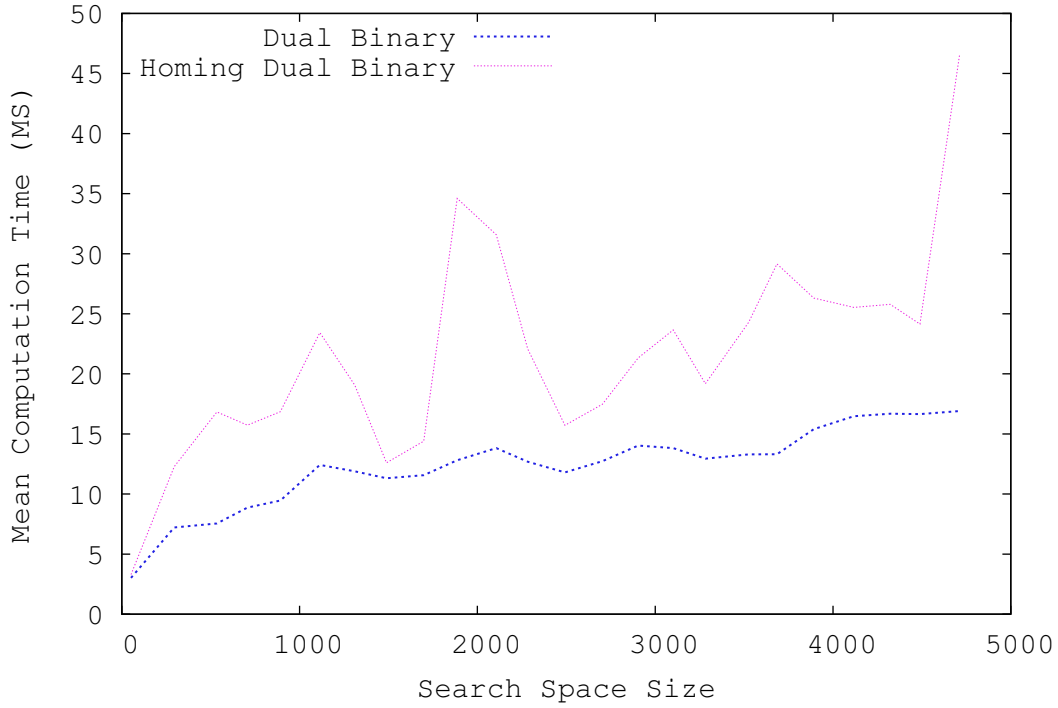
Figure 3.15 displays two graphs that plot the performance results for each algorithm. A single line on the graph represents the average computation times between the eight benchmarks it took for an algorithm to determine a cursor position, for a range of search space sizes. Because benchmarks varied in sample sizes (due to different screen resolutions between benchmarks), the averaged times for forming the lines came from average computation times per benchmark session within sample size ranges of 200. For example, the first peak that occurs in Figure 3.15(a) for the sequential algorithm at a search space size of approximately 2100 was calculated as follows: for each benchmark, the average computation time for the sequential algorithm was calculated between samples that had a search space size within the range 2000-2200. These (eight) averages calculated for each benchmark were then averaged together to give the final value of approximately 1875 milliseconds.

The *maximum delay threshold* for which computation times become unsat-

[9]Multiple machines running Windows with different browser versions were tested.



(a) All algorithms.



(b) Dual binary searches only.

Figure 3.15: Average computation times vs search space size.

isfactory was set at 200 milliseconds. This was determined empirically as the point we found the delay to become noticeable when placing a cursor in a web page. Computation times larger than the *maximum delay threshold* begin to lose the feeling of direct engagement with the editable content.

As expected the sequential search performance has a big O notation of *O(n)*, confirmed by the linear correlation between computation time and search space size. This method is clearly unsatisfactory since it breaches the *maximum delay threshold* at a search space size of just 324 characters (for the average case).

The pin-point algorithm's performance is significantly better than the sequential search. There does not seem to be a strong correlation between computation time and search space size. The raw plots of the benchmark sessions reveal that the pin-point algorithm's performance can dramatically vary for the same search space size (see Appendix C for examples of raw plots). The most extreme case recorded was 5325 milliseconds at a search space size of 4410 on a virtual Windows XP machine running Firefox 3.5. This algorithm is not fit to use since there were too many cases that lasted longer than *maximum delay threshold* milliseconds, including cases with small sized search spaces.

The dual binary search algorithms are significantly faster than the others as shown in Figure 3.15(a). Figure 3.15(b) shows the same graph but with the averaged benchmark results exclusively for the dual binary search algorithms. In the average case neither of the algorithms breach the *maximum delay threshold* in search spaces up to 5000.

Surprisingly the homing dual binary search has a slower performance compared to the other. There is no clear trend with the performance results for the homing search. The graph shows that the plain dual binary search (non homing) has a big O notation of *O(log n)* which is as expected since the performance of binary search algorithms in general are *O(log n)*. The largest computation times for the plain and homing dual binary searches were 336 milliseconds and 969 milliseconds respectively.

Due to the performance advantage, reliability and less code required for implementation, the plain dual binary search algorithm was chosen over the

homing dual binary search for the cursor's spatial placement algorithm.

Generalisation

The search space definition used for discussing the cursor algorithms only includes text nodes for cursor placement next to characters. The actual implementation supports cursor placements next to HTML elements such as to the left/right of <*img*> elements, or to the right of <*br*> elements. The implementation of the dual binary search algorithm is virtually the same as described, however the search space can include HTML elements instead.

The spatial information for elements can be measured directly without having to use measuring nodes. The <*br*> element is the only exception, as some browsers do not provide spatial attributes for this element. In these cases, the spatial information is calculated from a <*span*> element with a text node containing a single character placed directly after the <*br*> element.

3.5.3 Vertical Placement Algorithm

When users press the down or up arrows on the keyboard, the cursor is placed on the next or previous line in the same x coordinate. Lines in HTML documents are not uniformly spaced, thus using the dual binary search algorithm with an offset y coordinate to infer the cursor placement on the line above or below a given cursor position does not suffice. The dual binary search algorithm was therefore adjusted to discover cursor placement above and below a given cursor position.

The vertical placement search space differs from the spatial placement search space as previously described. For the downward placements, the search space is collected by traversing in-order, starting from the given cursor position, and stops until it finds a node which is inferred to be at least two lines away from the start point. For an upward search space, the traversal is reversed. The nodes in the search space are not capped to only include nodes that are displayed in the browser's view-port because the new cursor placement might be on a line that is not in view. If the discovered cursor position is

not in view, the web browser is automatically scrolled to reveal the new cursor position.

For downward searches the lower bound initially becomes the starting point. Otherwise if searching upward, the upper bound initially becomes the starting point. Instead of searching for the placement nearest to a given position, the algorithm is tweaked to search for the placement nearest to the start point coordinate *AND is not on the same line as the start point.* If a position is found to be on the same line as the starting point, then the sample becomes the new upper bound if performing a downward search, or lower bound if performing an upward search. For each measurement, the *current closest* measurement is only updated if the measurement is not on the same line as the starting point.

3.5.4 Offset Placement Algorithm

If the user presses the left or right arrow keys on the keyboard, the cursor should move one placement to the left or right of the current cursor position.

If moving right, an in-order traversal is performed, starting from the current cursor position. The traversal is reversed for moving to the left. The search is finished when the first cursor placement is found.

If the next cursor placement is calculated to be on a new line, then the cursor's *isRightOf* flag may have to be flipped (see section 3.5.1). For example, if the cursor is placed to the right of a character that happens to be the last character on a wrapped line and is to be moved to the right, the next cursor placement would be set to the first character on the line below, and the cursor must be placed to the left of that character.

If during the traversal an empty container element is found then a *modifiable node* place-holder element is created and inserted into the container. In such cases, the new cursor position is set to the inserted place-holder. For example, if a <*p*> element with no content is encountered, a place-holder would be inserted inside the <*p*> and the cursor placement would be set as the place-holder.

3.6 Selection

Selection is a fundamental aspect to any document editor. All web browsers support selection within any HTML document. Unfortunately the native selection facilities cannot be utilised for the Seaweed editor due to the dual binary search algorithm.

3.6.1 The Problem with Native Selection

In all web browsers, when selecting within an editable section, the native selection mechanism is disturbed by the dual binary search algorithm. Once the selection gesture begins, the cursor placement algorithm manipulates the DOM to discover where to place the cursor. Even though the algorithm recovers the DOM state, the brief change in the DOM structure causes the native selection process to lose track of the start point of the selection. Thus, in some cases, the selection gesture is cancelled. In a more common case, the selection range rapidly changes, resulting in unpredictable behaviour. For example, during selection gesture, the selection range's start sometimes resets to the beginning of the whole document or nearest block-level element, and the selection range's end point wildly "spasms", randomly changing position all over the document as the user drags the mouse.

The selection randomly changes whenever the DOM is manipulated, therefore this becomes an issue when performing WYSIWYG actions on the DOM against a selected range. Ideally a work around would be to manually re-select the selection ranges after a selection gesture or WYSIWYG editing action is complete. However, due to security reasons, web browsers do not allow native selection in non-form/control elements to be set via JavaScript.

3.6.2 Seaweed's Selection Model

To support selection, a selection model was implemented in JavaScript. The selection has a start and end point two-tuple containing a node and an index representing cursor positions in the DOM. The selection can be in one of three states:

1. *no selection*: where nothing is selected and no cursor is present.

2. *single point selection*: where the start and end points of the selection range are the same, which is set to the cursor position within an editable section.

3. *ranged selection*: where the start and end points of the selection range are different and do not have to be in an editable section. If the range is fully within a single editable section, the cursor appears at the end-point (which can visually occur before or after the start point). Ranged selection includes both editable and read-only content in a web page so users can copy content anywhere within a web page.

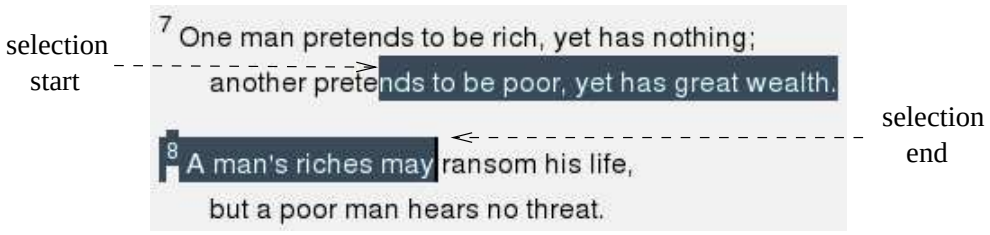


Figure 3.16: Ranged selection example.

When a selection is ranged, highlighting is used to visually display the range. This is achieved by rendering the range using the CSS *color* and *backgroundColor* properties. Text nodes may have to be split into two or three nodes and encapsulated with <*span*> elements to apply CSS highlighting to the selection range within a text node. For example, Figure 3.16 shows a ranged selection between two editable paragraphs that begins and ends within text nodes. The text node where the selection begins is split into two nodes at the middle of the word "pretends". The text node where the selection ends is also split into two nodes, at the end of the word "may". The new DOM structure created for CSS highlighting is not seen by the rest of the Seaweed framework: the selection model restores the structure before any WYSIWYG actions are executed.

To override the native selection, the *mousedown*, *click* and *selectionstart* DOM events had to be consumed. Consequently keyboard strokes are unable to

be observed by Seaweed because the web page could not obtain focus due to the consumption of the selection-related events. To overcome this, whenever the user created a new selection the web browser's focus is set to a hidden <*input*> element resided within the document's DOM. Thus after each selection change the focus remains within the document, and keyboard events can be observed by Seaweed.

3.6.3 Keyboard-controlled Selection

Typical keyboard combinations for selecting content were implemented. For example, CTRL+A selects all of the content in the current editable section which the cursor resides. Holding shift while either using the arrow keys or mouse to move the cursor extends/shrinks the selection. This created a closer feeling to a typical document editor.

3.7 Fragments

When a user selects a range of content, they usually intend to either format the range or delete it. The DOM level 2 specification includes a DOM object called a *Range*. The *Range*'s data-structure is represented in the same way as Seaweed's selection model. The specification outlines methods for manipulating content within a specified range, which includes deleting the content. Unfortunately, not all browsers support the range object, for example, Internet Explorer 7 and below do not support it. Furthermore the DOM manipulations are not reversible (for conducting undoable operations). A JavaScript implementation of *Range* objects, called *fragments*, was developed as part of the Seaweed framework to address these shortcomings.

Figure 3.17 shows a fragment for a range of nodes in a DOM tree containing a numbered list of travel destinations. The fragment begins within a text node at a zero-based character index of 19, and ends within another text node at a one-based character index of 4 (a white-space symbol is included at the end of the range). The DOM nodes within the fragment are displayed with surrounding boxes. In the case of the start and end points, the boxes are

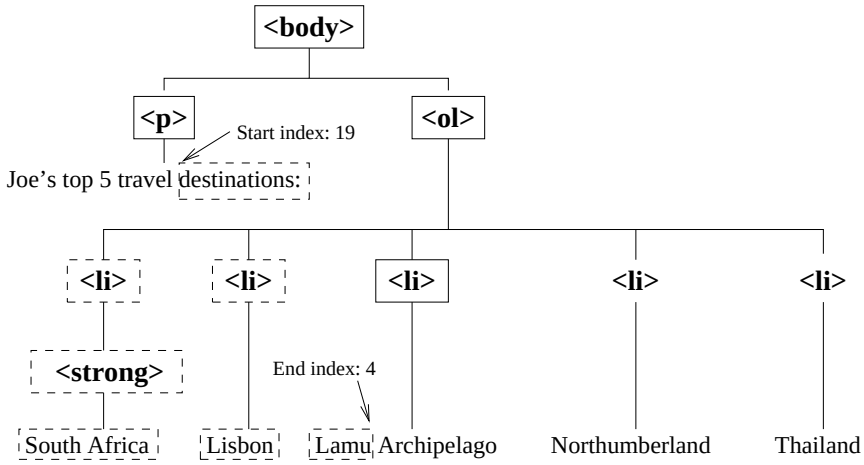


Figure 3.17: An example of the fragment data structure in relation to the DOM.

within the text nodes to signify that the fragment begins within those nodes.

The start point can have an index set to the length of a text node. This includes the text node in the range yet does not physically encapsulate the content within the range. For the end point, an index of zero includes a text node in the range yet does not physically encapsulate the content within the range.

Fragments are used to aid DOM manipulations used for highlighting selections, as well as a range of algorithms that provide WYSIWYG editing. Fragments take care of splitting nodes for start and end points which occur within a text node, and can be easily traversed for analysing the DOM content within a fragment.

3.7.1 Disconnection Algorithm

A fundamental operation in any document editing software is the removal of a selected range. This section outlines a simple algorithm used as the basis of removing a selected range.

Fragments mark DOM nodes as being either shared or not shared. A node is considered shared if it contains children in the DOM that are not within the fragment. For example, the fragment displayed in Figure 3.17 contains four shared nodes and seven non-shared nodes. The shared nodes are illustrated by

surrounding them with boxes using hard lines, whereas the non-shared nodes are illustrated by surrounding them with boxes using hashed lines. The <*p*> element is considered shared because it contains one text node that is not within the fragment, which is the text node at the start point that has been split into two containing the text "Joe's top 5 travel ". The left-most list item element displayed in the figure is considered non-shared because all of its descendants are also non-shared. Leaf nodes (such as text nodes or <*img*> elements) can be considered shared only if they are on the start or end points, and the index excludes the node from being visually within the range. For example, in the case of a selection range that ends to the left of the the first occurring character inside a paragraph (index of zero), the range includes the text node, yet the actual text content is not encapsulated within the range itself.

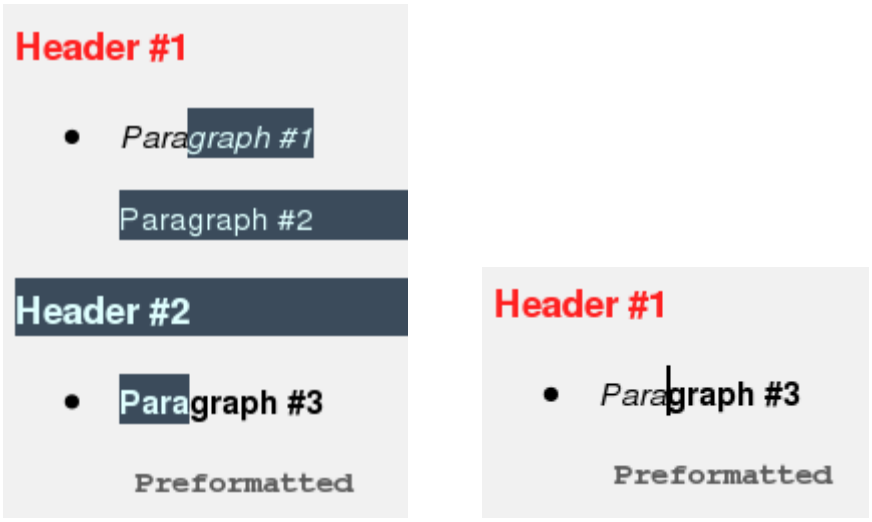
To remove DOM nodes that are within the fragment, all non-shared nodes can be safely removed. This algorithm is simply implemented by traversing the fragment, and removing each non-shared node from the document.

3.7.2 Collapse Algorithm

The disconnection algorithm does not provide the typical behaviour of removing a selected range in a WYSIWYG editor. For example, disconnecting the fragment displayed in Figure 3.17 will leave the text " Archipelago" untouched, still residing in its parent (the shared <*li*> element). However, one would expect the <*li*> to also be removed, and the text to merge with the text at the start point of the fragment.

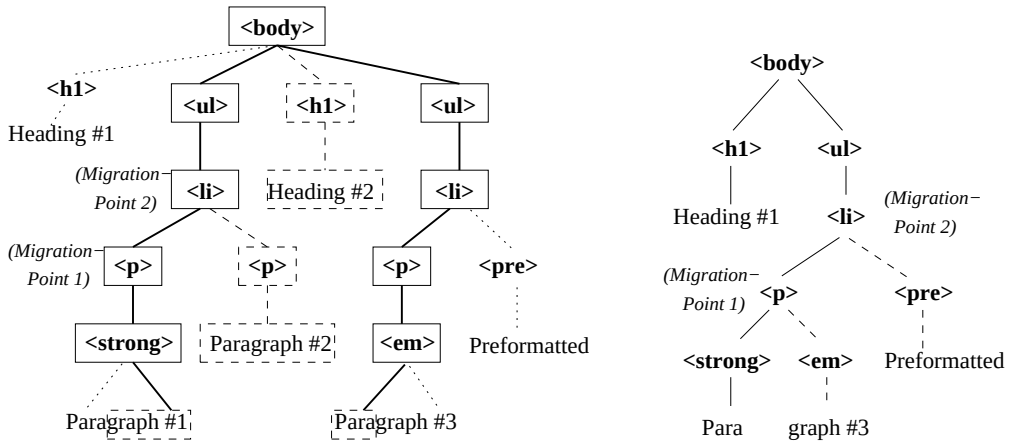
Table 3.2 details the operations used by the an algorithm to achieve WYSIWYG style removal of complex ranges. This algorithm is called the *collapse algorithm*. The disconnection algorithm only deletes nodes thus clearly does not suffice, however it is used as the starting point of the collapse algorithm. Once a fragment is disconnected, the nodes on the outer bounds of the fragment (exclusive of the fragment's range) are collapsed together using a combination of operations listed in Table 3.2.

Figures 3.18 and 3.19 presents an example of the collapse algorithm in ac-



(a) Fragment to collapse. (b) After collapse (cursor shown).

Figure 3.18: Example of collapsing range: before and after from the user's perspective.



(a) Fragment to collapse. (b) After collapse.

Figure 3.19: Example of collapsing range: before and after view of the DOM tree.

Operation	Description
deleteNode	To remove non-shared nodes. To remove generic block-level list item elements along the end bounds. To migrating nodes into migration-points in the start bounds of a fragment.
insertNode	To insert new place-holder elements in empty container elements. To migrate nodes from the outer-end bounds of fragments to migration-points.
createNode	To create new place-holder elements for empty container elements.
cloneNode	To duplicate inline elements in the end bound which need to stay, yet be copied across with migration to retain CSS formatting.

Table 3.2: Operations required for collapsing.

tion. Figure 3.18(a) shows a screen-shot of a selected range of editable content in a web page using Seaweed. When a user deletes this selection, the result is shown in Figure 3.18(b). Figure 3.19(a) is the underlying DOM tree for Figure 3.18(a). It shows the fragment for the selection using the same convention as previously described for distinguishing between shared/non-shared nodes. The dotted lines connecting nodes simply clarify that they are not part of the fragment data-structure (since they are not within the selection range). The hashed lines connecting nodes represent connections to non-shared nodes. The hard-lines connecting nodes represent the outer-bounds of the fragment.

There are many facets to the collapse algorithm. The first step disconnects the non-shared nodes. The end outer-bound of the fragment is visited from the deepest node up to the fragment's root (the <*body*> in this case). The <*em*> node (and its contents) is then migrated into the first *migration point* as labelled in Figures 3.19(a) and 3.19(b). Migration points are container-like nodes such as generic block-level elements. The <*p*> element in the end outer-bounds of the fragment is then removed since it is being merged with the first migration point. The adjacent <*pre*> element is migrated across to the second migration point. The second migration point was chosen because <*pre*> elements cannot validly be inside <*p*> elements. A DTD utility package

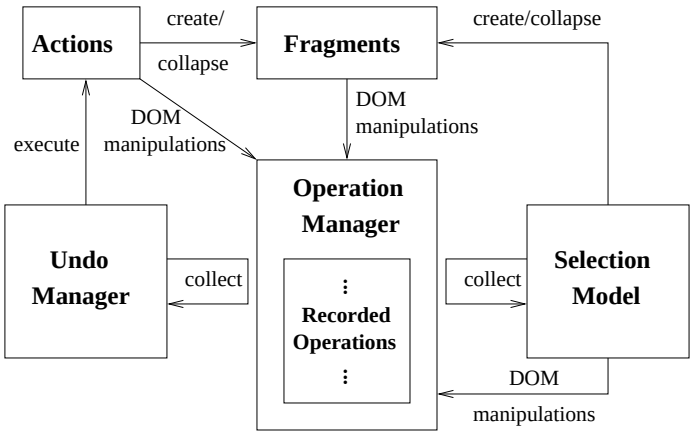


Figure 3.20: High level view of the undo/redo system.

was written to determine validity for carrying out such decisions. The <*li*> in the end outer-bounds was removed because it was being merged into the start outer-bounds of the fragment. The <*ul*> in the end outer-bounds was removed because it no longer contained any content (that is rendered).

3.8 Undo and Redo Management

It is important to give the ability for users to move back in their edit transaction history. The ability of reversing an action is considered a virtue for direct manipulation systems (covered in Section 2.2). The undo interaction model is built upon the principle of reachability: a system is reachable if from any state the system is in, a user can get to any other state [16]. Returning to a state which the user has just been in (undo) yields better reachability. Moving forward in the undo transaction history (redo) also helps improve the reachability of a system. All of Seaweed's edit actions are built upon an undo/redo model, which is described in this section.

Figure 3.20 displays a high-level view of Seaweed's undo/redo system. The *undo manager* is used to execute, undo and redo editing actions as well as manage the transaction history. Actions are broken down into a sequence of operations, which are undoable/redoable DOM manipulations governed by an *operation manager*. Fragments use the *operation manager* for all their DOM manipulations, which the actions and the selection model both make use of.

The undo manager and selection model collects the list of recorded operations from the *operation manager* for undoing and/or redoing.

3.8.1 Actions

A total of 16 actions were developed to support the typical editing commands of a HTML document editor (see Appendix B for descriptions of all actions developed for Seaweed). When a user constructs a task for editing content they must carry out one or more actions to complete that task. For example, for the task of replacing the word "good" with the word "excellent" in a paragraph, the user might select and delete the word "good", thus using a *RemoveText* action, then type the word "excellent" one character at a time, thus using nine *InsertText* actions.

3.8.2 Transaction Data Model

Every action manipulates the DOM tree. For example, using the *InsertText* action for inserting the letter "t" in a paragraph changes the text content in a text node residing in the DOM tree. To undo the action, each DOM manipulation used to carry out the edit action must be reversed. The DOM tree must be in the exact same state it was just after the action was first executed for the undo to work. To redo the action, the same DOM manipulations used to execute the operation can be simply re-executed in the same order. The DOM tree must be in the exact same state it was just before the action was first executed for the redo to work.

Figure 3.21 displays the data model used to maintain DOM tree states between undo and redo transactions. The transaction history is represented as a doubly linked list. The tail of the list represents the first executed action (in this case). The head of the list represents the last executed action. The figure displays a transaction history of four actions. One of the nodes in the linked list is set as the *current action* — representing the last executed action that has not been undone. If an action is undone, then the current action is set to the previous node in the transaction history.

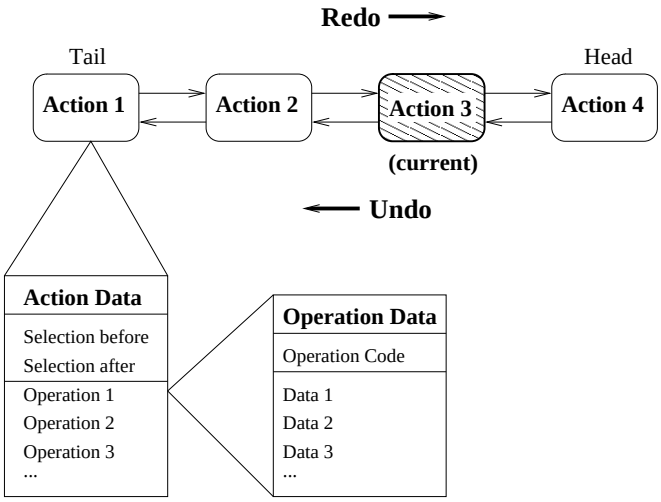


Figure 3.21: The undo and redo transaction data model.

To avoid consuming too much memory, the *undo manager* limits the transaction history size. By default Seaweed limits this to 100 actions. When the transaction history is full, the tail is dropped. To avoid memory leaks, whenever one or more actions are dropped all the pointers used in the doubly linked list are nullified, since some browsers do not garbage collect cyclic references.

Each action can be broken down into a chronological list of operations on the DOM tree. Figure 3.21 displays the data-structure of an action, consisting of selection data along with the list of operations carried out to execute the action. The selection data is used to maintain the selection state after the user performs an undo or redo on the action. Each operation stored in the action data-structure consists of an operation code and operation-specific data. The operation code is a number used to identify the type of primitive operation used. For example, the operation used to insert a node is represented by the number one. The operation's data may consist of DOM nodes, strings and/or numbers, which are required to execute, undo and redo the operation. For example, the operation used to insert a node stores a reference to the new node being inserted, a reference to a parent node which the new node is being inserted into, and the index number of the child-position the new node is being inserted into the parent.

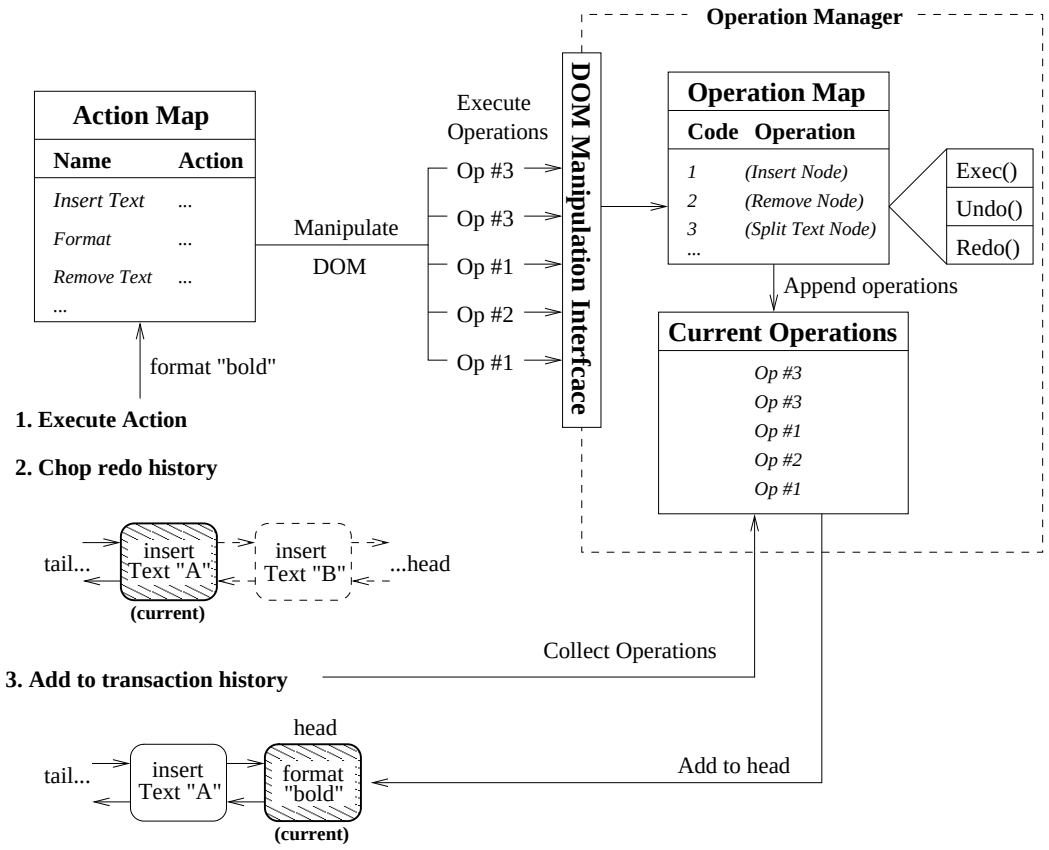


Figure 3.22: Execution example via the undo manager.

3.8.3 Operation Manager

A set of seven primitive operations were written to accommodate for all editing actions used in Seaweed. The operations contains execution, undo and redo logic which is performed against the DOM tree with supplied operation-specific data (as previously described). They are stored in a map, which translates operation codes to actual operation objects. Using operation codes as opposed to self-containing objects helps reduce the memory footprint used by the transaction history.

The selection model collects operations used by fragments and directly by the selection model for creating CSS highlighting. Before an action is executed the selection model will undo any highlighting operations via the *operation manager* to keep the DOM tree in a pristine state.

3.8.4 Undo Manager

Figure 3.22 presents an example of executing an action via the *undo manager*. In the first step, the *undo manager* locates the requested action object via a look-up map and executes it with the supplied information. The selection state as well as other parameters is passed to the action. For example, the parameter "bold" is passed to the *Format* action shown in Figure 3.22, to bold the currently selected text. The action will manipulate the DOM tree to carry out the action, where each manipulation is performed via the *operation manager*. In this example, the format action uses five operations to bold selected text: two splitting operations are used to isolate the selected text, a <*strong*> element is inserted next the isolated text node, the isolated text node is then removed from the DOM, and finally inserted in the <*strong*> element. The *operation manager* records each operation in a list. The *undo manager* then drops the redo history (since the DOM tree state has changed and all undone operations cannot be redone). Finally the *undo manager* collects the operations from the *operation manager* storing them in an action data-structure (see Figure 3.21), and appending the data-structure to the head of the transaction history.

The *undo manager* performs an undo by invoking the undo logic for each operation listed in the current action is reverse chronological order. Redoing an action is achieved by invoking the redo logic for each of the action's listed operations in chronological order.

Table 3.1 specifies a property called *actionFilter*: a regular expression of the names of the actions which can or cannot be executed in the editable section the property is assigned to. The *undo manager* only executes an operation if the name of the action to be executed satisfies the regular expression.

The undo/redo model for Seaweed takes care of the undo and redo logic for every action, thus making it simple to develop them and reduces the amount of code needed for implementation.

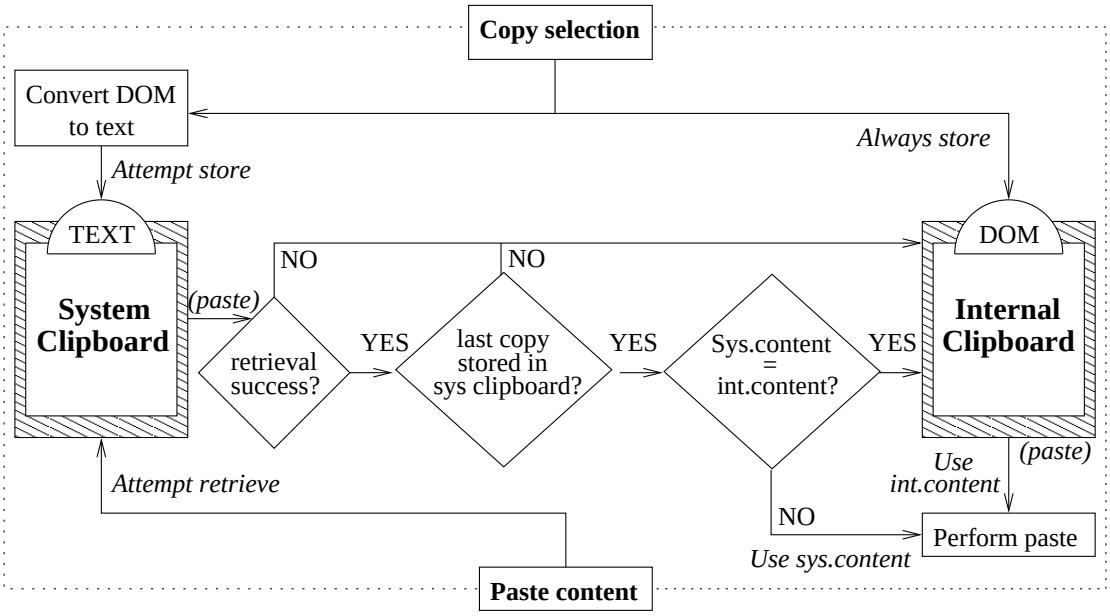


Figure 3.23: The clipboard interface.

3.9 The Clipboard

Copy, cut and paste are essential features found in any editor. Unfortunately the W3C DOM specifications do not specify a clipboard interface. This section reports on the research and development of a clipboard interface developed for Seaweed.

3.9.1 The Internal and System Clipboard

Ideally when a user copies or cuts content, the content is transferred to the system clipboard. Likewise when the user pastes content, the content to be pasted would ideally be sourced from the system clipboard. However because the system clipboard is a system resource which can contain users private information, web browsers restrict JavaScript from accessing the system clipboard to only allow certain access in specific contexts. Because of this restriction, cut/copy/paste commands may not be able to access the system clipboard. Furthermore, due to lack of DOM specifications web browsers have their own way of accessing the system clipboard.

To avoid losing the user's copied content, the content is also stored in an internal clipboard. Figure 3.23 displays a high level view of the clipboard

interface. Whenever a user copies selected DOM, the DOM is converted into a textual representation and an attempt to transfer the text to the system clipboard is made. The DOM must be converted to text because the browser-specific methods for storing content in the system clipboard only accepts string data. A local copy of the selected DOM nodes is kept in an internal clipboard. If an attempt to store the content to the system clipboard fails the user cannot paste the content into another program or another web page. However they can still recall the content within the Seaweed session via the internal clipboard, for pasting, as depicted by the logical flow diagram in Figure 3.23.

The internal DOM nodes retain that formatting which is lost during the conversion of DOM to text when storing the content in the system clipboard. Therefore when a paste is performed and the internal clipboard content is the same as the system clipboard content, the internal clipboard content takes precedence because it contains a richer representation.

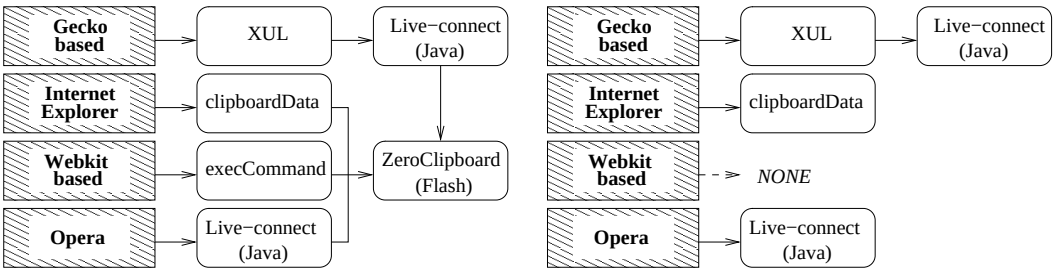
3.9.2 Access via the mouse

Web browsers only allow the system clipboard to be accessed either via native clipboard events (which cannot be fabricated via JavaScript), or with explicit permission by the users. In the worst case browsers do not provide any way to access the clipboard. Common use cases for on demand access to the system clipboard is a user clicking on custom cut/copy/paste buttons. The JavaScript used to perform the clipboard commands is executed via a mouse click event. In these cases, system clipboard access is denied for web browsers that only allow system clipboard access via native clipboard events.

Figure 3.24 displays a flow diagram of the browser-specific methods used to access the system clipboard. Upon failure of an attempt to use a method, another method is used. Each method is briefly described in turn.

XUL

Gecko-based browsers are build on a framework called XUL (XML User-interface Language). An inbuilt XUL plugin called *clipboard helper* is available for use via JavaScript. The plugin provides full read and write



(a) Copy methods. (b) Paste methods.

Figure 3.24: On demand clipboard access methods.

access to the system clipboard, given that the calling JavaScript has permission. A script can get permission if it is digitally signed or explicitly allowed by the user via a prompt. Due to financial costs, digitally signing was not an option for this project.

Live Connect

Live Connect is an older web browser technology developed for Netscape which allows JavaScript to directly communicate with Java code and Java applets embedded within a web page. Java applets that are digitally signed are allowed full access to various system resources, including the clipboard. Only Opera and Gecko-based browsers support this technology.

clipboardData

Internet Explorer provides a global object called *clipboardData* that gives full read and write access to the system clipboard. Browser versions 7 and above prompt the user to explicitly allow JavaScript access, thus can fail if the user denies the request.

execCommand

In a pursuit to discover ways of accessing the clipboard, a security hole in Webkit was discovered which allows JavaScript to copy to the system clipboard without prompts or digital signing and in any context.[10] This is achieved via a hidden <*iframe*> element with *designMode* turned on

[10]The security hole has been reported to the Webkit development team by the researcher.

(see section 3.1) stored in the web page. The *designMode* turned on exposes a method called *execCommand()*. To copy content, the body of the <*iframe*> is set to contain nothing but the DOM content to be copied. The <*iframe*> is selected/focused, then a "copy" command is executed via the *execCommand()* on the <*iframe*> and the content is copied to the system clipboard.

ZeroClipboard

A commonly used method called ZeroClipboard exploits Adobe's Flash plugins to copy text to the system clipboard via mouse clicks.[11] Invisible Flash elements are hovered over cut and copy buttons in the web page. When a user clicks these invisible elements, action script is executed in a mouse event, where Flash allows copy-access during mouse events. All major browsers support Flash plugins for embedding flash elements on a web page. This method fails if the user does not have Flash installed for their browser.

3.9.3 Access via the keyboard

Two methods were used to get access to the system clipboard when users cut/copy/paste using globally known keyboard combinations.

Hijacking clipboard commands

When the user presses a clipboard key combination for cutting or copying, a <*textarea*> element hidden from view is used to hijack the event.

If a cut or copy key combination occurs, the selected content (in Seaweed's selection model) is copied to a hidden <*textarea*>. The document's selection and focus is then set to the <*textarea*>. The keyboard event is not consumed, but instead bubbles up to execute the web browsers native event handler for the key combination. Due to the document's focus and selection being changed to the hidden <*textarea*>, the native event handler executes a native copy command, which in turn, successfully transfers the selected text to the system

[11] See http://code.google.com/p/zeroclipboard.

clipboard. The native copy is executed on either a *keydown* or *keypress* event, which depends on the browser. A <*textarea*> is used to support multi-lined content.

Pasting is achieved in a similar way as copying to the clipboard. Instead of copying the text from Seaweed's selection model into the hidden <*textarea*>, the content in the <*textarea*> is cleared. A callback function is scheduled to execute after the native paste event handler using a timer. The callback function then retrieves the pasted content from the <*textarea*>.

Using clipboard events

Webkit-based browsers on Windows platforms allow access to a *clipboardData* object during clipboard copy and paste events (but not cut events). The *clipboardData* object is the same implementation as Internet Explorer's, except that it does not prompt the user for allowing access to the system clipboard. The clipboard events are raised by the user pressing clipboard key combinations and via the browser's main menu/context menus. Browsers that support clipboard events use this method in favour of the high-jacking method, so that native menus can be utilised as well.

3.10 Managing White-space

HTML layout engines following a white-space processing model outlined in W3C's CSS specification.[12] Given that "normal" text wrapping is used, a sequence of white-space symbols is collapsed into one single symbol. Furthermore, all white-space is removed at the beginning and ends of HTML content within a block level element.

White-space symbols encompasses all types of values such as new-lines, tabs and carriage-returns. Apart from separating words, they are typically used for structuring HTML syntax for readability purposes.

The white-space processing model poses a problem when the user edits

[12]See http://www.w3.org/TR/CSS21/text.html#white-space-model and http://www.w3.org/TR/html401/struct/text.html#h-9.3.5 for white-space specifications.

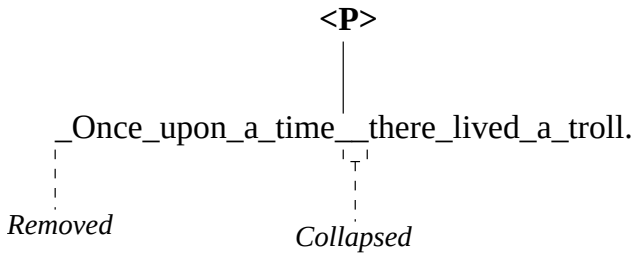


Figure 3.25: White-space processing model example.

the DOM. Consider the scenario in Figure 3.25: when the user deletes the letter "a", the two surrounding white-space symbols are left side by side. Note that the figure represents white-space with underlines for clarity. Visually, the HTML engine would collapse the symbol into one, thus giving the effect that a white-space was deleted as well, where in fact it is not. In another example, if the cursor is placed to the right of the white-space directly after the word "time" in Figure 3.25, the cursor is visually positioned before the letter "t" in the following word "there" since the second white-space is not rendered due to the HTML engine collapsing it. If the user then enters a letter such as "I", the second white-space would no longer be collapsed since the inserted letter separates the two white-spaces. In these cases, from the user's perspective an extra white-space seems to appear from nowhere.

To prevent unexpected white-spaces from appearing or disappearing, two white-space management protocols were implemented.

3.10.1 Pre-processing DOM Protocol

A JavaScript implementation of the white-space processing model was developed. The implementation takes a given fragment and physically removes white-space symbols from the DOM which would be ignored by the HTML engine due to collapsing.

When the Seaweed framework is initialised, all editable sections are pre-processed using the custom white-space processing model. Any new editable sections added at runtime are also pre-processed. One other case is the *InsertHTML* action, where the DOM tree created from the HTML is pre-processed before inserting into the document. Thus in all situations where

HTML is first loaded into the document, Seaweed ensures that the initial state of the editable HTML contains white-space that is only rendered.

3.10.2 DOM Manipulation Protocol

In cases where a sequence of white-space needs to be physically rendered, the NBSP (non-breaking space) character is used. The NBSP character is not considered as white-space by the HTML layout engines so they are always rendered.

Whenever the DOM is manipulated in a way which could lead to forming white-space sequences, the edited DOM range is post-processed to ensure all white-spaces are rendered. For all white-space sequences found in an edited range, the white-space symbols are alternated between plain white-space symbols and NBSP symbols. The alternation breaks all white-space sequences. Alternation is used instead of simply replacing all white-space with NBSP character so that the HTML engine can wrap the text.

3.11 Reducing the Download Bloat

Build scripts were developed to generate two versions of the Seaweed framework: one for development, and the other for public release. The development version pulls in the raw source of all the framework's JavaScripts, which is necessary for debugging the framework using tools such as Mozilla's Firebug. The release version compresses the framework into a single JavaScript. Compression was necessary because in its raw form there are 47 scripts that add up to a total of 619.4 kilobytes of code. This translates to long download times when using Seaweed.

3.11.1 Bootstrapped Release

The development release creates a bootstrap script for pulling in the full Seaweed source. The bootstrap infers the URL of the framework's directory by using regular expressions against the bootstrap script elements source attribute

contained within the web page. From this URL, all scripts are pulled in one by one. The order of the scripts are determined by the build scripts, which extracts dependency trees from encoded commands in the header of the JavaScript files.

3.11.2 Compressed Release

The release version was built by joining all the source files into one. The order of where scripts were included was controlled by declaring dependencies encoded in JavaScript comments. The YUI compressor tool[13] was used to compress the merged JavaScript file. The JavaScript for each file was written using a convention to yield high compression, which ended up at 105 kilobytes for the entire framework.

To aid development, a debug name-space was developed. The name-space provided cross-browser routines for printing debug messages and making assertions throughout the code. The release version removed all code which used the debug name-space. This was achieved by extending the YUI compressor tool to have the ability to remove name-spaces from JavaScript code.

[13]See http://developer.yahoo.com/yui/compressor for project web site.

Chapter 4

Seamless Editing for WordPress

WordPress, a popular open-source CMS for blogs,[1] was enhanced to support seamless editing by developing a plugin, which is then evaluated through two user studies in the following chapter. The development process undertaken serves as a road-map for integrating the Seaweed framework into any CMS. The chapter begins by taking a glance at WordPress' features, and is followed by a summary of the key features supported by the Seaweed plugin. Section 4.2 presents an architecture overview of the Seaweed plugin. The features provided by the GUI, along with the implementation detail are discussed in Section 4.3. The method used for creating editable sections in blog pages is covered in Section 4.4. A detailed look into seamless editing features, and how they are implemented, is covered in Section 4.5. Overall details surrounding the client to server communications are lightly covered throughout the chapter — in Section 4.6 we delve deeper into the problems that had to be addressed for supporting seamless editing. Lastly, Section 4.7 and Section 4.8 cover the creation and deletion of content respectively.

4.1 Feature Overview

Before discussing the technical aspects of the Seaweed plugin for WordPress, the features of the WordPress CMS are briefly described, followed by a showcase of the key features implemented for the Seaweed plugin.

[1] See http://wordpress.org for the project web site.

4.1.1 WordPress at a Glance

WordPress provides two views of a blog: the front-end view that displays published blog content such as posts and images, and the back-end view that provides administration facilities for managing blog content. Figure 4.1(a) shows a screen-shot of the back-end view, where as Figure 4.1(b) shows the front-end view.

Types of Content

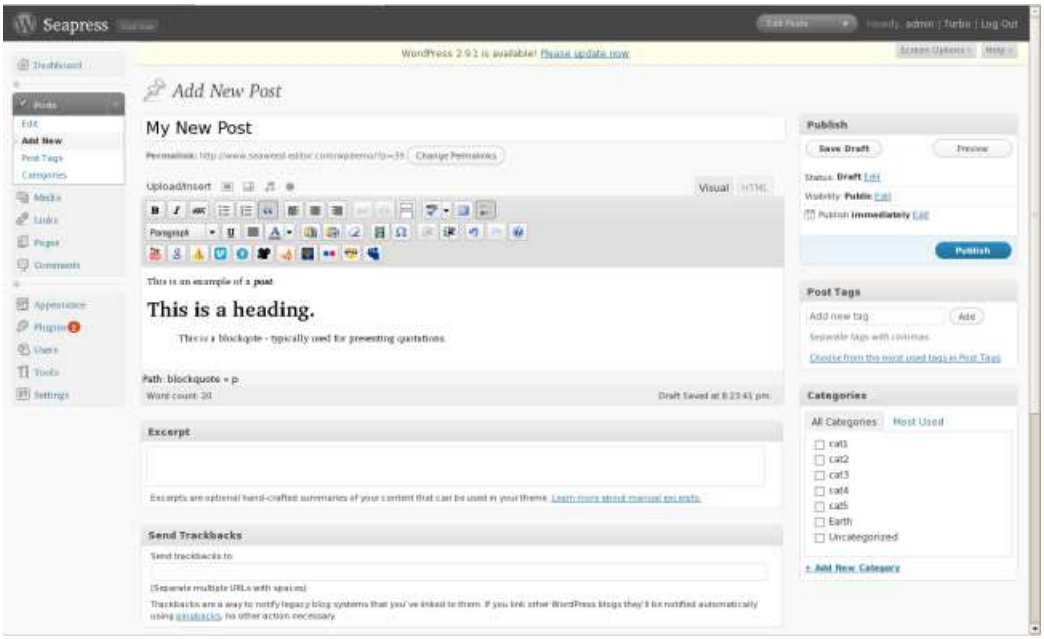
The main forms of content in WordPress are posts and pages. A post is a typical blog entry or article consisting of a title, body-content and a range of metadata including tags and categories. WordPress can display multiple posts on a single web page, usually listed in reverse chronological order. Pages are used for static content, such as general information about the blog or contact details of the author(s). Pages consist of a title and body-content, and are usually accessed via the blog's navigational bar.

All blog visitors and authors can comment on posts or pages. Comments consist of an author name and the HTML content of the actual content, which can contain a limited set of HTML formatting tags.

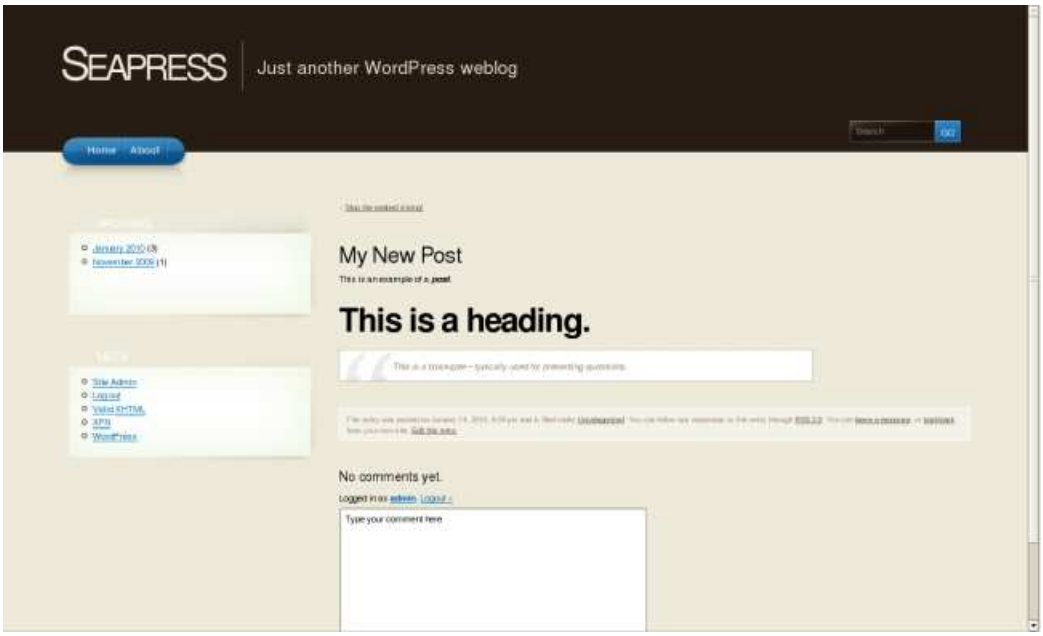
User Accounts

A WordPress blog can support multiple user accounts. There is one administrator account that has full permission to perform any operation in WordPress. Other account types have certain restrictions that prevent users from performing various operations. For example, a user with authorship capabilities may be able to add new posts and edit their own, but is unable to edit other users' posts. Visitors do not have accounts and never see the back-end view.

The back-end view is accessed by visiting a separate web page that presents a login form. Some WordPress configurations are setup to embed the URL to this page as a link within the front-end view for easy access. Users must first login to enter the back-end view using a username and password. They can ask the WordPress login page to remember their login credentials, so that they



(a) Creating a new post in the back-end view.



(b) Previewing a post's draft.

Figure 4.1: A screen-shot of creating posts with WordPress.

will be automatically logged in the next time they visit their blog.

Users can be logged in via the front-end if automatic-login is enabled. When users are logged in while browsing the front-end, exclusive content is supplied, such as edit links. Edit links are placed next to posts and pages which link directly to an editor in the back-end view, acting as a shortcut to edit content without having to find it first in the back-end view's post/page manager.

Themes

WordPress blogs can be themed. There are thousands of themes freely available for users to download. Themes control the content and HTML structure of a blog.

Example of Creating a Post

Figure 4.1 displays an example of creating a new post titled "My New Post" in a WordPress blog named "Seapress." The user begins by logging into the back-end view using their WordPress account. They then click on a button labelled "posts" to bring up the post-manager. Next they click on the "new post" button, which brings them to the post-editor as shown in Figure 4.1(a). Initially the new post is in a draft state. They begin filling in the fields to create their new post. The editor shown in the figure is in *visual-mode*: which provides a WYSIWYG editor for editing the post content. *HTML-mode* is also supported for editing raw HTML. To preview the content, the user presses the preview button — located on the top-right of Figure 4.1(a) — and a new page is opened showing the front-end view of their current draft as shown in Figure 4.1(b). When the user is satisfied with the draft, they press the publish button in the post-editor, which makes the post publicly visible in the front-end view.

4.1.2 Key Features of the Seaweed Plugin

The Seaweed plugin brings WordPress' standard editing facilities directly into the front-end view so users will never have to deal with the back-end view

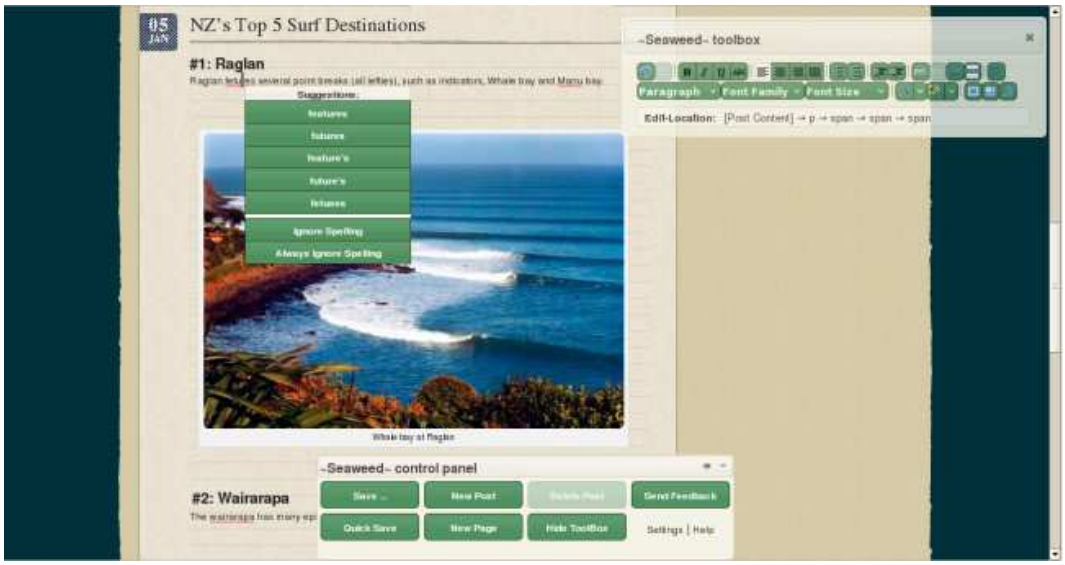


Figure 4.2: A screen-shot of the Seaweed WordPress plugin.

when managing content.

When a user is logged into their blog, all post, page and comment content becomes editable and a GUI for supporting content-management and formatting operations is displayed. Figure 4.2 shows a screen-shot of the user editing a post using the Seaweed plugin. A control panel at the bottom of the web page appears, along with a toolbox dialog providing WYSIWYG editing controls displayed on the top-right. New posts and pages can be created directly from the control panel. Existing posts and pages can be deleted. Changes to content can be saved directly via the front-end without having to load new web pages.

The Seaweed plugin was developed as a prototype for research purposes only. However all WYSIWYG editing facilities the visual editor for WordPress were implemented in Seaweed. Only features that were assumed as being valuable were implemented. For example, moderating or deleting comments via Seaweed was not implemented as they were considered as uncommon tasks (and therefore not valuable).

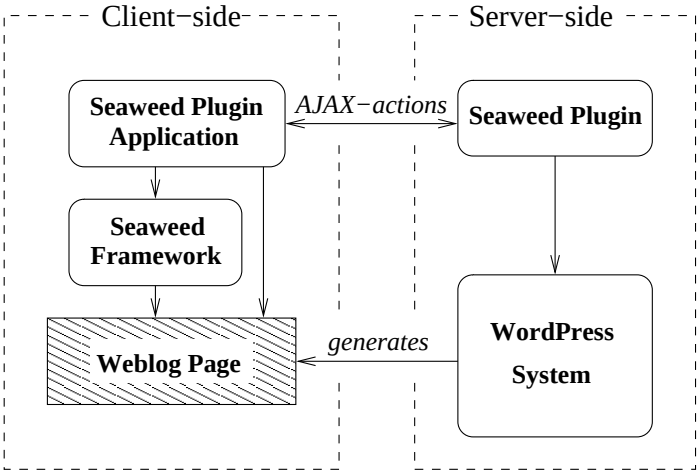


Figure 4.3: Interactions of sub-systems involved in providing a seamlessly editable environment for WordPress.

4.2 Architecture Overview

The Seaweed plugin is organised into two separate sub-systems: one that operates at the server-side and another that operates at the client-side. Figure 4.3 displays a high level diagram of the sub-systems involved in a seamlessly editable web page generated by WordPress and the Seaweed plugin. The arrows represent general interactions between the components.

The Seaweed plugin on the server-side enhances blog pages so that certain regions become editable. It also embeds several compressed JavaScripts so Seaweed plugin can function on the client-side. The client-side sub-system (labelled as "Seaweed Plugin Application" in Figure 4.3) pre-processes the web page and prepares/configures the Seaweed framework. It also dynamically creates and displays a DOM-based GUI that provides controls for various document editing and content management commands.

During a session, the user may save edited content, where AJAX messages are passed between the Seaweed plugin's client-side and server-side sub systems. When the server-side receives AJAX messages for saving, the plugin updates blog content via the WordPress system.

4.3 The GUI

The GUI used for the Seaweed plugin is built upon the JQueryUI framework,[2] which provides a set of widgets that can be themed. Themes can be selected in a settings page dedicated to the Seaweed plugin via the back-end view, as some WordPress themes may look to similar to the widgets making it difficult for users to distinguish between the plugin's GUI and the content within the web page. There are two main dialogs which provide the essential functionality for Seaweed discussed in this section: the control panel and the toolbox.

The GUI is only shown if the user has permissions to edit content on the requested web page. More precisely, if a user requests a page containing editable content, and the state of the session is authenticated, and the authenticated user has authorship capabilities, then the server-side embeds scripts providing all of the plugin's editing facilities and GUI resources in the requested web page.

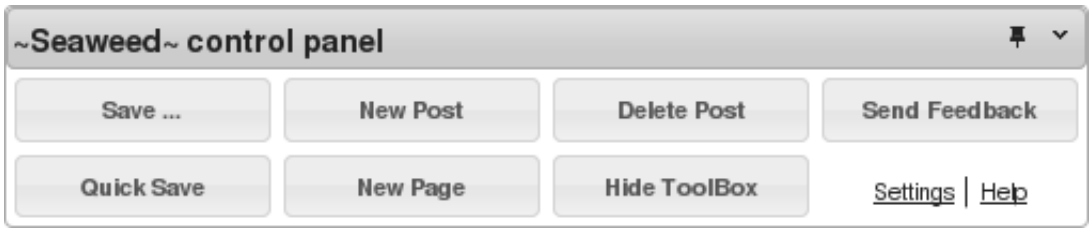
4.3.1 The Control Panel

The control panel provides buttons for managing posts and pages. Figure 4.4 shows a screen-shot of the control panel. When a user is creating a draft post/page (by clicking either the "new page" or "new post" button), different options are shown. The feedback button presents a dialog for users to submit general feedback about the plugin, which is used as part of the evaluation phase in Chapter 5.

The control panel can be minimised and maximised. Initially, by default the control panel starts in a maximised state where the control panel is fully in view. The user may minimise the panel by clicking the arrow button on the top-right corner, as shown in Figure 4.4. Minimising it docks the panel at the bottom-centre of the screen and hides all the buttons below the dialog title.

The control panel can be docked and undocked. Initially the control-panel is docked to the bottom-centre of the screen. However if the panel gets in the way of the content (even if minimised) then the user can undock it can drag it

[2]See http://jqueryui.com for information regarding JQueryUI.



(a) Standard.

(b) Draft.

Figure 4.4: Screen-shots of the control panel.

to anywhere in the web page. Docking is controlled by the button to the left of the minimise/maximise button as shown in Figure 4.4.

Saving Content

The control panel provides a way to save changes to editable content. There are two saving methods: standard-save and quick-save. The former presents a dialog containing a list of editable sections that have been changed, where the user can choose which sections to save, as shown in Figure 4.5. The latter saves all content without confirmation.

The saving operation is asynchronous, and typically lasts one to four seconds (depending on the speed of the Internet connection). As the save operation progresses, the control panel displays a message, as shown in Figure 4.6(a). Once the save operation is complete, the save result message is displayed and fades outs by animating the *opacity* CSS property using jQuery.[3] Figure 4.6(b) is an example of a success message, on the brink of fading out. During this period the user can continue to edit the content.

The save buttons are only enabled if changes have actually been made.

[3]jQuery is a cross-browser API, which jQueryUI is build upon.

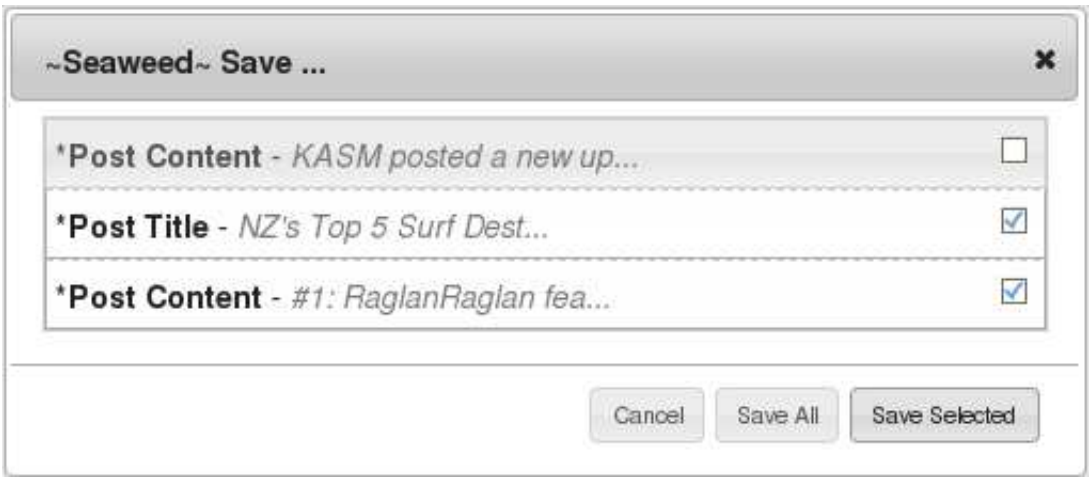
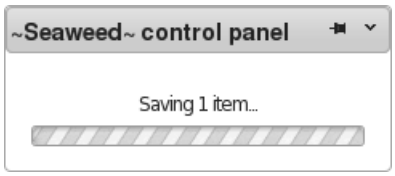


Figure 4.5: The save dialog.



(a) During save.

(b) After save.

Figure 4.6: Displaying the save progress.

Figure 4.7 shows a flow diagram of how the save buttons are synchronised with the actual change-states of the editable sections. The control panel observes the transaction history in the Seaweed framework's undo manager using a model-view-controller design pattern. Whenever an action is executed/undone/redone, the *undo manager* notifies the control panel. The control panel then checks whether there are any changes in any of the editable sections currently on the web page by querying the *change manager* in the Seaweed framework. The save buttons are then enabled if there are changes made. Otherwise they are disabled.

If the user leaves a web page with unsaved changes, a prompt is displayed, warning them that the changes will be lost if not saved. The prompt gives the

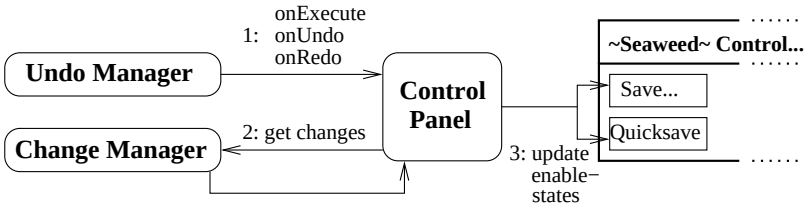


Figure 4.7: Button enable-state management for the control panel.

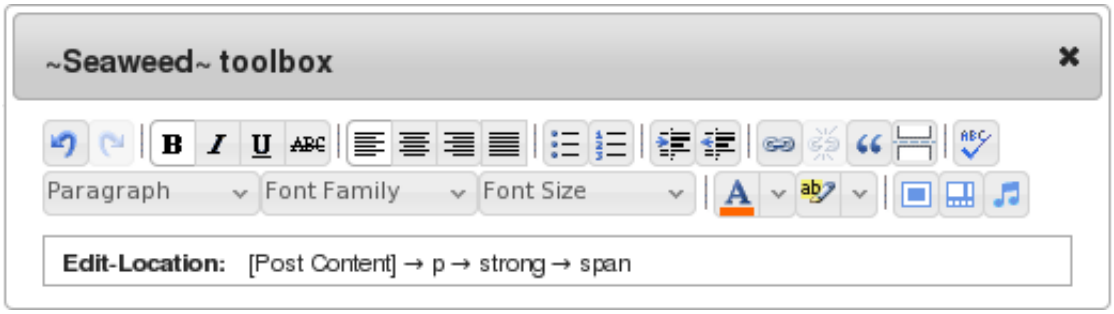


Figure 4.8: The toolbox dialog.

user the option to stay and save the changes before navigating away or closing the web page.

Creating and Deleting Posts and Pages

Users can create new pages or posts from the control panel. Section 4.7 discusses the details regarding the creation of new content. Deletion of pages and posts are also possible via the control panel. Section 4.8 covers the details involved with the deletion of posts/pages.

4.3.2 The Toolbox

By default, the toolbox dialog appears when the user makes their first edit. The toolbox provides a user interface for all the editable actions that can be performed on an editable section. Figure 4.8 shows a close-up of the toolbox dialog.

The initial hidden state of the dialog avoids monopolising screen space while users navigate through seamlessly editable web pages. The plugin can be configured, however, to be initially in a visible state. Furthermore, to help the user see more of the content on the web page, the toolbox opacity is set

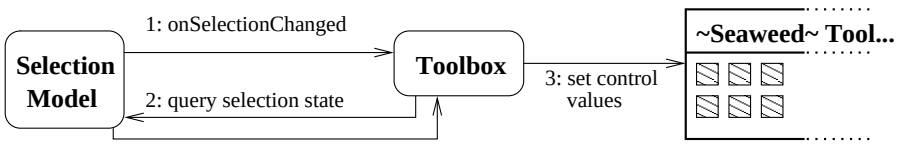


Figure 4.9: WYSIWYG control state management for the toolbox.

to 80 percent by default. Figure 4.2 shows a toolbox with the default opacity, whereas in Figure 4.8 it has full opacity.

The toolbox can be moved around anywhere over the web page, and can also be closed via the close button on the top-right corner of the dialog, or the button labelled "Hide ToolBox" on the control panel. The toolbox can be shown again (or manually for the first time) via the control panel.

The *edit location* presents the current location of the cursor in the document. The preceding part of the label identifies the current editable section. For example, Figure 4.8 shows that the cursor is in an editable section named "Post Content". The succeeding parts of the label reveal the HTML structure within the editable section at the current cursor position. For example, the edit location in Figure 4.8 indicates that the cursor is in a <*span*> element, within a <*strong*> element, within a <*p*> element. The edit location serves as an aid to help identify the structure of the content being edited.

The enable states for the WYSIWYG-style controls reflect the types of actions that can be performed on the currently selected editable section. If there is no selection then all the buttons are disabled, except for the undo and redo buttons (unless there is no undo/redo transaction history). Section 4.5.5 discusses the enable states in more depth.

The toolbox component observes the Seaweed framework's selection model, following a model-view-controller design pattern. Figure 4.9 is a flow diagram showing how the toolbox's WYSIWYG controls are synchronised to the formatting of the selection state. When the Seaweed framework's selection model changes, its observers are notified of the change. When the toolbox is notified, it queries the selection model's selection state. The selection state includes both the current range and HTML/CSS formatting information about the range. The toolbox then sets the button values to match the state of the se-

lection. For example, if the selection state is within an anchor element, the *Unlink* button on the toolbox would be enabled and the *Create/Edit link* button would be in a pressed state to show that the selection is within a link. The selection state in Figure 4.8 is within bold and orange-coloured content (as well as other formatted content), which is reflected by the pressed/value states of the bold and colour controls respectively. The container-type on the toolbox is also set to be "Paragraph" since the selection is fully within a $<p>$ element. The edit-location is updated in a similar way (by observing the selection model).

If there is no explicit text alignment formatting specified, the default is set left aligned if the local direction of the web page is left-to-right. Otherwise if the local direction is right-to-left, right alignment is set. The local direction is detected by the Seaweed framework.

4.4 Establishing Editable Content

Although WordPress has many types of content, four of the most common types of content were chosen to be seamlessly editable: post/page titles, post/page content, comment authors and comment content (the post and page content are essentially the same from the back-end's perspective). This section explains how the Seaweed plugin creates editable sections used for making specific content editable, then presents the approach used for visually marking the editable content on the client-side.

4.4.1 Creating Editable Sections

The WordPress server-side API provides a mechanism (called *hooks*) for linking into the WordPress system without having to change WordPress' code. There are two classes of a hook in WordPress: filters and actions.

- An action hook notifies registered listeners whenever a specific action occurs. For example, when WordPress initialises when processing an HTML request, an action hook called "init" is raised and all registered

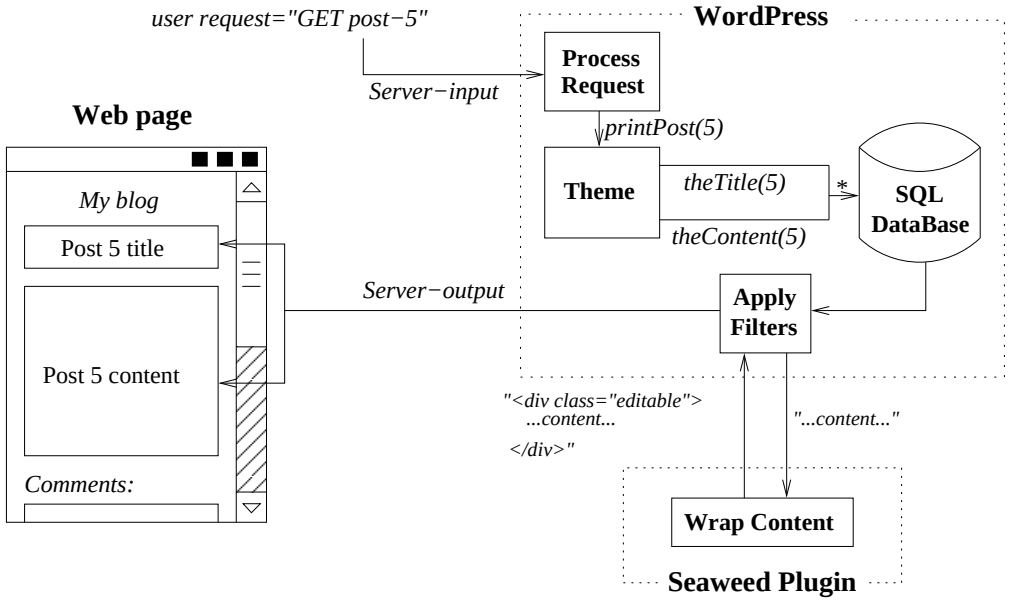


Figure 4.10: Wrapping editable content on the server-side.

hooks are executed.

- A filter hook manipulates content of a specific type created for a requested blog page. For example, whenever a comment's content is about to be written to output, filter hooks can change the comment's content, such as censoring cuss-words.

The layout and presentation of a WordPress page is controlled by a theme. Themes are server-side programs that are executed by the WordPress system. Themes must use the WordPress API in order to output blog content such as posts. When a theme prints a post title, it uses a method in the WordPress API called *theTitle()*, which applies title-related filter hooks, then prints the title to output.

Figure 4.10 shows the data-flow diagram of a user requesting and receiving a result for a blog page containing a post identified with an ID number of 5. Posts, pages and comments are all identifiable by ID numbers. The figure shows how the theme uses methods in the WordPress API to retrieve a post's title and content. These methods query an SQL database containing the raw content, then applies all registered filters to transform the raw content before writing it to output. The figure only shows one of Seaweed's filters, but in

```
<div class="editable-postContent" id="editable-postContent-19">
  ...body-content...
</div>
```

Figure 4.11: Example markup used to wrap post content with editable sections.

practice, WordPress, as well as other plugins, also register filters. Seaweed registers filter hooks for wrapping all editable content with editable section elements. These hooks are registered so that they are the last filters to be applied, to account for cases where other filters may strip/change the HTML of the editable section wrappers.

On the client-side Seaweed's plugin application and the Seaweed framework can then identify the editable sections. The following three sections discuss Seaweed's filtering processes for each type of editable content.

The Content Filter

Post and page content filters wrap the HTML content with a * * tag. Figure 4.11 shows an example of wrapped content. The ID number to which a wrapped post content belongs is encoded in the *id* HTML attribute of the * * tag. Figure 4.11 displays a wrapper for a post content of 19. The encoded ID is required in order for the client-side code to determine which post to save edited data for. A * * tag was used to support any type of element found in a post without producing invalid markup. CSS styles for the editable section classes explicitly set style properties such that the appearance and layout of the web page is not effected by the presence of the wrappers.

Many WordPress themes display "teaser" content for posts, where a small portion of the full HTML content for a post is displayed. Visitors can click on the teaser posts to navigate to the full versions. In these cases the Seaweed plugin avoids wrapping the downsized content to prevent users from overwriting the full-versions of posts with teasers.

The filter for post and page content only wraps the HTML if there is a logged-in user requesting the content, and has the permissions to edit the

```
[editable-postTitle-19]My post[/editable-postTitle]
```

Figure 4.12: Encoded text used to wrap post titles.

content.

The Title Filter

WordPress strips HTML tags from titles after all filters have been applied, just before writing the title to the output. Seaweed's title filter wraps the titles with encoded text, resembling a bulletin-board style syntax, as opposed to using HTML so that the wrappers are not stripped out by WordPress before being written to output. The post ID that the titles belong to are encoded within the opening tag of the wrapper. Figure 4.12 shows an example of a wrapper for a title with content "My post" belonging to a post with an ID of 19.

When the web page loads on the client-side, the Seaweed plugin pre-processes the web page, replacing all occurrences of encoded wrappers for editable titles with editable section elements. This pre-processing phase occurs before the web page is first rendered, so users will not see the encoded wrappers.

The title filter only wraps content for users who are both logged in, and have the permissions to edit the post to which the title belongs.

The Comment Filters

A comment is comprised of two separate parts: the author who wrote the comment and the actual comment's content. Each part has its own filter. Both of the comment filters used for creating editable sections use the same approach as previously described for the post content.

(a) Before mouse hover. (b) On mouse hover.

Figure 4.13: A standard visual indicator.

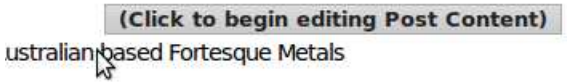


Figure 4.14: Visual indicators for large editable sections.

4.4.2 Visual Indicators

A web page generated by WordPress and the Seaweed plugin contains a combination of editable and read-only content. Visual indicators are displayed whenever the user hovers over the editable content with the mouse pointer. These indicators help users distinguish between editable and read-only content, as well as make associations between the editable content with the types of content they represent in WordPress.

Figure 4.13 shows a sequenced screen-shot of a mouse hovering over an editable title for a post. A tip-tool text style message is displayed above the title identifying the type of editable section (in this case, a post title). The area that the editable section occupies is outlined and highlighted by manipulating the CSS via JavaScript. The indicator is only shown for editable content if the content is not already selected. Thus, when an editable section is clicked, a cursor appears inside it (setting a single-point selection), and the indicator disappears.

The visual indicator in Figure 4.13 is not suitable for large editable sections such as post/page content. Consider the example of a web page where the scroll position is in a state such that the top of a large editable section is not in view. In this case the tip-tool text style message would not be visible to the user since it would appear above the editable section. Furthermore, highlighting large editable sections on mouse hover events tends to distract users when browsing the page. To address these issues, an alternative indicator is used for

large editable sections. Figure 4.14 presents an example of these alternative indicators. The indicator appears when the mouse pointer remains stationary over a large editable section for one second (and the Seaweed cursor is not already placed within the editable section).

Two benefits for dynamically displaying visual indicators — as opposed to a static approach, such as drawing a box around every editable section — are:

1. It avoids creating a distinction between edit mode and view mode. A static approach would change the appearance of a page when using the plugin.

2. More information can be provided about the editable sections with a dynamic approach (in this case tip-tool texts).

A first-time user may find it difficult to identify the editable parts of a web page since the indicators are only displayed when the mouse is hovering over them. Although initially, users will have to discover the editable parts for themselves,[4] they could predict what is editable based on their knowledge and experience of their WordPress blog. Users will know what parts of their web pages are post/page/comment content as they are the authors of such content, and so they are familiar with the way their blog's theme presents their content. Once a user discovers that a piece of content is directly editable, it will be natural for them to extrapolate that all other types of content on their blog's pages are also editable.

4.5 Editing on the Client-side

This section covers developments for supporting GUI-centric editing facilities on the client-side, including editing of links and mixed media. These topics are then followed with a description of the spell checking feature developed for the Seaweed plugin, and ends with an approach for restricting editing actions on the client-side.

[4] Assuming that they do not read the system manual.

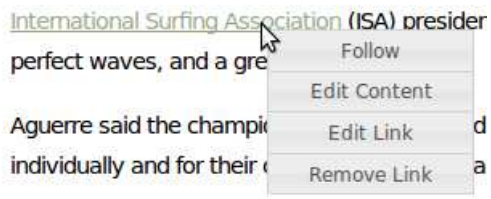


Figure 4.15: An example of a context menu for editing links.

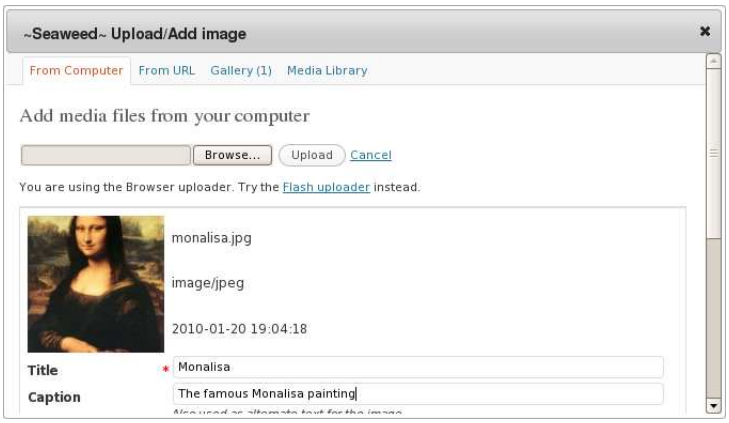


Figure 4.16: Image upload dialog.

4.5.1 Editing Links

The editing of links within an editable section is controlled by the use of a context menu. Due to limitations of the DOM, it is impossible to reliably determine whether a mouse click event is cause by a left or right button. Furthermore, there is no way to reliably extend native context menus for all web browsers. To work with these limitations, whenever the user clicks on a link when using the Seaweed plugin (no matter which mouse button is pressed), a custom DOM-based context menu is revealed (as well as cancelling the navigational action), as shown in Figure 4.15.

4.5.2 Editing Images

The plugin supports inserting and editing images within editable content.

Inserting New Images

Seaweed's toolbox provides a button for inserting images. When this button is clicked, a dialog appears as shown in Figure 4.16. The dialog displayed contains options for inserting images. It is identical to the standard image uploading dialog used in the back-end view for WordPress. This was accomplished by wrapping an <*iframe*> within a custom dialog which can be closed, and setting the <*iframe*> element's *src* attribute to a URL which provides the inner content. The URL is a standard WordPress PHP page hosted on the server-side. The developers of WordPress designed the forms to be used inside an <*iframe*>. When the user clicks an insert button within the wrapped <*iframe*>, JavaScript within the <*iframe*> invokes a global JavaScript method in its parent document, supplying the HTML markup for the image to be inserted. The Seaweed plugin application declares this method in the global scope to intercept the HTML markup. The markup is then inserted at the current cursor position using the *InsertHTML* action via the *undo manager* in the Seaweed framework.

Editing images via the Seaweed Framework

Images can be manipulated using the Seaweed framework in a limited way. Images can be deleted by placing the cursor next to an <*img*> element and pressing the delete or back-space key. They can also be deleted by selecting a range in which they are included, then deleting the range. The delete actions can be undone to recover the deleted images.

Enhanced Image Editing Features

In WordPress images can be inserted with or without captions. An image with a caption is comprised of three to four elements: the <*img*> element displaying the image, a <*p*> element containing the caption text, and a <*div*> element that wraps the <*img*> and <*p*> elements as well as providing a border around the content, as shown in Figure 4.17(b). Optionally images can be linked with an <*a*> element. A decision was made against packaging the captioned image

(a) Without a caption. (b) With a caption.

Figure 4.17: Captioned and non-captioned images.

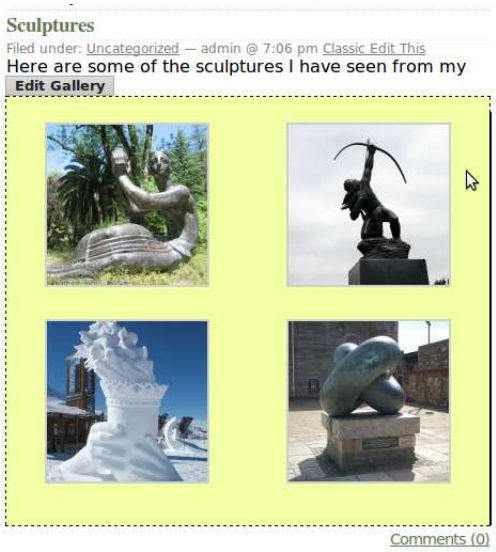


Figure 4.18: Visual indicator for a gallery in Seaweed.

nodes, which would prevent users from "tearing" up the captioned images, in favour of allowing users to directly edit the caption text.

An image dialog is used for editing image attributes that cannot be edited directly via the Seaweed framework, such as the *width* and *height* attributes, as well as creating captions for images that do not have a caption. The dialog is accessible via a context menu. Figure 4.17(a) shows an example of a context menu, which is shown when the image is clicked on.

4.5.3 Editing Shortcode Content

A WordPress post/page can contain HTML generated from content written

```
[gallery columns="2" orderby="rand"]
```

Figure 4.19: Shortcode example for a gallery.

in a bulletin-board style syntax called *shortcode*. An example of a type of shortcode is the *gallery*. Figure 4.18 shows an example of a gallery, displaying an array of images within a post. They are comprised of multiple DOM nodes, including <*img*> nodes for each image, and embedded <*style*> elements for formatting the presentation. HTML content for galleries are generated from shortcode within post/page content, Figure 4.19 reveals the shortcode used for generating the gallery HTML in Figure 4.18. Shortcodes can have options, and can be mixed with HTML content. Two options are set for the shortcode in Figure 4.18: "columns," the amount of columns to display all images in a gallery, and "orderby," the order in which the images should be placed (either in ascending, descending or random order).

The Seaweed plugin packages galleries such that the inner content cannot be edited directly using WYSIWYG actions: they can only be fully removed via actions that delete content. If the user could directly delete the two images on the right of the gallery in Figure 4.18, then the HTML content of the gallery must be re-downloaded to display the new table layout for the remaining two images (where they would become horizontally adjacent to each other rather than vertically) since the HTML must be generated on the server-side. The user would not be able to make any further edits until the new HTML is downloaded. The negative factors (such as code complexity, room for bugs, increased download bloat) outweigh the positive factors for implementing support for directly editing content.

The HTML content for captioned images is actually generated from shortcodes called "caption." Captions and galleries are the only shortcodes shipped with WordPress. Other shortcodes are provided by plugins. For example, a plugin called "Viper's Video Quicktags" generates HTML for embedding videos (such as content from YouTube) from shortcodes.

All shortcode generated content, except for captions, are manipulated in

the same way: visual indicators in Seaweed are shown when a mouse pointer hovers over all shortcode-content — following the same convention as previously described for editable sections (Section 4.4.2). These are used to help indicate that the content is not directly editable (with the cursor/selection). Figure 4.18 shows an example of an indicator displayed for a gallery.

In the WordPress back-end view, the visual post/page editors do not render the generated content for shortcodes (except for captions) — but instead just display the raw shortcode markup. The shortcodes can be manually written by users, as they are designed to be easy to remember. Some plugins provide custom dialogs for generating the shortcodes for users. There is no generic way of using these dialogs in Seaweed. Although it is possible to extend the Seaweed plugin to provide a dialog for editing shortcode content (like it does for captions and galleries), it in infeasible for the Seaweed client-side application to be extended to support every existing plugin for WordPress as there are thousands of plugins available (and more are released daily). Thus, to insert plugin-powered content via Seaweed, a user must type the shortcodes themselves. The next time a page is loaded after a user saves a post containing shortcode, the (packaged) generated content is displayed in place of the shortcode.

4.5.4 Spell Checking

Any modern document editor is expected to support spell-checking features, as many users rely on them, especially in contexts such as editing blog content. To meet this expectation, spell checking features were implemented for the Seaweed plugin.

Some web browsers support spell checking within textual input controls such as *<textarea>* elements. Seaweed cannot benefit from browser-based spell checking facilities, so a custom JavaScript implementation was needed.

Marking Spelling Mistakes

The Seaweed toolbox provides a spell-check button, which when pressed, performs a server-side based spell checking routine that visually underlines all the

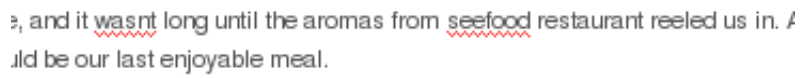


Figure 4.20: Rendering spelling mistakes on the client-side.

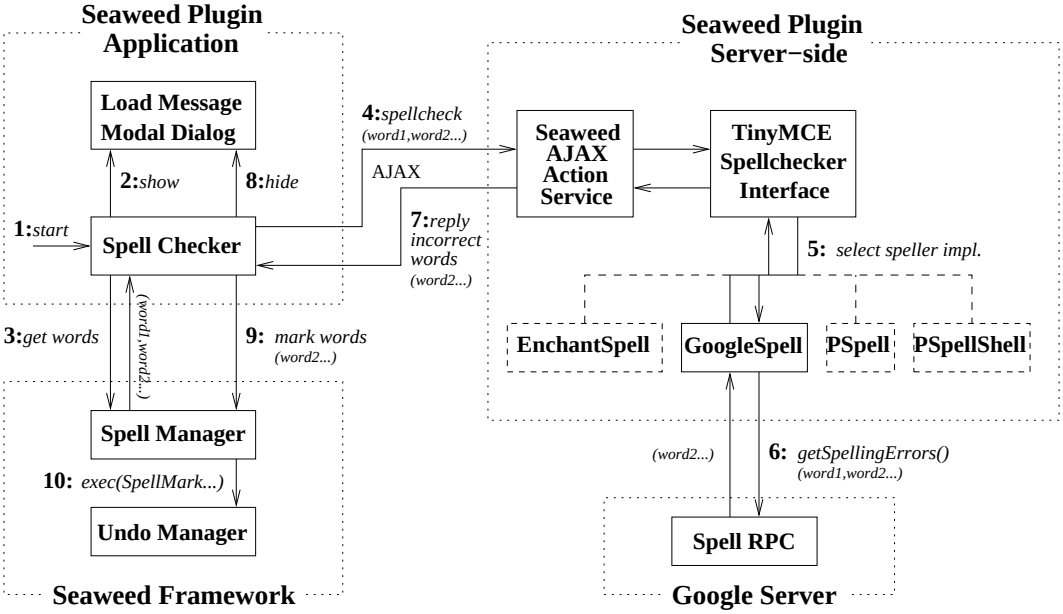
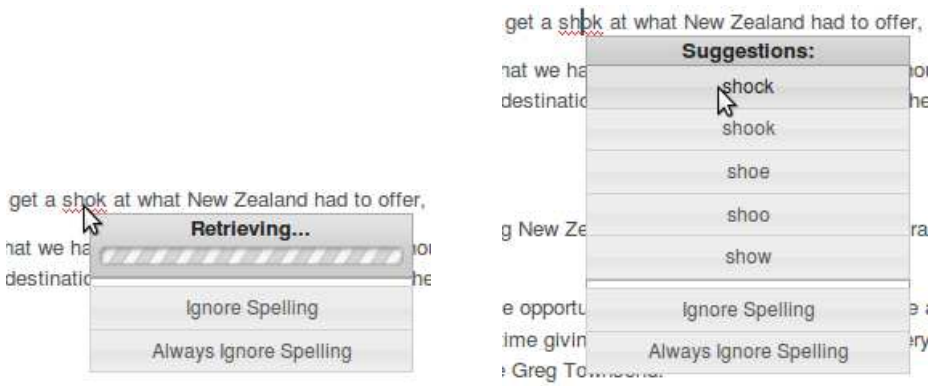


Figure 4.21: Client/server communications for spell checking.

misspelled words within editable sections on the web page. Figure 4.20 shows an example of the end result of a spell-checking routine.

Figure 4.21 presents the sequence of steps carried out by various components of the Seaweed plugin for marking spelling mistakes. Once the user clicks the spell-check button, all the words within all editable sections on the page are extracted via the *Spell Manager* (a component within the Seaweed framework). The words are then sent via AJAX to the *Seaweed AJAX Action Service* provided on the server, to remotely perform an action named "spellcheck." The Seaweed plugin uses TinyMCE's Spell Checker Interface[5] to spell-check the words. The interface may use one of four possible implementations for performing the spell check, which depends on the Seaweed plugin's spell-checker settings. Once a result is obtained, all the incorrect words are sent back to

[5]The interface was salvaged from TinyMCE's open-source spellchecker plugin, see http://wiki.moxiecode.com/index.php/TinyMCE:Plugins/spellchecker for more information.



(a) Initial display. (b) One received suggestions.

Figure 4.22: Spelling context menu for Seaweed.

the Seaweed client. During this period the web page is dimmed, showing the message "Spell checking..." along with an in-determinant progress-bar — preventing the user from making any edits on the web page. Upon receiving a reply this message is hidden and the user can continue to edit. For every incorrect word, a *SpellMark* action is used to wrap misspelled words with *spelling-error elements*. These are <*span*> elements containing a class name called "sw-spell-error", where CSS is used to render red squiggled lines beneath the words. All of these actions are executed via the Seaweed framework's *undo manager* since they manipulate the DOM. The actions are grouped together, so all of the DOM manipulations can be undone with a single undo.

Providing Suggestions

When a user clicks on a spelling-error element, a context menu is displayed, as shown in Figure 4.22(a), and an AJAX action is sent to the *Seaweed AJAX Action Service* for retrieving spelling suggestions on the clicked-on word. The server returns a list of suggestions and the context menu is updated to display them (but only if the user has not hidden/cancelled the menu before the response), as shown in Figure 4.22(b). The user can then click a suggestion, which replaces the misspelled word (again via the Seaweed framework's *undo manager*). In the case where there are no suggestions, the context menu shows a message indicating that there are no choices.

If users do not want to use suggestions, but instead manually correct

spelling mistakes themselves, they can manually make corrections via the cursor and keyboard even when the context menus are displayed.

Ignoring Marked Mistakes

Spelling-mistakes can be ignored in two ways via a spelling context menu (as shown in Figure 4.22): a user can choose to ignore either a single instance, or all occurrences of a misspelled word.

Ignoring a single instance of a spelling mistake removes the spelling-error element used to mark the mistake via the *undo manager* using a *SpellUnwrap* action. Ignoring all occurrences of a spelling mistake uses the *SpellUnwrap* action against all words that match specific mistake. All executed *SpellUnwrap* actions are grouped together so they can be undone in a single undo.

4.5.5 Restricting Edit Actions

On the server-side, whenever content is saved to the WordPress database, WordPress "sanitises" all HTML. Tags and attributes that are not permitted for the specific content type are stripped. For example, WordPress does not allow any formatting for page/post titles, thus strips all HTML tags from a title.

There is a chained relationship between HTML sanitising in the WordPress system, action filtering in the Seaweed framework, and the GUI control states for Seaweed's toolbox. For example, WordPress strips all block-level, list and table HTML tags from comment content, except for <*p*> and <*blockquote*> tags. Therefore, the *actionFilter* property for content-comment editable sections prevents actions that generate content which would be stripped. For example, the *Itemize* action is filtered out since it creates list HTML tags. Whenever a user places a cursor within an editable section for a comment's content, all WYSIWYG controls in the toolbox GUI that represent a Seaweed framework action that is filtered out are disabled. In the case of the selection being within a comment's content, the buttons used for creating bullets and numbers are disabled (amongst others).

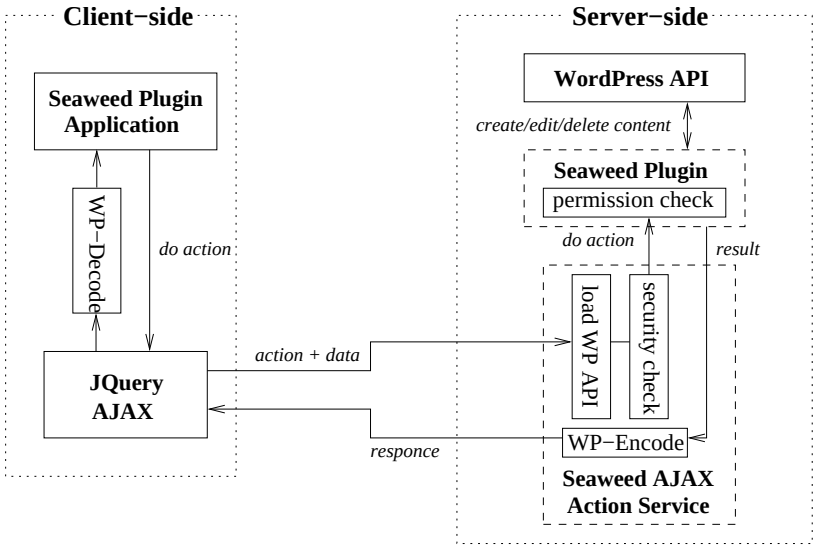


Figure 4.23: Client-server communication architecture.

4.6 Asynchronous Content Management

Asynchronous saving is a key element to provide a seamless editing environment. This section takes a closer look into the client-server communication architecture developed for the Seaweed plugin, and explains the transformations needed to make the editable HTML compatible on the client-side with WordPress so that the HTML can be saved into the WordPress database.

AJAX was the obvious choice as the communication protocol for passing asynchronous messages. Figure 4.23 presents the architecture design for the Seaweed plugin's communications. The *Seaweed AJAX action service* is a secure web-service, providing an interface for performing server-side actions via AJAX. These actions includes saving, creating and deleting content, as well as other miscellaneous actions such as spelling-checking (as previously described in Section 4.5.4). jQuery, a readily available API that is part of the GUI framework, is used to providing cross-browser AJAX communications on the client-side. jQuery encodes data sent to the server as standard HTTP request variables. To parse responses from the server, WordPress' standard AJAX message library was used.

4.6.1 Permissions and Security

When the *Seaweed AJAX action service* receives an action from the client, the WordPress API is loaded, which in turn loads the Seaweed plugin, in preparation to perform a remote action (see Figure 4.23). Before an AJAX action is carried out by the Seaweed plugin, a security check is performed. WordPress uses *nonces* (number used once) for security checking. When a web page is generated for a user who is logged into WordPress, the Seaweed plugin embeds a nonce code into the web page, which is generated for the Seaweed plugin via the WordPress API. Whenever the client sends an action to the server, the nonce code is sent with the AJAX action data. The *Seaweed AJAX action service* extracts the nonce code from the AJAX data and verifies the code via the WordPress API. If the verification fails the action is rejected.

Once an AJAX action request is verified, and is a valid request, the *Seaweed AJAX action service* performs the action via the loaded Seaweed plugin. The plugin only performs the action if the user who sent the AJAX action is logged in, and has permissions to perform the specific action. The permissions are determined via the WordPress API's capability system.

4.6.2 Round-trip Compatibility

The HTML presented on the client-side does not have a one-to-one mapping with the raw form on the server-side. When transmitting HTML content back to the *Seaweed AJAX action service* for saving changes, a series of transformations must be performed to prevent loss of data.

Preserving Shortcodes

When saving a post or page containing shortcode generated content (see Section 4.5.3), the original shortcode that generated the HTML snippets must be saved instead. Preserving shortcodes is necessary because of the following reasons:

- The user can adjust settings for plugins used to generate the HTML for shortcodes, such that the HTML is generated differently. For example,

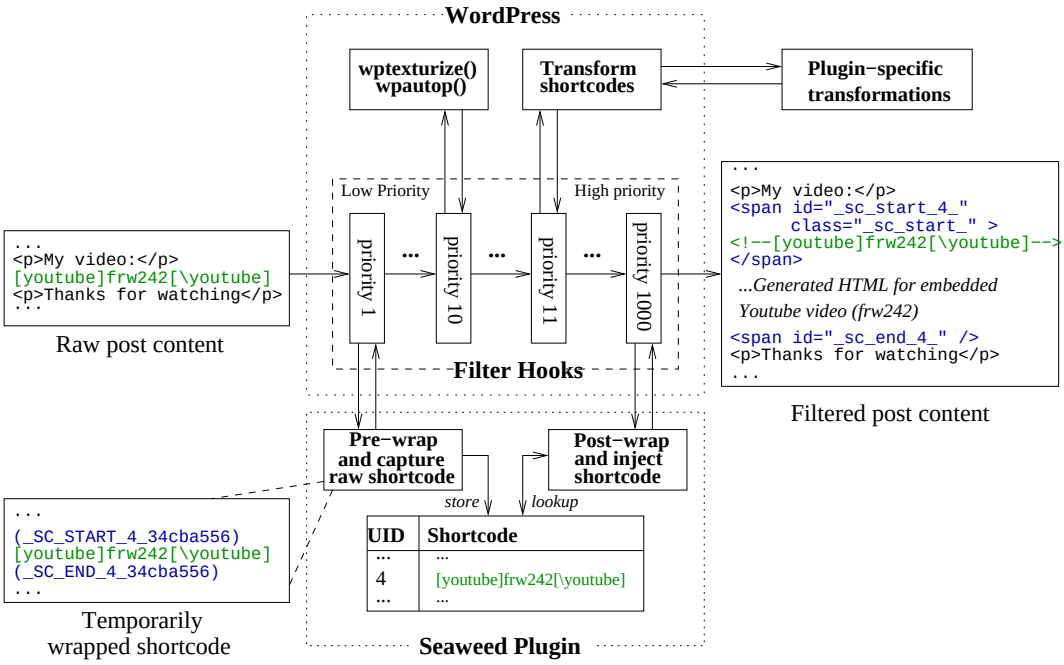


Figure 4.24: Content filter-hooks used for sending raw shortcode with generated content.

the appearance settings for a plugin used to embed audio-player widgets within posts or pages could be tweaked to include extra playback-control buttons.

- Plugins can be upgraded to new versions, so that HTML generated by plugins contain the latest improvements and bug fixes.

- When saving content, the WordPress HTML sanitiser can strip certain tags for shortcode generated HTML. For example, many interactive shortcode content embeds JavaScript, inline within the content, which would be removed by the sanitiser since <*script*> tags are not permitted.

To preserve the original shortcodes, the generated HTML content and the original shortcode markup are sent together when the server produces a web page. Therefore when saving content on the client-side, all HTML generated from shortcodes can be replaced with the original shortcode markup.

To include the original shortcodes with the generated content, the Seaweed plugin applies two filter hooks. These hooks are registered for page and post body-content, since they are the only types of content which support short-

codes. Figure 4.24 presents the shortcode filtering process: starting with a snippet of example raw content on the left, containing shortcode for embedding a YouTube video, and ending with the final filtered result on the right.

When registering a filter hook, a priority level can be declared. Low priority filters are applied first, and high priority filters are applied last. The Seaweed plugin uses these priorities, and with the knowledge of priority levels for certain WordPress filters, to apply the two filters used for preserving shortcodes at specific times during the filtering process. Using Figure 4.24, this process is explained in turn:

1. The first filter involved in the shortcode preservation process has two purposes: capturing raw shortcode before it is transformed by the shortcode filter, and placing temporary wrappers around the raw shortcodes. A UID (Unique ID) is generated for each shortcode and is encoded within the temporary wrappers. A copy of the raw shortcode is stored in a collection for later reference, mapping UIDs to raw shortcodes. As an extra precaution, in a case where the raw unfiltered content contains a sequence which happens to be in the format of a temporary wrapper, a unique hexadecimal string that does not occur within the post content is also included in all the temporary wrappers.

2. Various other filters may be applied straight after the first filter (depending on the WordPress configuration). Two common standard filters are *wptexturize*, used for Unicode formatting, and *wpautop*, used for creating paragraphs for content not already in a block-level element. The latter of these filters can sometimes encapsulate the temporary wrappers within their own paragraph. These extra paragraphs have to be removed, and are done so on the client-side.

3. WordPress then applies its shortcode filter, where all shortcodes are replaced by generated HTML. Shortcodes generated by plugins transform the HTML via this filter (using WordPress' shortcode API) as opposed to transforming the shortcodes directly via their own content filters.

4. When all filters are done, Seaweed applies a shortcode injection filter. This filter parses all temporary wrappers created in the first filter, and replaces them with two separate <*span*> elements. <*span*> elements had to be used as opposed to a <*div*> element to ensure the HTML remains valid — since shortcodes can occur within elements such as <*p*> elements, which do not permit block-level elements. Two separate <*span*> elements were necessary instead of wrapping the shortcode within a single <*span*> to ensure the HTML remains valid — since the generated HTML for the shortcode being wrapped can contain block-level elements.

   The wrappers contain the UIDs to identify the start and end <*span*> elements for a single transformed shortcode. The raw shortcodes of the wrapped transformed shortcodes are inserted as HTML comments within the starting <*span*> that make up the wrapper elements. The raw shortcode is retrieved via looking up the UID-to-shortcode map created in the first filter using the UID extracted from the parsed temporary wrappers.

On the client-side the web page is pre-processed before rendered. Every shortcode tuple (start wrapper, generated shortcode HTML, end wrapper) is encapsulated within either a single <*span*> or <*div*> element. A <*span*> is used if a tuple contains all inline-level elements, otherwise a <*div*> is used. Extra paragraphs introduced from the *wpautop* filter on the server-side are removed. The element used for encapsulating the shortcode tuples are packaged to prevent users from "tearing" up the tuples and directly editing the generated HTML shortcode content (see Section 4.5.3 for details about how these packaged elements are edited).

Saving Changes

When a user wishes to save changes (or save a new page/post), the save process is performed asynchronously via the *Seaweed AJAX action service*. The client-side HTML goes through the following transformations before being sent to the server:

- HTML for galleries and captions are replaced by their shortcode equiv-

alents. The shortcode is generated on the client-side, which is possible since the specification of the shortcode syntax for galleries and captions is available.

- Wrappers used for encapsulating plugin generated shortcodes are replaced with the original shortcode. The original shortcode is extracted from the comments residing within the shortcode wrappers.

- Elements called "more tags", used for visually marking where "teaser" content should end, are replaced by HTML comments containing the text "more" — the convention used by WordPress for expressing more tags.

- Spelling error wrappers are removed.

- Highlighted content used for rendering selection by the Seaweed framework is stripped.

The WordPress HTML sanitiser only accepts colour CSS information in hexadecimal notation. Web browsers that supply HTML content via the DOM may express colour CSS values in either RGBA or hexadecimal notations. To avoid losing colour formatting when saving, the server-side parses the HTML, replacing RGBA CSS colour notations with their hexadecimal equivalents.

4.7 Creating New Posts and Pages

New posts and pages can be created using the Seaweed plugin. An ideal way of creating a new post would be to create a skeletal layout for a new post directly in the same web page — where a post would usually be displayed — so users could then begin writing their content in a context as if it were published. However, due to the vast variety of HTML layouts employed by themes, and that there are many different views where posts are not displayed (such as a single page presenting contact information), a less seamless approach was taken.

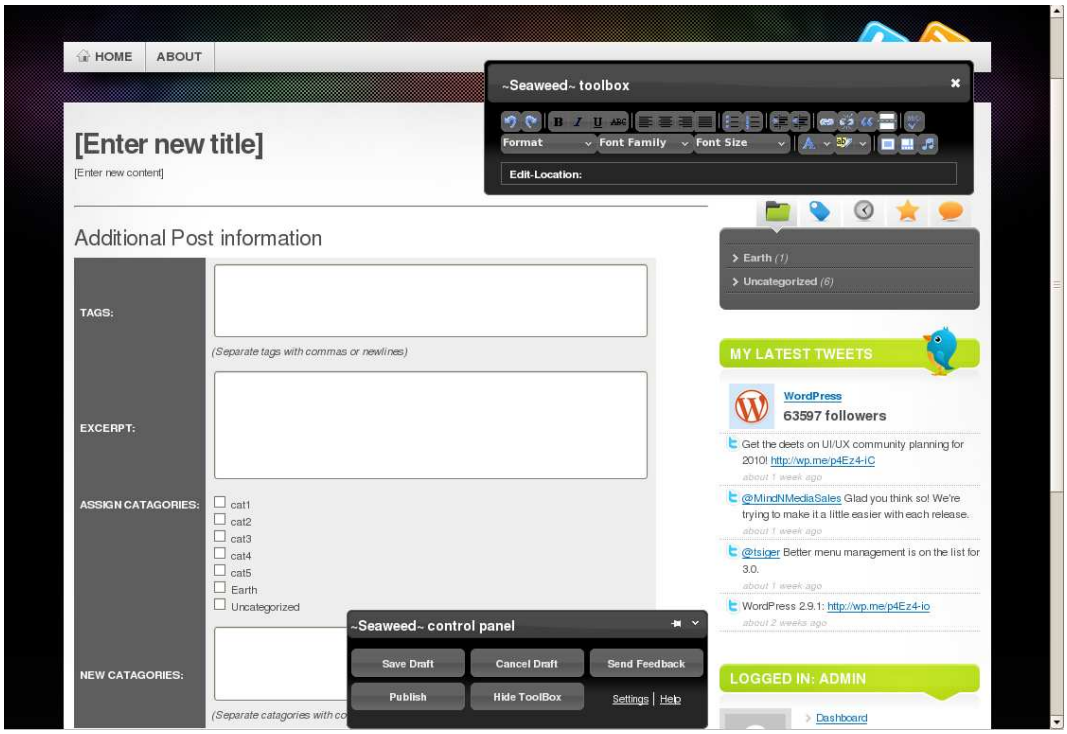


Figure 4.25: Skeletal layout generated for a new post.

When a user requests a new page or post, they are directed to a web page displaying a skeletal page/post. The skeletal content is generated by the Seaweed plugin. Figure 4.25 presents an example of a new post. The web page is constructed using a PHP script to create a skeletal layout, such that the whole web page appears as close to the real-time published view as possible. Every theme has its own PHP script tailored for laying out the skeletal pages. Although the skeletal PHP scripts are small and simple to create, it is infeasible to write one for every WordPress theme available. Therefore, a generic PHP script is used to provide skeletal content with themes that have no specially tailored script. In some cases, the layout produced by the generic skeletal PHP script can look disorganised, nonetheless is still functional.

Extra metadata fields are provided when creating new posts. These are fields that are not seamlessly editable, such as categories and tags.

When the user is creating a new post or page, the content is in a *draft* status. The first time a save is made, the server returns an ID generated by WordPress, which is used for subsequent saving, publishing and deleting actions within the session. The save action for drafts is the same for published

content, where the process is asynchronous.

When the user is ready to publish the post to the public, a publish action is sent to the *Seaweed AJAX action service*. If the publish action is successful, the page is redirected to the published view of the post.

If the user wants to cancel the draft (that is to say, delete it), if they have not made a save on the draft they are redirected to their blog's main page (at the front-end view). However, if they have made a save, a delete action is sent to the *Seaweed AJAX action service* to remove the draft in the back-end before redirecting them.

4.8 Deleting Posts and Pages

Users can delete a page or post via the control panel, but only if a single page or post is displayed on the web page. When deleting a post, a "deleting..." message is displayed to the user in a model dialog that cannot be closed by the user. An AJAX action is sent to the *Seaweed AJAX action service*, and the result is then presented to the user. If the result is a success, the browser redirects the URL to their blog's main page (in the front-end view), since the currently displayed post in the web page no longer exists.

4.9 Summary

This chapter presented the development process for integrating the Seaweed framework into a CMS called WordPress. The development process provides a guide for making other content management systems seamlessly editable using the Seaweed framework. The following obstacles, common amongst content management systems, were addressed:

1. Providing a GUI on the client-side (Section 4.3).

2. Knowing what parts of a web page should be made editable, and how to make them editable (Section 4.4).

3. Managing rich editing commands in the client-side (Section 4.5).

4. Handling security and permissions (Section 4.6.1).

5. Transformation of content from a presentation form in the client-side, back into a raw form for the server-side, in order to preserve data for saved content (Section 4.6.2).

6. Creation of new documents (Section 4.7).

7. Deletion of existing documents (Section 4.8).

Although the implementations for addressing the obstacles above in this chapter were specific to WordPress, the general approaches can be adopted for supporting seamlessly editable environments in other content management systems. The next chapter evaluates seamless editing, where the Seaweed plugin developed for WordPress is used in two observational studies.

Chapter 5

Evaluation

Two different types of observational user studies were performed for evaluating the concept of seamless editing on the web. This chapter begins by outlining the set of research questions that were established for guiding the design of both of the studies. Section 5.1 discusses the reasoning behind the studies. The first study conducted was a prescribed study: where the participants followed a set of tasks using temporarily assigned blogs. Section 5.2 details the design for the first study, and is followed by a discussion of the observational data and participant feedback in Section 5.3. The second study was an unprescribed study: where participants were naturally observed using seamless editing in their own WordPress blogs. Section 5.4 details the design for the second study, and is followed by a discussion of the observational data and participant feedback in Section 5.5. Lastly, Section 5.6 summarises and concludes the findings for both of the studies.

The following set of research questions was established to direct the design for each of the user studies:

1. What are the situations in which people prefer using Seaweed over using an external WYSIWYG editor? And what are the situations in which they do not?

2. What are the situations in which people prefer using Seaweed over writing raw HTML markup? And what are the situations in which they do not?

3. How intuitive is Seaweed?

4. How well people who have substantial experience with the traditional way of editing in WordPress adapt to seamless editing?

5. Do people who access the web, like the concept of seamless editing?

6. What are other contexts where people who access the web, could see seamless editing being helpful (other than blogs)?

Gathering feedback on usability aspects of the Seaweed plugin and its stability (that is, identifying bugs/quirks in the software) were also incorporated in the designs of the studies. Although this type of feedback is not directly related to the research questions formulated for the study, usability and system stability issues can potentially be explanatory variables for responses given by participants.

The outcomes from both of the studies showed that the participants generally preferred the Seaweed plugin over the WYSIWYG and HTML source editing facilities in WordPress, as they all liked the concept of seamless editing and generally found it easy to adapt to the new way of editing. They found the ability to seamlessly edit content particularly valuable for making small sized edits on published content. Both studies indicated that some people prefer HTML editors over visual editors because they like to have complete control over the HTML source. The results also suggested that the ability to create new posts in the context of a published view is useful.

5.1 Two Observational User Studies

Two studies were conducted for addressing the research questions established for evaluating the Seaweed software and the concept of seamless editing: a study using participants who did not own their own WordPress blogs, and another study using participants who did.

To study use of the Seaweed software in a natural environment, both studies observed participants using WordPress and the Seaweed plugin both in their

own time, and in the comfort of their physical environments where they usually access the web. This had the following benefits:

- Observing the participants using their own computer avoided biases introduced by hardware and software setups different from what the participants were accustomed to. Different hardware setups — such as the sizes/resolutions of computer screens, or the size and shape of the keyboard and mouse — may have became troublesome for participants as they partook in the study. For software, the differences in the way the operating system manages windows for web browsers, as well as the GUIs of the web browsers, may have made it more difficult for users to use the Seaweed plugin than it would otherwise had been if they used their own computer.

- The studies included real-world influences that would have been unobtainable in a controlled environment, such as a lab. An *ecological gap* occurs when observations from controlled studies cannot be validly generalised to the population or situation of interest due to the absence of real-world variables [32]. For example, a controlled study for observing participants using the Seaweed plugin would miss out interruptions that may naturally occur during a task in the participants own time, such as a phone call at home or casual conversation with a colleague at work. These interruptions test how well the software assists users for remembering how far through a task they were before they were interrupted.

- The studies avoided observer-expectancy effects introduced from the physical presence of an observer or conductor [18].

5.1.1 The Prescribed Study

The first study was controlled, observing participants following a set of given tasks using WordPress. The participants had little or no experience with using the WordPress system. Temporary WordPress blogs were set up for the participants. The study began with two parts where the participants would learn how to use both WordPress and the Seaweed plugin.

The results gathered for the first study represent a more general population than that of the second study, since the second study required participants to have expertise in operating the WordPress system. Furthermore, using participants with little or no experience with WordPress offered an unbiased perspective: they were not directed favourably towards WordPress or Seaweed based on their familiarity of either of the two systems.

5.1.2 The Unprescribed Study

The second study observed real WordPress bloggers "in the wild," offering a perspective on seamless editing from experienced bloggers using the Seaweed plugin. The participants in this study used their own blogs. Unlike the first study, the participants did not follow a set of given tasks, but instead continued to perform their natural *blogging* activities. During the study, the only difference to their blogging activities outside of the study was that they were given a choice to use either the Seaweed plugin or their typical (moded) facilities when manipulating/managing their content.

The first study did not evaluate all the possible tasks that users may typically carry out in a real-world context, since participants were limited to a subset of all the possible tasks which were given to them by the researcher. An unprescribed study is open to evaluating more advanced tasks and/or common tasks that were excluded for the first study, due to the assumption that they were likely to be uncommon. For example, tasks using shortcodes provided by third party plugins within posts were not included in the first study because shortcode was considered to be an advanced feature, and there was no specific third party plugin that could have been considered as being commonly used.

5.2 The Prescribed Study Design

This section describes the design of the first study. It begins with outlining the process that the participants had to follow during the study, and is followed by a description of the raw data captured for the study. Lastly, a brief description of the infrastructure supporting the study is presented, which was used for

supporting the unprescribed study as well.

5.2.1 The Procedure

The procedure for each participant was as follows:

1. An online registration form was filled out. The form obtained their consent, and gathered information about their experience with computers and blogging software, as well as general information such as their age-group. See Appendix D for the full online registration form.

2. A unique ID and private key-code was emailed to the registered participants. The email included information, asking them to wait for further instruction from the researcher.

3. The researcher created a temporary WordPress blog for the registered participants. An assigned username and password was emailed to the participants for accessing their temporary blogs. Each blog was identically configured. The researcher emailed task-sheets to the participants once their blogs were ready for use, instructing them to begin the tasks.

4. The participants worked through the task-sheet. All participants were given the same tasks. Detailed explanations of these tasks are given in the following sections.

5. Once the participants had completed all the tasks, they filled out an online survey, where they reflected on their experiences during the study. See Appendix E for the full online survey.

The tasks were organised into three parts. The following three sections explain these three parts in turn.

Part 1: Learning WordPress

The purpose of the first part of the study was for the participants to learn how to use WordPress. It involved tasks such as the creation of new posts and pages,

as well as editing them after the posts/pages were published. Participants were instructed to complete the first part within a period of one to two days.

The participants had to teach themselves how to use the software in order to complete the tasks. They were directed to an online user manual for using WordPress which they could use to help them progress through the study.

To aid participants with creating content in their posts and pages, blog post exemplars were given to them. They were not permitted to use copy and paste functions for creating their posts and pages, except when posting small snippets as part of a post or page (such as quotations). The purpose of this restriction was to simulate a realistic blogging scenario: as if they were a real blogger creating the content themselves. Creating a realistic scenario would help participants experience a more full and realistic experience of the WordPress software.

Part 2: Learning Seaweed

The participants were instructed to start the second part at least one day after completing the first part. This prevented participants from completing the study all in one sitting to both help create a realistic blogging experience for them, and to avoid them from rushing through the tasks without taking the time to think about their experiences.

The purpose of the second part of the study was for the participants to learn how to use the Seaweed plugin. They followed the same tasks as given in the first part, except using the Seaweed plugin instead. Participants were instructed to complete the second part within a period of one to two days. As with the first part of the study, participants had to teach themselves how to use the software in order to complete the tasks. An online user manual for the Seaweed plugin was made available to the participants. Empowering the participants to take control of their learning process yields more valuable feedback on the Seaweed plugin's intuitiveness, as opposed to a study where the participants are instructed on how to use the software.

Although the Seaweed plugin was installed for the participants, they had to *activate* the plugin themselves. To activate the plugin, a button had to

be pressed in the back-end view. Due to the way the Seaweed plugin was enhanced for capturing data during the study, the participants also had to insert their automatically generated ID and private key-code (as described in the second step of the study procedure in the previous section) into a pop-up dialog (Section 5.2.2 explains why this step was necessary).

Part 3: Blogging

Participants blogged about current events over a period of four days for the final part of the study. They were instructed to create at least one post per day. They had the choice to create and edit their posts using either the standard editing facilities in WordPress or the Seaweed plugin. The purpose of the last part of the tasks was so that the participants could gain more experience with each of the editing systems for WordPress, helping them discover for themselves which of the two they prefer by means of their own experimentation.

As with the first two parts of the study, participants could copy from existing material, such as articles from current event web sites. However, they were not permitted to create posts that would be essentially constructed from using copy and paste functions.

5.2.2 Data Captured During the Study

Registration forms (see Appendix D), survey forms (see Appendix E) and activity logs were collected for qualitative and quantitative analyses. The instant feedback feature described in Section 4.3.1 was developed as a result of a finding in this study, and thus instant feedback was gathered from emails sent to the researcher during this study. The activity logs were captured by the Seaweed plugin. The logs stored changes that participants made to posts, pages and comments. The capturing of activity logs began for a participant once they entered their assigned ID and key-code into the dialog shown when they first activated the plugin for the second part of the study. This step is referred to as *initiating* the plugin. Table 5.1 outlines the metadata captured for the activity logs. The following types of actions were logged for the study:

- Saves for a new draft for posts and pages. These logs included full HTML content for each field in the saved post/page.
- Saves on existing posts and pages. These logs included full HTML content for each field in the saved post/page both before and after the save. The post status before and after the save was also recorded (for example, whether the post was still a draft, or transitioned from a draft to a published status).
- Deletions of posts and pages.
- Edits on comments.
- Initiating the plugin. When the participants initiated the plugin a log was created, serving as a mark in time which they began using the Seaweed plugin.
- Terminating their involvement. If a participant pulled out out of the study, they could explicitly end their involvement which would be logged.

Privacy of Data

All data captured for the study was anonymised to protect the identities of the participants. The activity logs were encrypted since full content of all saved posts/pages — including drafts — were transmitted over the Internet and stored on two web servers that were accessible by people who were not involved in the research. The encryption was a necessary precaution since the participants may have saved sensitive or information revealing their identity despite being warned that all of their activity was being logged during the study. The researcher was the only person who had the ability to decrypt the data.

Protection Against "Spoofed" Data

The entire study was performed over the Internet: a public network open to many potential threats. Only participants invited by the researcher could fill

Data	Description
Participant ID	The generated ID of the participant which the log belongs to.
WordPress server time	The UTC time when the action occurred on the WordPress server.
Research server time	The UTC time when the log arrived on the research server.
Client IP	The IP address of the participant's computer when carrying out the action.
User-agent	The type/version information of the operating system and web browser used to carry out the action.
Checksum	A 32-bit checksum used for validating the integrity of the log data and also serves as an additional security measure.
Content ID	Where applicable, the WordPress post/page/comment ID which the action involved.

Table 5.1: Metadata captured for activity logs.

out the registration forms for taking part in the study, as opposed to allowing anyone on the web to register. A security protocol was designed (using the automatically generated ID and private key-codes) for preventing the possibility of receiving bogus activity logs.

5.2.3 The Supporting Infrastructure

Figure 5.1 reveals the underlying infrastructure developed for supporting both the prescribed and unprescribed studies. The figure shows the Seaweed plugin "sniffing" all editing actions on the server-side that occurs in the WordPress system, such as the action of saving a post. A component, labelled as the "Activity Logger," stores encrypted information about the action into two databases: a local database automatically created by the Seaweed plugin for the participant, and a remote research server which stores data for all participants in the entire study within a centralised database. Once a participant finishes their survey, the research server downloads a web page from their

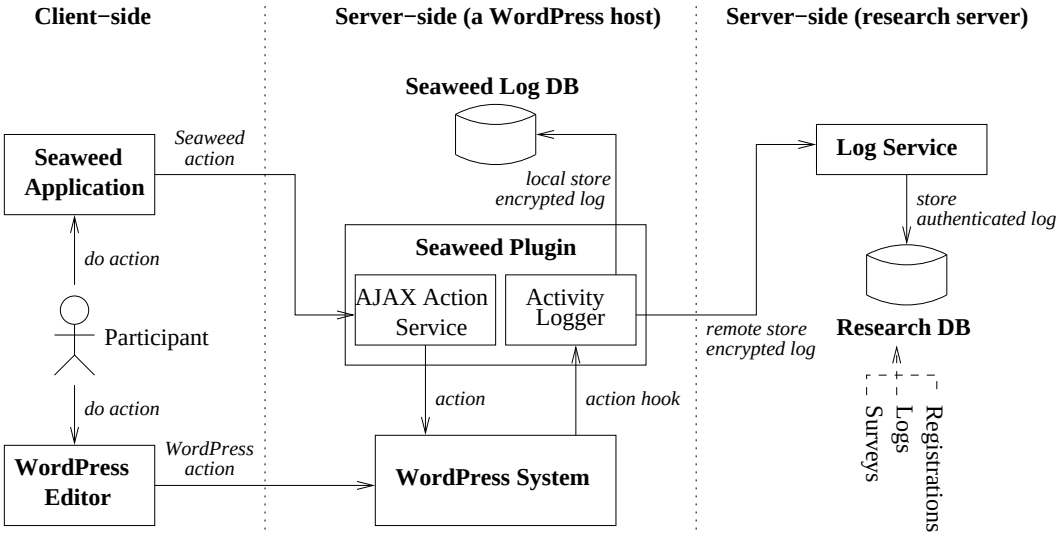


Figure 5.1: Underlying infrastructure for both studies.

assigned WordPress blog, passing a special URL-encoded variable in the request so the Seaweed plugin embeds all locally stored encrypted logs into the page encoded as HTML comments. Both methods for storing logs were used since neither of them were completely reliable,[1] thus maximising chances of capturing all activity logs during the study.

Figure 5.1 shows a single participant being observed during the study. However in both of the studies, multiple participants were observed simultaneously. In the prescribed study, a single web server was used for hosting all WordPress blogs, using a modified version of WordPress called WPMU (WordPress Multi-User).[2] In the unprescribed study there were many web servers involved, as each participant had their own WordPress server for hosting their blog.

The Seaweed Web Site

A web site was created for supporting both of the studies. The web site hosted information about the study and included an online user manual for the Seaweed plugin. The web site also provided online forms for the registration

[1] Due to possible network failures, or restrictions on the WordPress servers such as blocking outgoing web traffic from unauthorised processes.

[2] See http://mu.wordpress.org for project website, the Seaweed plugin was extended to be compatible with WPMU.

step and survey of both studies. The web server hosting the web site forwarded all of the submitted forms to the research server's centralised database.

5.3 Results and Discussion for the Prescribed Study

This section presents the results for the prescribed study. The results are analysed and discussed in turn.

5.3.1 Modifications to the Seaweed Plugin

Chapter 4 describes the following aspects of the Seaweed plugin that were implemented/changed as a result of the findings from this study:

- The ability to dock/undock and control panel was implemented, as described in Section 4.3.1.
- The ability to send instant feedback was implemented, as described in Section 4.3.1.
- The ability to edit image attributes via the image editor was implemented, as described in Section 4.5.2.
- The control panel was changed from starting in a minimised state, to starting in maximised state instead, as described in Section 4.3.1.

Although the implementations listed above were not present in this study, they were present in the unprescribed study. Enhancing the Seaweed software for the unprescribed study did not negatively effect the final overall conclusions of this research, since the two studies are not compared, but instead are treated as two difference types of studies.

5.3.2 The Participants

A total of nine participants took part and completed the study. All except for one participant had experience with blogging, using a range of blogging

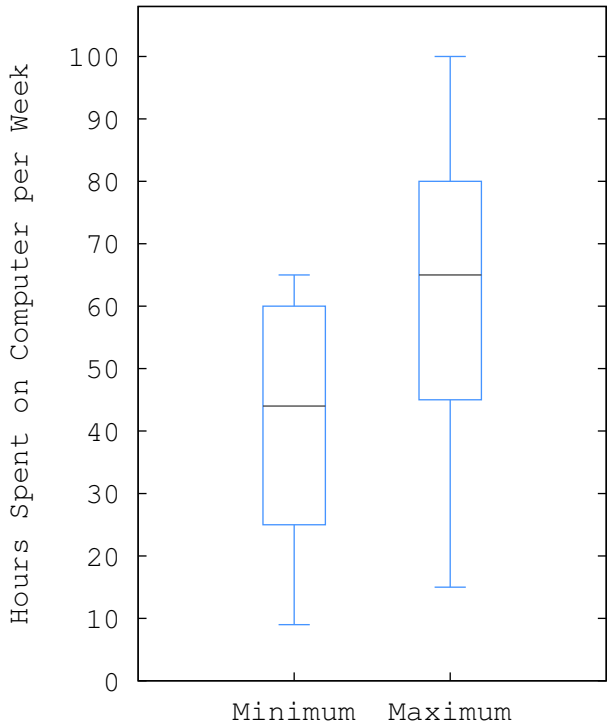


Figure 5.2: Minimum and maximum hours participants spend on a computer per week.

systems other than WordPress for at least one year. Figure 5.2 presents a box and whisker graph for both the minimum and maximum number of hours the participants typically spend on a computer per week. The participants were a mix of IT professionals and non-IT professionals, which is reflected by the variation of hours spent on the computer in the graph.

Figure 5.3 presents a histogram, plotting the number of participants who rated their experience with visual editors in three different contexts using a six-point scale. The scale ranged from zero, representing no experience, up to a maximum of five, representing high levels of experience. The "Office" plots refer to experience using visual editors for office software packages such as Microsoft Word or Google Docs. The "Webpage" plots refer to experience using external/stand-alone visual editors for creating web pages, such as Adobe's Dreamweaver. Finally, the "General" plots refer to experience using visual editors in other contexts other than the other two mentioned, such as WYSIWYG editors used for creating emails, and includes WYSIWYG editors for content

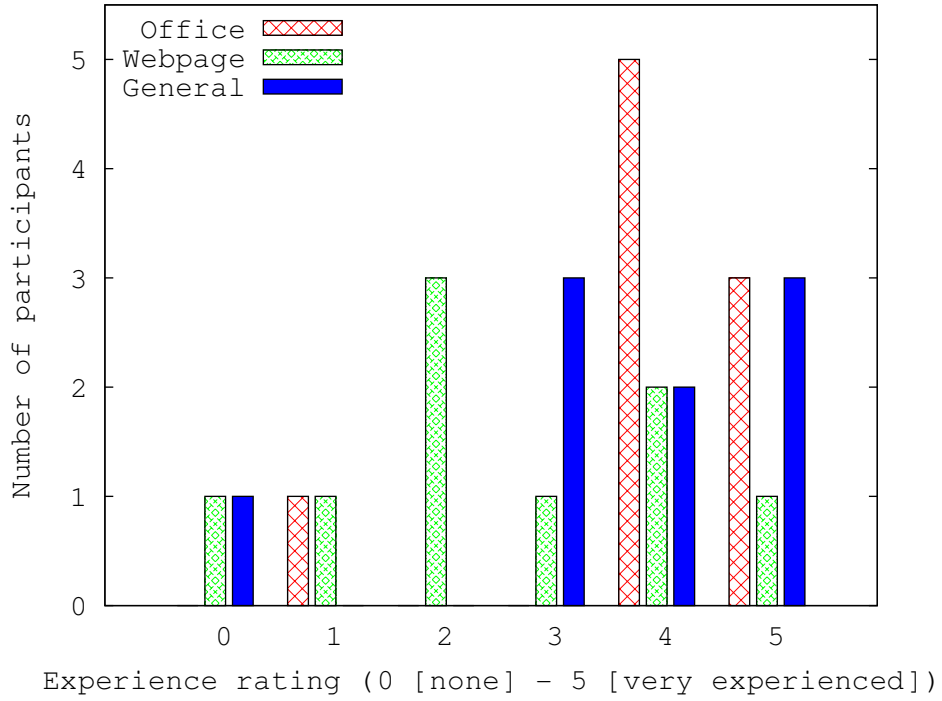


Figure 5.3: Participants' self-rated visual-editor software experiences.

management systems like WordPress. The participants generally considered themselves to have substantial experience with office-suite and general visual editor. Half the participants had limited experience with using stand-alone visual editors for web pages.

5.3.3 The Log Data

Only activity logs for the third part of the study were considered for analysis, since the prior parts consisted of specific tasks given to the participants for learning purposes. Only the activity logs for the third part would provide insightful information, as it was the only part where participants had the freedom to decide whether they wanted to use WordPress or Seaweed for carrying out their editing tasks. A total of 205 activity logs were recorded from eight of the nine participants, since one of them failed to press the button for marking the time when they began the third part of the study. A bug in the logging system resulted in corruption of two activity logs, which constitutes 1% of the total log data.

5.3.4 Usability and Functionality Issues

Appendix F details specific quirks with the Seaweed plugin/framework that were identified by participants. These quirks were considered to be minor issues, and all of them were fixed for the second study. The following sections discuss usability issues experienced by the participants using the Seaweed plugin.

The Control Panel

The control panel (described in Section 4.3.1) would sometimes obstruct participants from editing content, even when minimised. Participants found scrolling the web page to reveal editable content under the control panel to be unsatisfactory. To address this issue, the control panel was improved so that it could be undocked and moved to another position on the web page.

One participant found the initial minimised state of the control panel made it difficult for them to know what actions were possible when first using the software. This was addressed by setting the default configuration of the Seaweed plugin's settings to initially show the control panel maximised.

The Toolbox

One participant noted that it was not immediately obvious how to format the editable content until they made their first edit since the toolbox (described in Section 4.3.2) only would appear on the first edit. The reason for initially hiding the toolbox was to avoid cluttering the web page with too many GUI elements.

The control panel had a button for showing the toolbox, as described in Section 4.3.1. However because the initial state of the control panel was minimised, it would not have been immediately obvious for users on how to show the toolbox without having to make an edit. By adjusting the control panel so that it would initially be displayed in a maximised state (as described in the previous section), the button used for showing the toolbox became visible right away, thus making it more immediately obvious on how to format the

Action	Total Occurrences for Seaweed	Total Occurrences for WordPress	Mean Proportion for Seaweed	Mean Proportion for WordPress
Save	87	74	0.63	0.37
Create New	25	7	0.70	0.30
Delete	0	4	0	1

Table 5.2: Summary of post/page action activity.

editable content without having to make an edit.

High Editing Latency

Two participants found that in some cases they could type faster than the letters would appear on their screen. Furthermore, one of the participants found the selection to be "sluggish," such that the time it would take for the selection's highlighting to appear took too long. These are critical issues, as they violate the direct manipulation principles as discussed in Section 2.2, thus losing the feeling of direct engagement. For the unprescribed study, the Seaweed framework's performance was improved to lower the latency for typing and selection.

No Image Editing Support

Three participants noted that the Seaweed plugin should have provided a way to change the size and alignment of images. These participants said they resorted to using the HTML editor in the back-end view for WordPress for tweaking images. One participant suggested that the Seaweed plugin should have an HTML editing feature. Incorporating an HTML editing feature for the Seaweed plugin would conflict with the research focus of this work, since it would introduce a mode. Thus for the unprescribed study, a GUI for editing images in Seaweed was developed (described in Section 4.5.2).

5.3.5 Overview of Action Activity

Table 5.2 provides a summary of the action activity that was observed for the third part of the study. The table shows the different types of actions that were performed by the participants. The action labelled "Save" refer to the action of saving existing posts/pages, including saving of drafts. The action labelled "Create New" refers to the action of creating new posts/pages. The action labelled "Delete" refers to the action of deleting of post/pages. These three types of actions encompass all the types of actions that occurred during the study. The total number of occurrences are the sums of all occurrences of each action being performed amongst all the participants. The fractional values in the the columns labelled "Mean proportion for Seaweed" are the averages of the proportion of actions that were carried out using Seaweed amongst the participants. The values in the last column are the averages of the proportion of actions that were carried out using WordPress amongst the participants. For example, the proportion of save actions that happened in Seaweed and WordPress were calculated for each participant, and then averaged, giving mean proportions of 0.63 and 0.37 respectively. Analysing the total number of occurrences for each system would not provide meaningful information (as opposed to proportions), since some participants may have performed many more actions than others. Averaging proportions of system usage gives more representative information for summarising system usage over all of the participants.

Discussion

Table 5.2 reveals that of all the actions carried out by participants, saving existing posts/page was clearly the most common type. This was expected, as it is generally common for people to save a single document more than once while constructing drafts, and for tweaking the presentation and correcting published content. On average, the participants used the Seaweed plugin 63% of the time for saving posts/pages, as opposed to using WordPress. This indicates that in general the participants favoured Seaweed for editing/saving

existing content.

According to Table 5.2, participants generally used the Seaweed plugin more than WordPress when creating new posts/pages. With an average proportion of 70% of the participants using Seaweed instead of WordPress for creation new post, clearly they favoured Seaweed in this aspect.

The four delete actions only indicate that deleting posts are an uncommon action. Despite that all of the deletions were carried out via WordPress' administrator, due to the small amount of occurrences there is not enough data to make compelling conclusions.

5.3.6 Editing Activity

The previous section provided an overview of the observed activity at an action type level, where saving was the most common action. A participant may perform many edits before they save, or choose to save at every edit they make. Thus further analysis is necessary to break down the recorded save actions into a finer granularity: calculating the amount of edits made for each save action. This section analyses the different types of edits the participants made on published content during the third part of the study.

Counting Edits

Two types of edits were extracted from the full HTML captured for each save action log: *content edits* and *formatting edits*. Each of these types are described in turn.

Content edits. Content edits are changes on the content or structure of HTML markup. Figure 5.4 shows the pseudo code for calculating these edits. The identifiers *contentBefore* and *contentAfter* refer to the content before and after the save action respectively, where all HTML tags were removed and white-space collapsed (see Section 3.10 for an explanation on collapsing white-space). Images, line breaks, horizontal rules, more tags (refer to Section 4.6.2 for information on more tags) and closing tags for container elements (such as </p>) were replaced with a single character, as they were regarded as

```
diffResults = diff(contentBefore, contentAfter);
edits = []; # List of edit magnitudes

for (hunk in diffResults) {

  if (hunk.isDelete)
    edits.add(hunk.deletedSize);

  else if (hunk.isInsert)
    edits.add(hunk.insertedSize);

  else  if (hunk.isChange)
    edits.add(hunk.deletedSize + hunk.insertedSize);

}
```

Figure 5.4: Pseudo code for counting content edits.

being part of the content/structure. Shortcodes were replaced in the same way (see Section 4.5.3 for information on shortcodes). Shortcode for WordPress captions was handled specially, where the caption text was extracted from the shortcode and included as part of the content. Figure 5.8 shows an example of the content extraction process. The top-most group of content represents the content extracted from the HTML presented in Figure 5.6. The group of content in the middle of Figure 5.8 represents the content extracted from the HTML presented in Figure 5.7. These two examples of content extraction show HTML tags being stripped, and the white-spaces collapsed.

A *diff algorithm* was applied to the extracted content at a character level in order to discover all insertions and deletions made in the save (refer to [26] for a detailed explanation of the diff algorithm). The identifier named "hunk" in Figure 5.4 refers to a block of content that represented one of four possible types of differences: content that had not changed, content that had been inserted, content that had been deleted, or content that had been replaced with new content. A hunk that has no changes is not considered an edit, since nothing has changed. A hunk that represents a replacement of content is comprised of both inserted and deleted content, and is considered as a single edit. Thus all hunks except for hunks that have no changes are considered

```
diffResults = diff(tagsBefore, tagsAfter);
editCount = 0;

for (hunk in diffResults) {

  if (hunk.isDelete) {
    for (tag in hunk.deletedContent) {
      if (tag is an opening HTML formatting tag)
        editCount ++; # Deleted formatting
    }

  } else if (hunk.isInsert) {
    for (tag in hunk.insertedContent) {
      if (tag is an opening HTML formatting tag)
        editCount ++; # Inserted formatting
    }

  } else if (hunk.isChange) {
    for (insTag in hunk.insertedContent) {
       for (delTag in hunk.deletedContent) {
         if (insTag and delTag refer to the same opening HTML tag) {
           # Count attribute/style edits
           if (insTag.name == delTag.name) {
            editCount ++; # Tag's attribute/style changed
            insTag.consumed = delTag.consumed = true;
           }
           # Count changed containers. e.g. changing headings
           else if (insTag.name is a container type) {
            editCount ++; # Tag changed container type
            insTag.consumed = delTag.consumed = true;
           }
         }
       }
    }
    for (insTag in hunk.insertedContent) {
      if (insTag is an opening HTML formatting tag
          AND insTag.consumed == false)
        editCount ++; # Inserted formatting
    }
    for (delTag in hunk.deletedContent) {
      if (delTag is an opening HTML formatting tag
          AND delTag.consumed == false)
        editCount ++; # Deleted formatting
    }
  }
}
```

Figure 5.5: Pseudo code for counting format edits.

```
<h2>My favurite things</h2>\n
\n
<p>\n
  I <em>really love</em> the \n
  color green.\n
</p>\n
<p>\n
  I also enjoy movies \n
  that are comedies.\n
</p>\n
```

Figure 5.6: Example HTML content before changes are made.

```
<h3>My favourite thing</h3>\n
<p>I <strong>absolutely love</strong> the color
<span style=''color:#00FF00''>green</span>.</p>\n
```

Figure 5.7: Example HTML content after changes are made.

as an edit. Figure 5.8 shows the result of diff algorithm being applied to the extracted content before and after changes are made. The figure shows four edit hunks that occur. Text that is not labelled as a type of hunk in the result of the diff algorithm, shown in Figure 5.8, represents hunks that do not have changes.

Each edit has an *edit magnitude*, which is the total amount of characters inserted and/or deleted of the edit's hunk. For example, the edit magnitude of the first occurring edit in Figure 5.8 has an edit magnitude of one, since only one character was inserted in that hunk. The edit magnitude provides information about the types of edits that were being made, and is used while analysing content edits.

Formatting edits. Formatting edits represent changes to the appearance/style of the rendered HTML markup. Figure 5.5 shows the pseudo code used for calculating these edits. The identifiers *tagsBefore* and *tagsAfter* refer to the lists of HTML tags in the content before and after the save action respectively. The tags were ordered in the same order which they occurred in the content. For example, Figure 5.9 shows two lists of tags extracted from

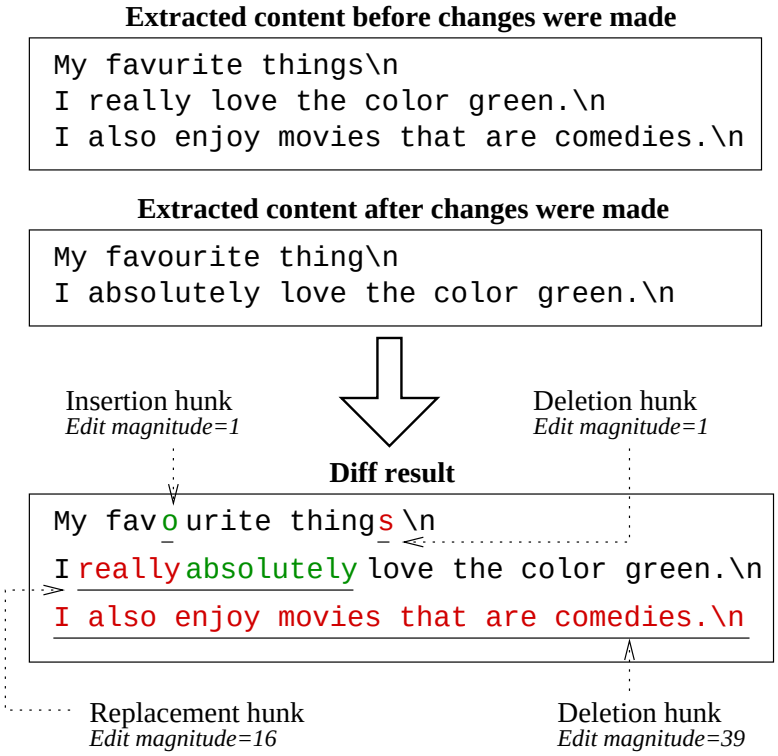


Figure 5.8: Example of calculating content edits.

HTML content: where the HTML tags on top-left in Figure 5.9 were extracted from the HTML presented in Figure 5.6, and the HTML tags on the top-right in Figure 5.9 were extracted from the HTML presented in Figure 5.7. For each opening tag, all attributes and CSS properties were arranged in alphabetical order, since web browsers can order them in a different way to each other. This prevented over-counting the formatting edits for changed attributes/CSS properties in cases where the participants would save content using a different web browser (or editing system) to which the changed content was last saved with.

A diff algorithm was applied to the HTML tags at a tag level in order to discover the various types of formatting edits, as detailed in Figure 5.5. Figure 5.9 presents an example of the formatting edits being calculated from HTML tags extracted from a save action. The example shows that the method outlined in Figure 5.5 calculates four formatting edits from the HTML content before a save (Figure 5.6) and after a save (Figure 5.7). The deleted paragraph tags in the diff results presented in Figure 5.9 were not counted as formatting edits

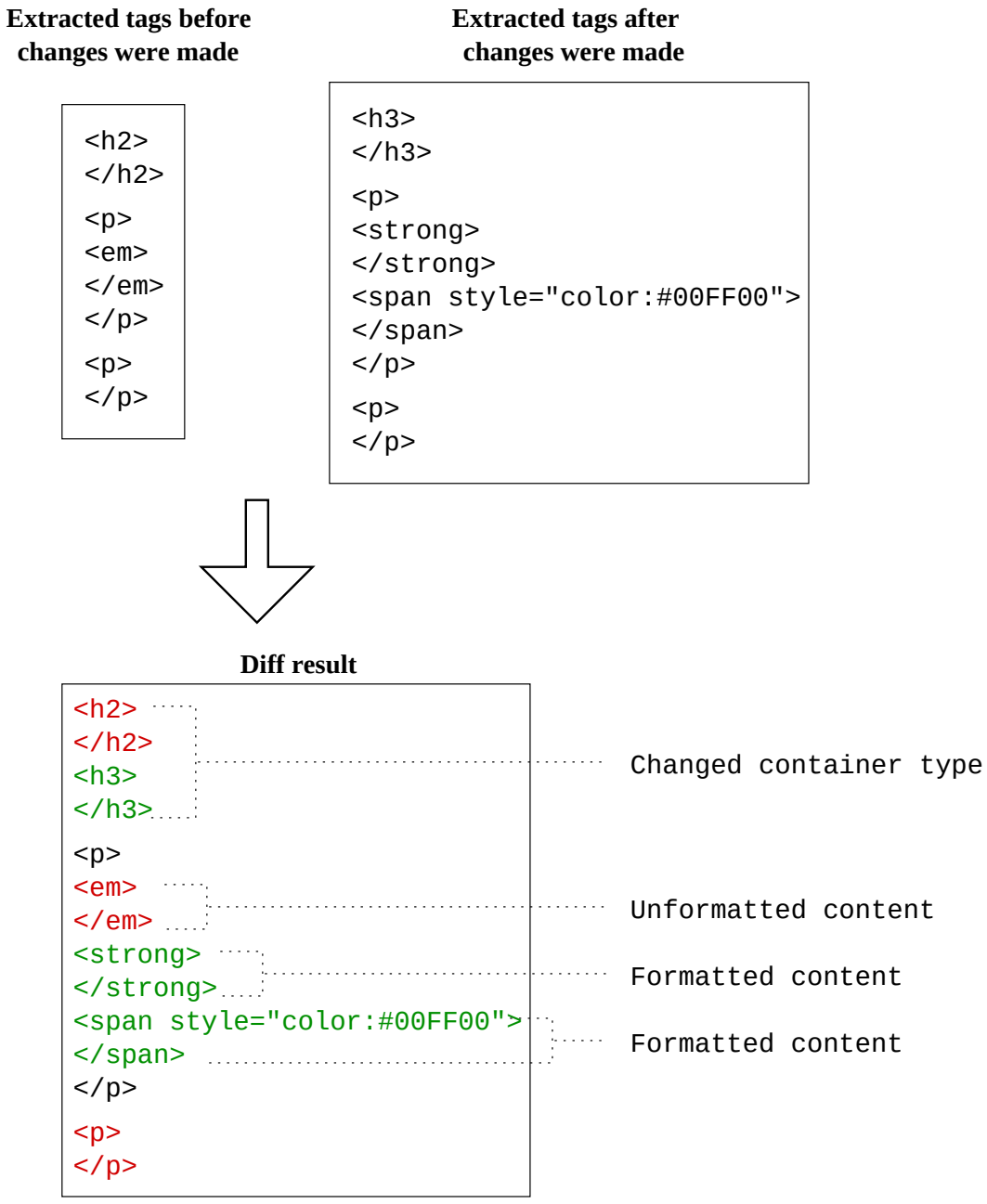


Figure 5.9: Example of calculating format edits.

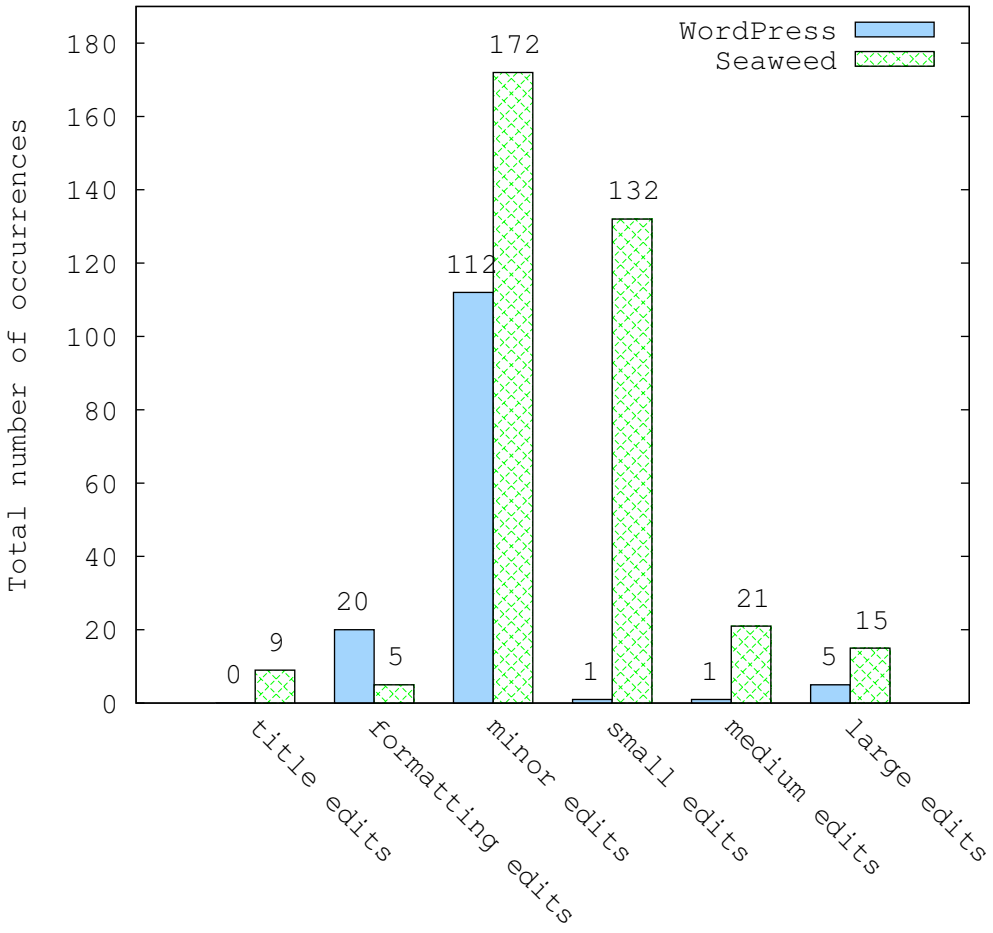


Figure 5.10: Total number of edits on published pages and posts.

since paragraphs are considered as structural content, and thus are counted as content edits instead.

Results

Figure 5.10 shows the total amount of edits that were applied to published content for all participants. Edits on published content were only considered, since edits on drafts and edits on published content are two different editing activities. Edits on drafts represent the initial construction of the content, whereas edits on published content represent improvements such as presentation tweaks or spelling/grammar corrections. Six exclusive classes of edits were defined, where an edit can only belong in one of the following classes:

- *Title edit.* These are content edits on post and page titles.

Edit Class	Edit Magnitude
Minor	1-2
Small	3-5
Medium	6-10
Large	11+

Table 5.3: Classifications of content edit magnitudes.

- *Format edit.* These are formatting edits on post and page body-content.
- *Minor, small, medium and large edits.* These are content edits on post and page body-content. Table 5.3 shows edit magnitude ranges represented by each of these classes.

Some participants edited more content than others, resulting in a large amount of variation in the amounts of edits for each of the participants. To make comparisons on editing activity between the editing systems (Seaweed and WordPress), Figure 5.11 presents the average proportions of editing systems for edits on published content over all participants.

Discussion

Figure 5.10 reveals that minor edits were by far the most common type of edit the participants made. These were most likely spelling corrections, as they represented edit magnitudes of one or two characters. Small edits were also a notably common edit type. Figure 5.11 indicates that participants clearly favoured Seaweed for carrying out minor, small and medium sized edits over using WordPress.

Medium to large edits were less common, as shown in Figure 5.10. It was no surprise for large edits to be uncommon, as one would expect large edits to occur on drafts only. Figure 5.11 shows that generally participants had no preference of an editing system when making large edits.

Editing titles was the least common type of edit observed during the study, as shown in Figure 5.10. Seaweed was clearly the preferred system for editing titles on published content, participants never used WordPress once for editing

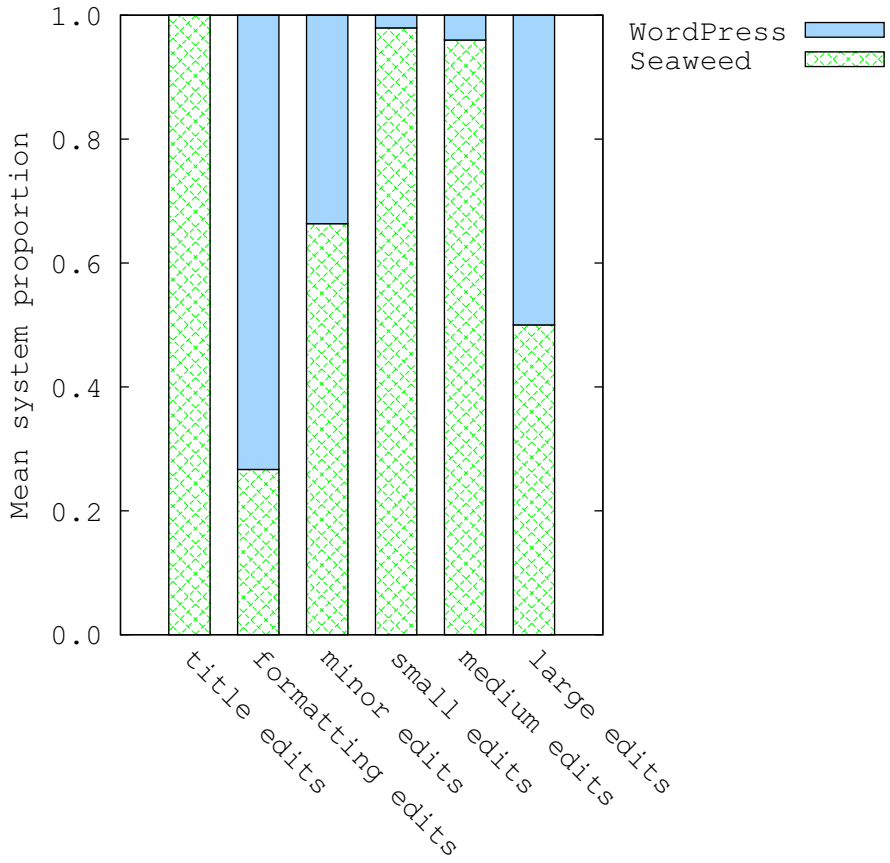


Figure 5.11: Mean proportions of editing systems for edits on published pages and posts.

titles.

Adjusting the formatting for published content was not a common editing activity, as shown in Figure 5.10. The participants generally favoured WordPress for formatting, since Figure 5.11 reveals that on average participants used WordPress for formatting published content 73% of the time. As confirmed in the qualitative analysis, Seaweed's lack of image editing support was a key factor that attributed to the participants preferring WordPress over Seaweed for formatting the content.

5.3.7 Qualitative Feedback

This section discusses qualitative feedback and Likert responses from the online survey which participants filled out after the study. Some of the feedback was also gathered from emails sent to the researcher during the study.

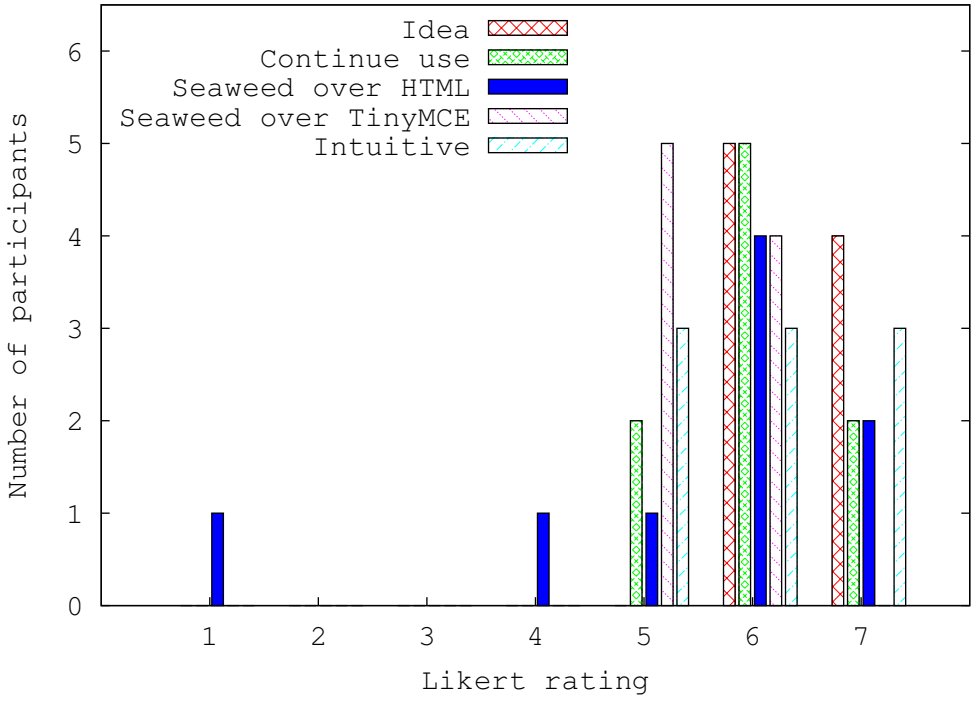


Figure 5.12: Likert responses from survey.

Figure 5.12 presents the Likert ratings for all participants who completed the survey. There were five statements that the participants had to rate on a seven point scale. The higher a Likert rating, the more a participant agreed with the statement being rated. For example, a Likert rating of zero means that a participant had completely disagreed with a statement, whereas a Likert rating of seven means that a participant completely agreed with a statement. The statements are described in turn while discussing the qualitative feedback. See Appendix E for the exact wordings of the statements that was presented in the online survey.

Overall Thoughts on Seamless Editing

The histogram plots labelled "Idea" in Figure 5.12 are the Likert responses from the participants on the concept of seamless editing for the web. The Likert ratings reveal that all of the participants liked the idea. Furthermore, all of the participants expressed positive feedback on the concept of seamless editing when asked to give general comment about the Seaweed software.

The histogram plots labelled "Continue use" in Figure 5.12 are the Likert responses for whether the participants would continue to use Seaweed plugin if they had their own blog. These Likert ratings were high, showing that they would likely continue to use Seaweed.

HTML Editor Comparison

The Likert responses for the preference of using the Seaweed plugin over the HTML editor varied more than the other Likert responses shown in Figure 5.12. Although the majority of the participants preferred the Seaweed plugin over the HTML editor, one participants did not have a preference (four point rating on a seven point scale), and another preferred HTML editing a lot more than the Seaweed plugin (giving a Likert rating of the lowest possible value). These two participants were IT professionals, who stated that they preferred HTML for tweaking images. The participant who gave the lowest Likert rating made the comment that they used the HTML editor since they "needed more control over the code than Seaweed was able to give." Although the lack of image editing support in Seaweed is a clear contributor to participants preferring HTML editing over Seaweed, it seems that some web savvy users prefer HTML editing over visual editors in general.

TinyMCE Editor Comparison

In general the participants preferred the Seaweed plugin over TinyMCE (the visual editor for WordPress), as shown in Figure 5.12. As a WYSIWYG editor, TinyMCE is better designed and more robust than Seaweed. However, despite issues that some participants experienced when using Seaweed (as covered in Section 5.3.4), they considered the ability to directly edit content in the web page to be a more valuable trait.

Intuitiveness

Overall the participants found the Seaweed plugin as being a reasonable to highly intuitive editor, as shown in Figure 5.12. The Seaweed's intuitiveness

could be improved by addressing the initial hidden state problem of the control panel as identified in Section 5.3.4.

One participant stated that they had trouble with identifying the editable parts of the web pages. However as discussed in Section 4.4.2, casual WordPress users would have less trouble since they would be able to identify the parts of their own blog pages, such as they parts that are titles, body content and comments. Thus the activity of identifying editable sections would be more intuitive for active WordPress users.

5.4 The Unprescribed Study Design

This section describes the design of the second study used for evaluation. It begins with outlining the process that the participants had to follow during the study, and is followed by a description on the raw data captured for the study. The infrastructure supporting the study was the same as described for the prescribed study in Section 5.2.3.

5.4.1 The Procedure

The unprescribed study was only open to people who owned a WordPress blog. The participants performed the following procedure for the study:

1. Participants had to download and install the Seaweed plugin for their own WordPress blog.[3]

2. Once the plugin was activated, a model dialog would pop-up containing an online registration form for the participants to register and take part in the study. See Appendix D for the full online registration form.

3. Upon registration, participants would receive a generated unique ID and private key-code via email. They would then enter the ID and key-code in the model dialog containing the registration form (the dialog also contained a small form for *initiating* the plugin).

[3]The plugin was hosted on the official plugin web site for WordPress.

4. The participants would continue to blog for two weeks with the plugin installed.

5. After two weeks the participants filled out an online survey about their experiences during the study. The survey would automatically pop-up after two weeks had passed. Alternatively, participants could complete via the Seaweed web site. See Appendix E for the full online survey.

The first two steps outlined above could be carried out in any order, since participants could sign up directly via the Seaweed web site before installing the plugin.

5.4.2 Data Captured During the Study

The data captured during the study was the same as for the prescribed study (as covered in Section 5.2.2), except for three additions which are described in turn.

A decision was made to integrate an instant feedback feature into the GUI, as described in Section 4.3.1. In the prescribed study participants had to either try and remember their experiences in the study when giving feedback in the survey, or email the researcher. The feedback feature simplified this process, maximising the potential for gathering qualitative responses from participants.

Although participants in the prescribed study could edit post comments (and the plugin was geared for logging comment editing activity), the study did not include editing of comments as part of the tasks. However the unprescribed study was open for capturing editing activity on post comments.

Participants could install the plugin on blogs that had multiple users. Each user in these blogs had to individually sign up to the study if they wanted to take part. The data captured for the study was not grouped together for each blog, but for each blog user. In the first study this was not a concern since participants were assigned their own blogs with only a single user. For users who did not want to take part, users were able to disable the plugin without deactivating it (which would cause the plugin to stop working for other users) for their personal WordPress account.

Privacy issues were more of a concern in the unprescribed study since participants were naturally being observed as opposed to following tasks as they did for the prescribed study. The same precautions for protecting the participants' identities in the prescribed study sufficed for the unprescribed study. Since the registration for signing up to the unprescribed study was open to the public, the online registration forms were spam protected by using Recaptcha.[4]

5.5 Results and Discussion for the Unprescribed Study

This section presents the results for the unprescribed study. The results are analysed and discussed in turn.

5.5.1 Caveat: Editing System Comparisons

The analyses for this study compares two editing systems: Seaweed and WordPress. However, the comparisons are more correctly defined as comparing Seaweed over a moded editor. This is because of the possibility that the participants used other editors (or enhanced versions of WordPress editors) for editing content, provided by other plugins. It is not possible for any other plugin to provide seamless editing while the Seaweed plugin is installed (and as far as we know, Seaweed was the only plugin available to date that provides seamless editing, and if one did exist it would conflict with Seaweed). Thus the activity logs that were specified as being carried out via WordPress would have been carried out via a moded editor.

5.5.2 The Participants

A total of 26 participants installed the Seaweed plugin on their own blog, and registered to take part in the study. Only five of those participants completed the full study procedure, that is, they took part in the two week observation

[4]See http://recaptcha.net for project web site.

period and completed the survey. Three other participants completed the survey early, and ended their observation period before two weeks had passed.

Although the majority of the participants did not complete the full procedure, many of them took part long enough to include their activity logs as part of the analyses. Only participants who used the plugin for at least 10 times or for at least two days were considered in the quantitative analyses, amounting to a final total of 19 participants. Participants who did not submit enough data were excluded from quantitative analyses because their experiences with the plugin would have likely been for trying out the Seaweed plugin — their observational data would not have been representative of their natural blogging experiences.

A total of ten participants were included in the qualitative analysis, as they had supplied feedback from completing surveys and/or the instant feedback feature integrated in the Seaweed plugin's control panel (described in Section 4.3.1). All participants included in the qualitative analysis were included in the quantitative analysis, except for two of them, who provided instant feedback and then immediately aborted the study.

If the participants who had registered to take part in the study and then immediately aborted the study, instead completed the full study, the impact of such a change on the results is unknown. This is because the reasons to why participants immediately aborted the study were not supplied. A reason to why they aborted early, that would change the results if they had otherwise stayed, would be because they did not like Seaweed at all, even if it ran without any issues/limitations. However it might have been that they aborted the study because the plugin did not work at all (or very well) with their WordPress version or theme. Another reason may be that they may never had intended to complete the full study, only wanting to try out the Seaweed plugin for satisfying their curiosity.

Figure 5.13 displays a box and whisker plot of the ranges of hours that the participants typically spend per week on the computer. Only hours for participants considered in the quantitative or qualitative analysis are shown. In comparison to the first study, the observed population generally spends

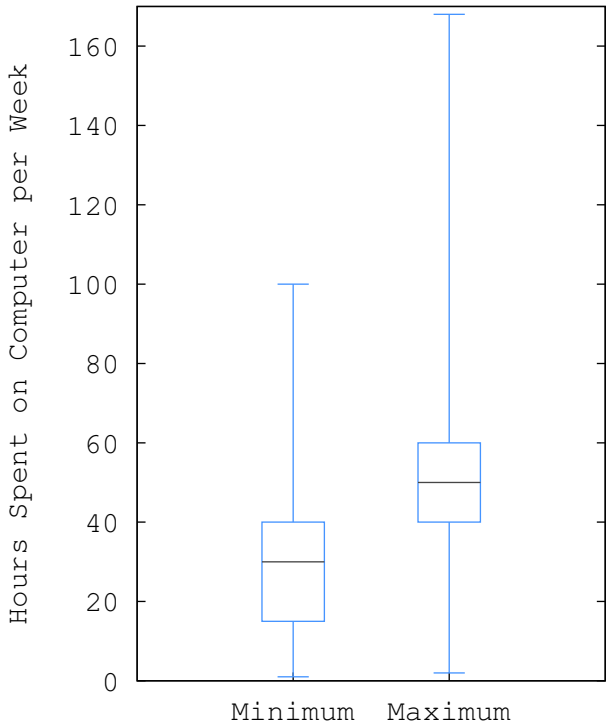


Figure 5.13: Minimum and maximum hours participants spend on a computer per week.

less time on the computer. The graph shows that there was a large variation between participants in the amount of time they spent on the computer.

Figure 5.14 presents the experience ratings of various visual editing software for participants included for analysis (that is, all participants who are included in the quantitative and/or qualitative analysis). The scale of ratings and types of experiences are identical to the graph in Figure 5.3, as discussed in Section 5.3.2. Although the participants have their own blogs on the web, a reasonable portion of them had limited experience with stand-alone visual editors for web pages. Most of the participants had a high level of experience with WYSIWYG editors in general. Four participants specified that they had none, which meant either they miss-understood the question being asked since the question stated that visual editors for blogs were included, or they only used HTML editors for managing their content.

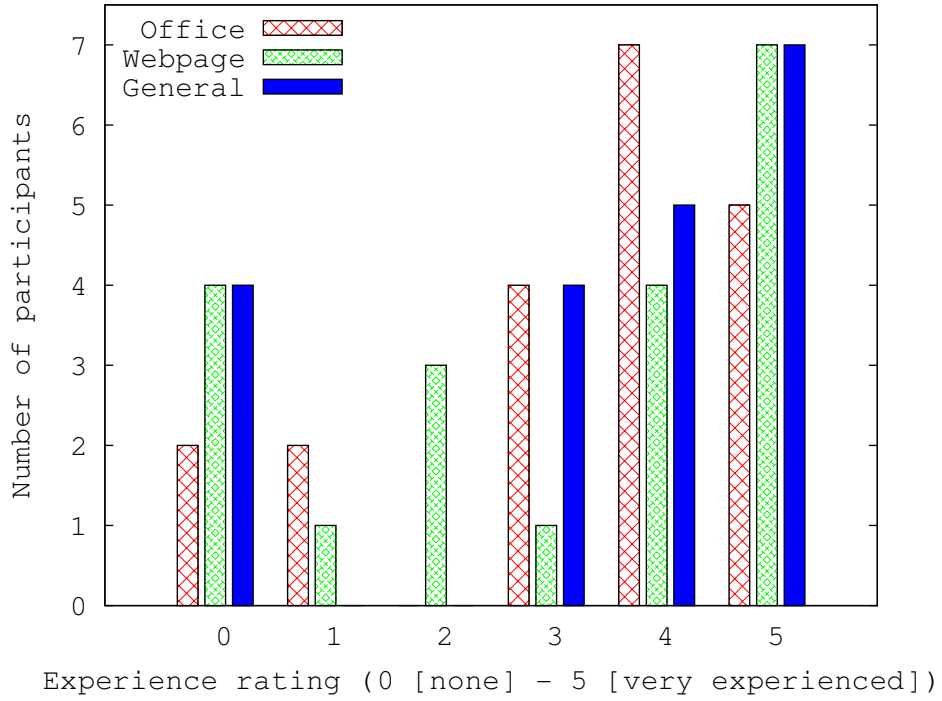


Figure 5.14: Participants' self-rated visual-editor software experiences.

5.5.3 The Log Data

A total of 1009 logs were captured for the 19 participants selected for quantitative analysis of this study. The corruption issue identified in the prescribed study (in Section 5.3.3) — where some logs were found to be corrupt — was unable to be fixed. The issue was unable to be reproduced, and was assumed to be from malformed packets from network failure, rather than a bug in the logging code. Furthermore because only 1% of the logs were corrupt, the issue was considered to be insignificant. Unfortunately the issue did turn out to be a bug, where the full HTML content used for analysing editing frequencies for 34.5% of logs were unable to be recovered. This only effected the analysis presented in Section 5.5.6 as it was the only analysis that analysed the full HTML content. Section 5.5.6 discusses the impact of the corruption and how it was handled when presenting the results.

5.5.4 Usability and Functionality Issues

A new set of quirks were identified by participants during the study which are outlined in Appendix F. These quirks went undiscovered from the first study as the first study used the same WordPress configurations for every participant, where as the second study had a diverse range of WordPress configurations. The participants in the second study identified a different set of usability issues and limitations of the Seaweed plugin not only because of the more diverse set of WordPress configurations, but also because of their expertise with the software. The following sections discuss the usability issues and limitations found for the Seaweed plugin during the study. The issues experienced by the participants ultimately effected their decisions to use Seaweed or WordPress when editing content.

Poor Layout for New Posts and Pages

Two participants found that the layout of web pages for creating new pages and posts was completely disorganised. For example, a participant stated: "the page that loads for entering my content does not at all follow my theme's layout. My copyright, which is the last thing on my page, ends up in the header area on top of my logo." Although the new post/page layout for some themes were poorly presented, they could still create new pages — however their experiences would most likely not have been as satisfying as using WordPress for creating new content. This issue was anticipated, as noted while discussing the approach for creating new pages and posts via the Seaweed plugin in Section 4.7, however the severity of the poor layout for some themes as mentioned by the participants was not expected.

No Support for Moving Cursor Between Words

Two participants noted that when editing content, they usually press the `CTRL` key while pressing the arrows keys on the keyboard to move the text-editing cursor between words; as opposed to just moving one character after another. This feature was not implemented in the Seaweed framework.

Control Panel and Toolbox

One participant found that the control panel and toolbox (described in Section 4.3.1 and Section 4.3.2 respectively) took up too much screen space, as they had a smaller screen resolution than the Seaweed plugin's GUI was designed for. The participant noted that condensing the large buttons for the control panel into small icons (with the use of tool-tips) would be more desirable. This however has a trade-off, although it would be a better design for expert users, it would be difficult for new users to know the actions that are available.

Two participants noted that when viewing their blogs without the intention of editing, the minimised control panel was "irritating" as it was still to large due to their low screen resolution. One participant suggested to make the minimised state of the panel smaller, and provide the ability to choose which side of the screen to dock the control panel.

Comments

One participant noted that the Seaweed plugin should support the deletion and moderation of comments, in order to complete the full set of editing commands available for comments in WordPress. These features were considered when developing the plugin, but due to limited time and its assumed low importance for the prototype, the features were disregarded.

5.5.5 Overview of Action Activity

Table 5.4 provides an overview of the observed action activity from the 19 participants considered for quantitative analysis in the study. The statistical summary presents the total amount of occurrences of actions, and the average proportions of the editing systems they were carried out with. The proportional values in the tables are computed in the same way as for Table 5.2, as described in Section 5.3.5. Unlike the prescribed study, this study observed actions where participants had changed comment content, as indicated in the last row of Table 5.4.

Action	Total Occurrences for Seaweed	Total Occurrences for WordPress	Mean Proportion for Seaweed	Mean Proportion for WordPress
Save Post/Page	239	499	0.43	0.47
New Post/Pages	37	27	0.45	0.55
Delete Post/Page	1	19	0.04	0.96
Save Comment	1	6	0.08	0.92

Table 5.4: Summary of action activity.

Discussion

As expected, save actions were significantly more common than other actions, as shown in Table 5.4. The mean proportions between editing systems for saves actions were reasonably even. Section 5.5.6 investigates these actions in finer detail.

Generally participants did not have a preference of system when creating new posts/pages. It seems that the layout issues for new posts in Seaweed (identified in Section 5.5.4) either was not present for at least approximately half of the participants, or many of the participants still preferred to create posts in Seaweed over WordPress despite experiencing layout issues.

Table 5.4 indicates that on average, the participants used WordPress a lot more than the Seaweed plugin when deleting posts/pages and editing comments. It appears that deleting posts from an external administration view is more desirable than deleting posts directly from a published view. Furthermore, despite having seamlessly editable comments in a blog, participants did not edit other users' comments very often.

Participants generally used WordPress over Seaweed when editing comments. This would have been likely because of the Seaweed plugin's lack of moderation features, as pointed out in Section 5.5.4.

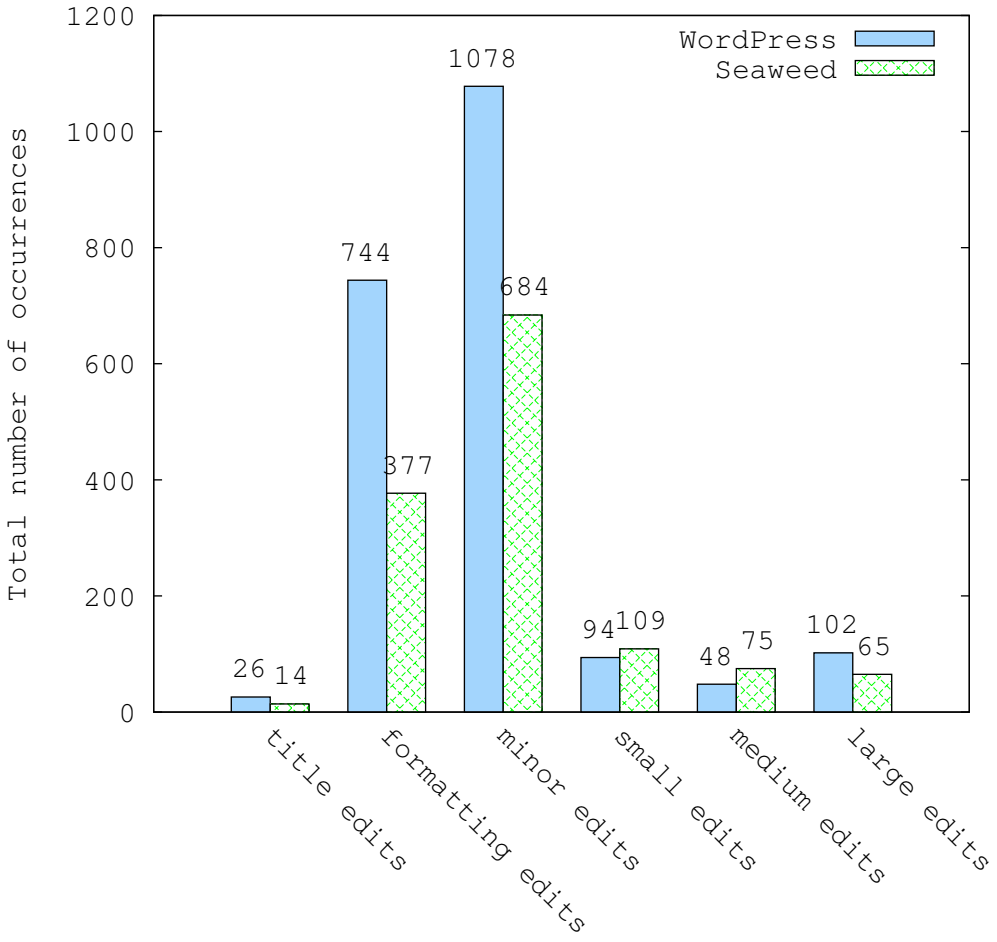


Figure 5.15: Total number of edits on published pages and posts.

5.5.6 Editing Activity

This section analyses the editing activity of the participants that was observed during the study. The same methods explained in Section 5.3.6 were used for classifying and computing the amount of edits made in published content. Figure 5.15 presents the total amount of edits for each editing system (WordPress and Seaweed). Figure 5.16 presents the average proportions of the editing systems that the participants used when making the edits.

As previously pointed out, the full HTML content in 34.5% of the editing action logs were corrupt. Because the nature of corruption was not purely random, as the cause of the corruption was from a bug, simply ignoring the corruption when analysing full content would produce biased results. Fortunately a reliable method for detecting whether an action log was corrupt was

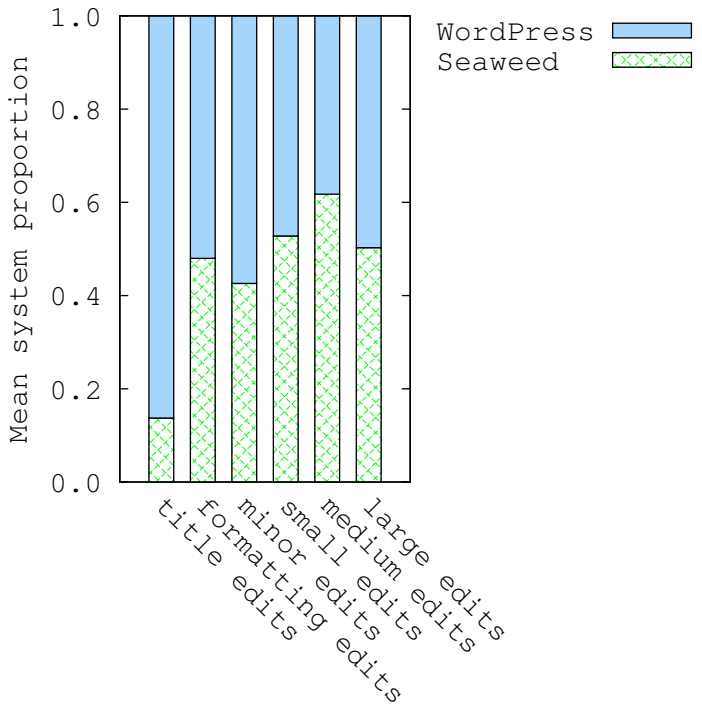


Figure 5.16: Mean proportions of editing systems for edits on published pages and posts.

devised. A log was detected as being corrupt if its checksum (see Table 5.1) did not validate. Corruption was not present amongst all of the participants. For participants who did have corrupt logs, the portion of corrupt logs was between 11.8%, and 98.6%, with a median of 43.4%. These participants were dropped from this analysis — that is, participants who had at least one corrupt log — because the percentages of corruption were too significant. Removing participants with corrupt logs in this study left a total of eight participants who were considered in this full-content analysis.

Discussion

Figure 5.15 reveals that most of the observed content edits were only one or two characters in length. Figure 5.16 reveals that, in general, the participants did not prefer either WordPress or Seaweed when carrying out all types of edits, except for when editing post titles.

The participants in this study were real bloggers, using a plugin labelled as a "prototype used for research purposes," in their own blog. Some people

would consider their blogging activities to be more than just a hobby. For example, some people use blogs as a way of marketing themselves to the world, or generating revenue of visitor traffic. Thus, the participants would have been concerned about the quality of the content. Because of the experimental nature of the plugin, and the discoveries of quirks (as previously discussed), the participants may have been apprehensive of using the Seaweed plugin as their choice of editor during the study. If the participants would not have been worried about using their own blogs, they may have used the Seaweed plugin generally more than WordPress.

Participants who edited titles of published content, generally preferred to use WordPress over Seaweed. This may have been because the WordPress editor also gives options for editing *permalinks*, an operation that Seaweed does not provide. A permalink is a static URL used for referencing to blog entries or new articles on the web. People tend to name these links to match the title of the blog, thus participants may have preferred WordPress because it supplied permalink editing, rather than because of a preference of moded way of editing titles over a seamless way.

5.5.7 Usage Over Time

Figure 5.17 presents the mean proportion of actions that were carried out via Seaweed per day over the whole study. Only participants who completed the full study were analysed. For each day, the proportions of actions which were carried out by seaweed were computed for each participant, then averaged giving the final proportion as shown in the graph. For days where a participant did not have any actions, the proportion was taken from the linear equation of the line between the proceeding and succeeding days when they last/next performed an action.

Discussion

Figure 5.17 shows that initially the participants began using the Seaweed plugin generally more often than WordPress. However over time they gradually

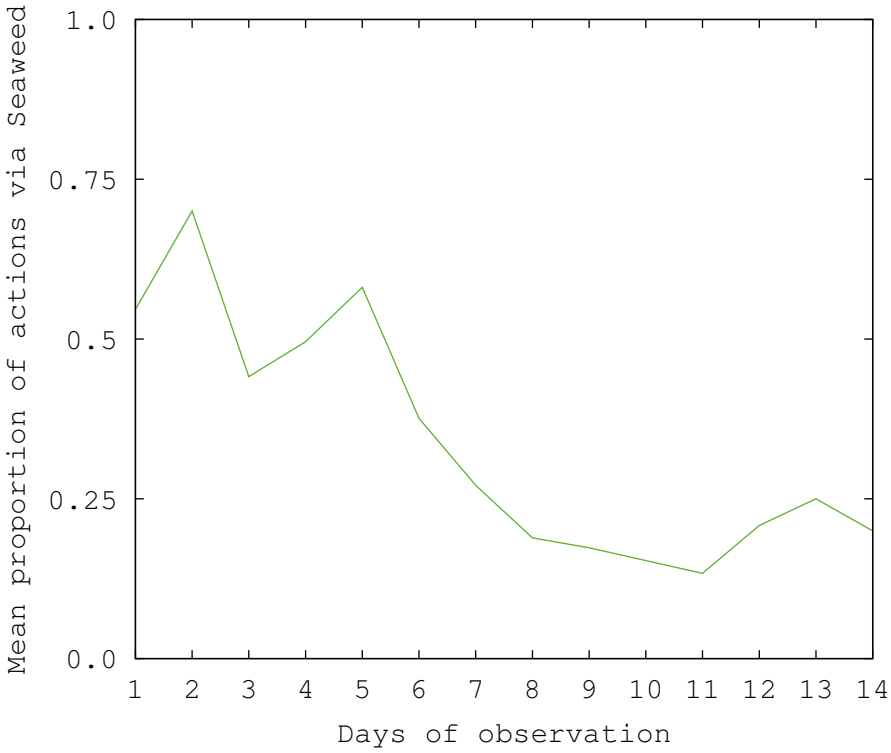


Figure 5.17: Mean proportions of all types of actions executed via Seaweed over time.

tended to use WordPress over Seaweed for managing/editing content. The initial peaks of Seaweed's usages early in the study may be attributed to the fact that the participants were experimenting with the plugin. It might have been that the participant lost confidence with using Seaweed as time went on, due to the discoveries of limitations and/or quirks of the prototype over time.

5.5.8 Qualitative Feedback

This section discusses qualitative feedback and Likert responses from the online survey which a total of eight participants filled out after the study. 11 general comments sent by participants via the instant feedback feature were also included as part of this analysis.

Figure 5.18 presents the Likert responses from the survey (see Appendix E), in the same format as Figure 5.12 as discussed in Section 5.3.7.

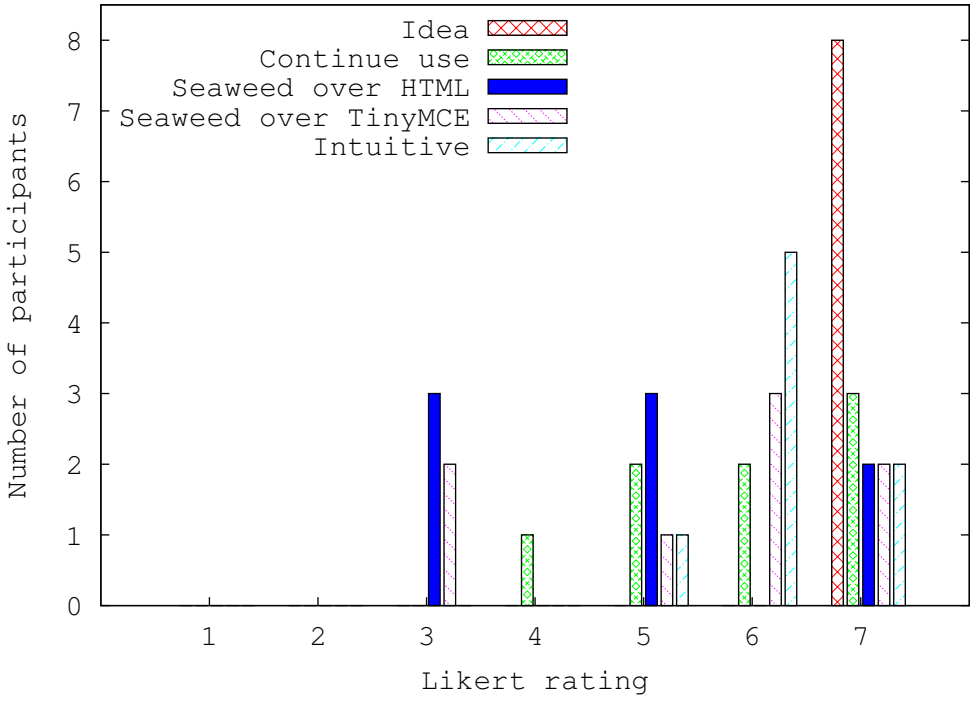


Figure 5.18: Likert responses from survey.

Overall Thoughts on Seamless Editing

Figure 5.18 shows that all participants who completed the survey gave the highest Likert rating for the concept of seamless editing. Participants who submitted general comments about the concept were all highly positive. One participant noted: "The editing-in-place interaction is something I've wanted forever, [I have] only seen [this type of interaction] in bits and pieces [on the web], and would love to see it become more common." Clearly the concept of seamless editing is an idea worth pursuing.

Generally the participants would continue to use the Seaweed plugin in their own blogs. One participant noted: "I am an editing minded person and find myself proof reading posts over and over again to present my best work. This plugin offers a whole new view and saves me a ton of time." The participants were aware that the plugin was just an experimental prototype, thus these ratings do not necessary mean that they interpreted the question as continuing to use the prototype, but improved versions instead. This indicates that seamless editing in blogs is considered to be useful.

HTML Editor Comparison

Overall the participants indicated that they preferred the Seaweed plugin over the HTML editor in WordPress, as reflected by the Likert responses in Figure 5.18 labelled as "Seaweed over HTML." Although three of them were undecided.

One participant who did not complete the survey, noted that the Seaweed plugin lacked the ability to add/edit JavaScript within posts. This participant quickly aborted the study, most likely based on this limitation. Editing JavaScript is not considered to be a common activity in WordPress — since in WordPress, content enhanced with JavaScript within posts are usually generated by plugins via shortcodes. As noted in the previous study, incorporating an HTML editing mode in Seaweed would defy the direct manipulation principles which seamless editing was built upon.

TinyMCE Editor Comparison

Generally the participants indicated that they preferred the Seaweed plugin over the TinyMCE editor in WordPress, as reflected by the Likert responses in Figure 5.18 labelled as "Seaweed over TinyMCE." Only two of them were undecided on which editor they preferred.

Intuitiveness

Overall the participants found the Seaweed editor to be highly intuitive, as indicated by the high Likert ratings in Figure 5.18 (labelled as "Intuitive").

5.6 Summary and Conclusions

This chapter analysed observation data and feedback from two different studies comparing seamless editing using Seaweed with moded editing using WordPress' administration facilities. The first study involved participants with little or no expertise with the WordPress system, following a set of tasks using temporarily assigned blogs. The second study involved participants using their

personal WordPress blogs as the environment for testing seamless editing. The outcomes of the two studies offered two different perspectives on the concept of seamless editing.

5.6.1 Findings for Research Questions

Here we summarise the key findings to the first five research questions formulated for the user studies. Given the broader nature of the final question — what are other contexts where people who access the web could see seamless editing being helpful — we defer this to the concluding chapter of the thesis.

Seaweed Compared with the External Visual Editor

The first research question established for the studies was as follows: what are the situations in which people prefer using Seaweed over using an external WYSIWYG editor? And what are the situations in which they do not?

Both the Likert responses (refer to Section 5.3.7) and the activity logs (refer to Section 5.3.5) of the prescribed study indicated that people without expertise using WordPress preferred editing content with the Seaweed plugin. The Likert responses in the unprescribed study (refer to Section 5.5.8) indicated that people with expertise using WordPress preferred editing content with the Seaweed plugin.

The full content analysis of the activity logs in the prescribed study (refer to Section 5.3.6) revealed that participants clearly preferred Seaweed for making minor to medium sized edits. The analysis on the activity logs for the unprescribed study did not provide compelling information for identifying the types of situations where Seaweed was preferred over the WYSIWYG editor used by WordPress for expert users of WordPress. This was because of the quirks and limitations of the Seaweed plugin: the participants probably did not want to compromise the quality of their blogs as they progressively discovered issues with the Seaweed plugin over time.

Seaweed Compared with the HTML Editor

The second research question established for the studies was as follows: what are the situations in which people prefer using Seaweed over writing raw HTML markup? And what are the situations in which they do not?

According to the Likert responses in both of the studies, the participants generally preferred seamless editing over using HTML syntax (refer to Section 5.3.7 and Section 5.5.8). However it was found that there are users who always prefer editing HTML source over using visual editors in general. This was indicated by the Likert ratings for both of the studies, and a participant's remark in the prescribed study noting that some people like to have absolute control over the HTML markup themselves. Including an HTML editor in the Seaweed plugin was disregarded because it was designed for research purposes that focused on seamless editing. Including an HTML editing mode might have made it difficult for users to get a real taste of seamless editing. However, for a usable version of the Seaweed plugin, in order to accommodate users who prefer HTML editing in general, it would be acceptable to include an HTML editing feature. The plugin would still be modeless, the only difference would be that it would provide a moded feature that is additional, not essential, to the editing process.

Seaweed's Intuitiveness

The third research question established for the studies was as follows: how intuitive is Seaweed?

In both of the studies participants had to teach themselves how to use the Seaweed plugin, thus participants were able to give authoritative feedback on the plugin's intuitiveness. Overall, the Likert responses (in Section 5.3.7 and Section 5.5.8) clearly indicated that the Seaweed plugin was highly intuitive.

Adapting to Seaweed's Editing Process

The fourth research question established for the studies was as follows: how well people who have substantial experience with the traditional way of editing

in WordPress adapt to seamless editing?

Generally, participants used the Seaweed plugin in the unprescribed more than WordPress at the beginning of their observation period (see Section 5.5.7). The initial high usage activity indicates that users quickly adapted to Seaweed. Therefore, people who are accustomed to a moded way of editing can easily adapt to seamless editing.

Seamless Editing Concept Reception

The fifth research question established for the studies was as follows: do people who access the web, like the concept of seamless editing?'

All participants in both of the studies gave positive feedback on the concept of seamless editing for content on the web in general (see Section 5.3.7 and Section 5.5.8). All participants who completed the online surveys rated the seamless editing concept with high Likert ratings. All ratings in the unprescribed study were rated the highest possible value on the Likert scale. The qualitative analysis in the unprescribed study indicated that the expert users were wanting to continue with using the Seaweed plugin (it was known to them that the plugin would eventually progress into a free non-prototype software). Therefore, seamless editing is a valuable avenue of work for pursuing.

5.6.2 Additional Findings

The findings of the studies also brought to light insights related to Seaweed and the seamless editing concept. These insights are discussed in turn.

Creating New Content In-Context

When a user wants to create a new post in Seaweed, they must download a new web page that attempts to present an empty post, as if it were already being viewed in a published mode. Creating new content in a published view does not require the users to switch between preview and edit modes, thus supporting a seamless way of creating new content. However, this approach for creating new posts is not the ideal seamless approach. For example, rather than using

URL redirects, it would have been ideal if empty templates for new posts were inserted directly in the web page that the users want to create a new post from. Due to the extreme diversity of themes, this was not possible. Despite having to use a URL redirect, and some participants experiencing layout issues with their themes, the creation of new posts was carried out by Seaweed as much as for WordPress during the observation period for both of the studies (refer to Section 5.3.5 and Section 5.5.5). This suggests that creating new content, in the context which it would be viewed in once published, is a useful feature.

Seaweed's Approach to Content Management

In Seaweed, users had to navigate to the posts that they wanted to delete, within in the published view. The administration view in WordPress provides an overview of all posts within a table, where users can directly delete specific posts from. According to the quantitative analysis for both studies (in Section 5.3.7 and Section 5.5.8), participants clearly preferred deleting content in WordPress.

The Seaweed plugin's approach to deleting posts, an administrative activity, followed the same direct manipulation principles as seamless editing: where users could delete a post directly while they are viewing it. However, it seems that users prefer to carry administrative tasks from an overview perspective of posts rather than a first person perspective.

Common Editing Activities for Bloggers

The unprescribed studies revealed that in real blogging situations, it is common for bloggers to edit content after they have published it to the world (see Section 5.5.5). Most of these edits were attributed to minor edits, most likely being spelling or grammar corrections (see Section 5.5.6). The results of the prescribed study also showed that minor/small edits were the most common editing activity on published content (see Section 5.3.6).

Seaweed has a clear advantage over the other editors with making edits on published content, since users can instantly make corrections without switching to another mode. The results of the prescribed study supports this, as

the editing activity for minor/small edits were predominately carried out by Seaweed.

5.6.3 Summary

In summary the study results indicate that the concept of seamless editing for the web, as exemplified by the Seaweed WordPress plugin prototype, holds great promise. Although the Seaweed software is a research prototype rather than a production system, it is both effective and usable, and its potential has clearly been recognised by the participants in both of the studies.

Although a CMS has served as a useful medium for the purposes of evaluation, seamless editing has a broader range of applications. In the concluding chapter we consider further contexts in which seamless editing could be effective.

Chapter 6

Conclusions

This thesis has established the utility of seamless editing, a new paradigm for authoring content on the web. A framework called Seaweed was developed to provide seamlessly editable environments for all common web browsers, and for any type of web page. The framework was integrated into a CMS called WordPress by developing a plugin. Two different types of observational user studies were carried out, using the WordPress plugin, to investigate seamless editing. One study, the prescribed study, observed participants with minimal to no experience with the WordPress system, using temporarily assigned blogs. The other study, the unprescribed study, observed participants using the Seaweed plugin on their own existing blogs. Both studies observed participants in their own time, and in their natural environment, in which they typically access the web. This chapter begins with a summary of the findings from the user studies conducted for exploring seamless editing. Section 6.2 discusses how seamless editing can be applied to contexts other than blogs. Section 6.3 suggests future work. Lastly, Section 6.4 presents the final conclusion.

6.1 Summary of Findings

The user studies sought to answer six research questions related to the Seaweed plugin and seamless editing. The findings for the first five research questions are summarised in turn, along with additional insights.

- The first research question was: what are the situations in which people

prefer using Seaweed over using an external WYSIWYG editor? And what are the situations in which they do not? People generally prefer the Seaweed plugin over the WYSIWYG editor for WordPress. The Seaweed plugin provided the same set of content editing functionality as the WordPress WYSIWYG editor. According to qualitative feedback and Likert responses gathered from surveys, participants generally agreed that Seaweed's modeless way of editing is superior to a moded way of editing.

- The second research question was: what are the situations in which people prefer using Seaweed over writing raw HTML markup? And what are the situations in which they do not? People generally prefer the Seaweed plugin over the HTML editor for WordPress. However, a minority of people do prefer HTML editing over visual editors in general, as they like to have complete control over the markup. For example, the Seaweed plugin did not provide advanced content editing features, such as embedding JavaScripts within content, for which some participants used in their blogs.

- The third research question was: how intuitive is Seaweed? According to the Likert responses in the surveys, participants in both studies found the Seaweed plugin to be highly intuitive.

- The forth research question was: how well people who have substantial experience with the traditional way of editing in WordPress adapt to seamless editing? Participants accustomed to a moded way of editing, who used the plugin on their own blogs, quickly adapted to seamless editing. When asked whether they would continue using the plugin, the overall response clearly suggested that they would.

- The fifth research question was: do people who access the web, like the concept of seamless editing? All feedback on the concept of seamless editing given from participants in both studies was positive, as all participants gave high Likert responses when rating the idea of seamless

editing. Furthermore, qualitative responses given in emails, surveys and via an instant feedback feature integrated in the Seaweed plugin, were all positive with regards to the concept of seamless editing.

- Creating new posts in the context in which the new post would be viewed when published, was seen as a valuable feature. Creating a new draft as if it was already published, eliminates the need to switch to a "preview" mode for tweaking the presentational aspects.

- Deleting posts from the published view was not a useful approach for managing content. Quantitative analysis on activity data for both studies clearly showed that the participants preferred the administration view for WordPress when carrying out deletion operations on posts.

- Minor edits were the most common type of edit on published content, significantly more than larger sized content edits and presentational edits. A minor edit is a change on content up to two characters in length — most likely attributed spelling corrections — as well as structural type edits (edits that add or remove new lines due to new content structure). Overall, participants preferred Seaweed for making minor edits.

6.2 Seamless Editing the Web

This thesis explored seamless editing using the WordPress CMS. However, the concept of seamless editing can be applied to any situation where there exists a way to edit a web page. The Seaweed framework was designed to work with any type of web page. The approaches explained in Chapter 4 for integrating the Seaweed framework into WordPress, can be used as a guide for converting moded content management systems into seamless environments.

The last research question posed for evaluating seamless editing was: what are other contexts where people who access the web, could see seamless editing being helpful (other than blogs)? To address this question, the participants in both of the user studies were asked to suggest other contexts in which they may find seamless editing useful (refer to Appendix E for exact phrasing of this

question in the survey). Table 6.1 summarises the ideas suggested by participants. The left column identifies the contexts that the participants suggested. The middle column presents examples given by them (where applicable). The right column is the amount of participants who suggested the context.

Context	**Given Examples**	**Amount of Participants**
Any type of web site	*Not Applicable*	4
Wikis	Wikipedia	3
Simple/static web sites	University home pages, stand-alone XHTML web pages	3
Online digital libraries	Greenstone	2
Forums	vBulletin	2
Auction web sites	TradeMe	1
Public video sites	YouTube	1
News web sites	*None*	1
Annotation Systems	*None*	1

Table 6.1: Contexts in which seamless editing may be useful, suggested by participants.

The diverse range of contexts identified by the participants suggests that they saw seamless editing as a fundamental concept, and is applicable to many situations. The first row in Table 6.1, where four participants suggested that it could be applied in any web page, supports this further. The suggestions of other contexts from participants proposes directions for future endeavours into the seamless editing concept.

6.3 Future Work

This section is separated in two parts: the first part, Section 6.3.1, suggests other web-based systems that may benefit from seamless editing. The second

part, Section 6.3.2, provides suggestions for further research into the concept of seamless editing.

6.3.1 Exploring Seamless Editing in Other Contexts

Inspired from the participants' suggestions in Table 6.1, the following sections suggest other web-based systems where seamless editing would be worth exploring, for research and/or commercial purposes.

Static Web Pages

Table 6.1 shows that seamless editing could be useful for editing static web pages. As described in Section 2.3.2, editing static web pages is a relatively cumbersome process. Users must remotely login to web servers, or deal with FTP, to manipulate content. This type of editing process would be significantly simplified with seamless editing.

Collaboration Systems

Web-based collaboration systems with seamlessly editable environments was not embarked upon in this thesis. Collaboration systems would be a worthwhile exploration, as supported in Table 6.1, as it shows that the participants in the user studies could see seamless editing being useful in Wikis. The simplification of the editing process in an open Wiki environment, as illustrated in the opening example for this thesis, may increase the amount of public contributions. A long term observational study comparing a Wiki with and without seamless editing, using a similar approach in this thesis for analysing quantitative data, would help address this question. Furthermore, such a study may also provide further insights into the seamless editing concept itself.

Digital Libraries

Digital libraries contain many spelling mistakes, as optical character recognition systems cannot extract full text from images with a 100% success rate. Many online digital libraries rely on readers who visit the web sites to con-

tribute spelling corrections for full-text content. For example, National Library of Australia's online Australian Newspapers archive[1] has integrated a system for obtaining spelling corrections from visitors as they read the content. The observational studies in Chapter 5 showed that minor edits, most likely being spelling or grammar corrections, were the most common type of edit on published content. Thus, seamless editing may be particularly useful for making it easier for visitors to contribute spelling corrections.

Xanadu on the Web

Vitali's pursuit for reviving project Xanadu's visions for the web would benefit from seamless editing (see Section 2.5.3 for a review of Vitali's work). In order to support a globally editable web, simplifying the authoring process is necessary — where a strong mechanism that connects reading and writing exists, so they can be carried out at the same time without technical skills [20]. An active project that resonates project Xanadu's visions, called ShiftSpace[2], provides a way to personalise any page on the web. It enables users to annotate and edit any web page that they visit. To edit web pages in ShiftSpace, users must create *Source-shifts*, where they edit HTML source for a selected area on a web page. The Seaweed framework would alleviate usability issues that ShiftSpace has with its reliance on HTML source editing. ShiftSpace's client is written as a GreaseMonkey script.[3] A fully working GreaseMonkey script version of the Seaweed framework has been written while exploring seamless editing. The marriage between ShiftSpace and Seaweed would make the entire web a seamlessly editable environment.

6.3.2 Suggestions for Further Research

This section provides suggestions for further researching the concept of seamless editing on the web that either came from limitations of the observational studies, or was out of scope of the focus of this thesis.

[1] See http://newspapers.nla.gov.au/ndp/del/home for project web site.

[2] See http://shiftspace.org for project web site.

[3] See http://www.greasespot.net for mote information.

Generalisation

Exploring seamless editing in a wide range of contexts, such as contexts described in Section 6.3.1, would provide insights into the types of situations that seamless editing is useful and is not. Furthermore, a well defined reference model for integrating Seaweed into any CMS, could be formulated from developing working prototypes of seamless editable environments for a range of different contexts.

Richer Quantitative Comparisons Between Editors

The logged activity in the observational studies could only distinguish whether an action was carried out via the Seaweed plugin or not (see Chapter 5). It would be worthwhile repeating the experiment with richer data, such that activity logs contain data that identifies the type of editor in WordPress used for carrying out each action. For example, whether a saving action (that is not carried out via Seaweed) was carried out via the HTML editor or standard visual editor for WordPress. A quantitative analysis on the action and editing activity on a richer data set would provide refined insights into the types of editors people prefer. Furthermore, the observational experiments repeated with a larger population, and without partial corruption of full HTML content, may provide more clear insights into the types of situations in which seamless editing is preferred, and in which it is not.

Seamless Structural Editing

Many content management systems, such as Wikis, do not allow, or generally discourage, the use of inline formatting on content (such as explicitly changing the font of a heading). These content management systems are based on the CSS design philosophy of separation of structure and presentation, as described in Section 2.1.3. These content management systems prevent authors from generating content with formatting that is inconsistent from the rest of the presentation of the CMS, and also help the authors to concentrate on the content and structure, not the presentation.

One pitfall of WYSIWYG editors is that they tend to conflict with the CSS design philosophy of separation of structure and presentation. Some people may prefer editing source (such as WikiML for Wikis, or HTML) because they provide a clear view on the structure and content that they are editing. WYSIWYG editors hides the actual structure of content, and only give a presentational view. Seaweed suffers from this same problem, as it essentially supports WYSIWYG editing interactions within published views.

An editing system for the web that seeks to bridge the gap between source editing and WYSIWYG editing is the WYSIWYM (What You See Is What You Mean) editor.[4] It provides a visual editor that exposes structure, and restricts formatting, to adhere to the CSS design philosophy. A seamless approach for providing structural editing, similar to the WYSIWYM editor, would be worth exploration: as it would include some of benefits that HTML source editing has over WYSIWYG editors.

6.4 Final Conclusion

The user studies provided clear indication that seamless editing is a useful, usable and satisfying way of authoring content on blogs. Seamless editing has been shown to be a fundamental concept that can be applied universally on the web. It significantly simplifies the authoring process by eliminating the existence of modes, and in effect, reduces the amount of cognitive complexity involved, in comparison to moded ways of editing. Clearly seamless editing is an ideal candidate for paving the way into a new revolution for authoring content on the web.

[4]See http://www.wymeditor.org for project web site.

Appendices

Appendix A

Seaweed Framework Web Browser Support

Web Browser	**Operating System**	**Supported Versions**	**Untested Platforms**
Microsoft Internet Exporer	Windows	6-8	*None*
Mozilla FireFox	Windows, Mac, Linux	1.5-3.5	*None*
Apple Safari	Windows, Mac	3-4	versions 2 and below
Google Chrome	Windows	*All*	Non-Windows OS releases
Opera	Linux, Windows	9-10	versions 8 and below, Mac OS

These were tested both manually and via a suite a unit tests. It is possible that the Seaweed framework is supported by other web browsers that were not tested.

Appendix B

Seaweed's Table of Actions

Name	Description
InsertText(str, n, i)	Inserts text (*str*) within a text node (*n*) at a character index (*i*).
InsertHTML(html, pn, i)	Inserts HTML (*html*) within a node (*pn*) at a given position (*i*). The position is either the character index if the given node (*pn*) is a text node, or the child-index if an element node.
RemoveText(n,i,len)	Removes text within a given text node (*n*). The text removed starts at a character index (*i*) and spans to a length of (*len*) charactors.
RemoveDOM(sn, si, en, ei)	Collapses a fragment range between a start point (*sn, si*) and end point (*en, ei*) in the DOM.
Blockquote(sn, en)	Encapsulates a DOM node range (*sn, en*) with a <*Blockquote*> element.
SplitContainer(n,i)	Splits the container element in two at the given text node (*n*) and character index (*i*).
ChangeContainer(tag, sn, en)	Changes all block-level elements in a given DOM node range (*sn, en*) to a given block-level HTML element (*tag*). Encapsulates inline-level DOM-trees (that have no parent block-level elements) with a new element of the type (*tag*).

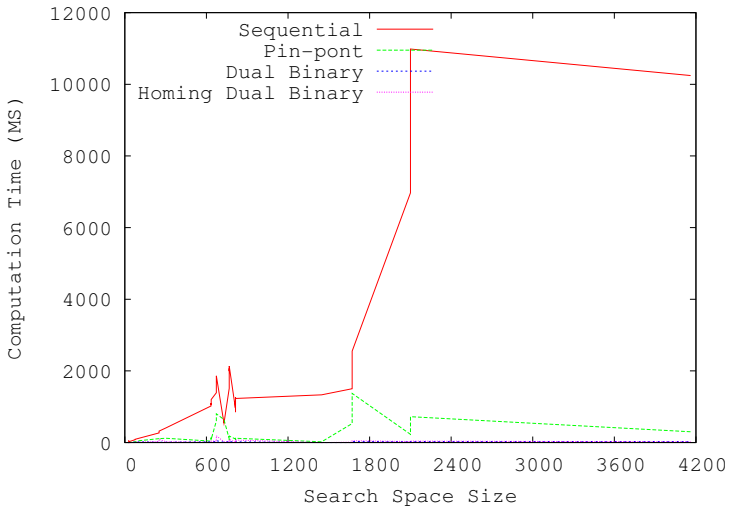
Continued next page...

Continued...

Name	**Description**
Itemize(t, sn, en)	Creates/converts a list of items of/to type bullets or numbers (t) in a given range (sn, en). If the top-level block-level elements in the given range are all $<Li>$ elements, they are removed instead.
DemoteItem(sn, en)	Demotes all list items in the given DOM node range (sn, en) to a lower level.
PromoteItem(sn, en)	Promotes all list items in the given DOM node range (sn, en) to a higher level.
Format(t, v, sn, si, en, ei)	Formats/unformats a fragment range between a start point (sn, si) and end point (en, ei) in the DOM to a value (v). Supported format types (t) are: "bold," "italics," "underline," "strike," "color," "backcolor," "fontsize," "fontfamily" and "link".
Indent(v, sn, en)	Set margins to a value of (v) pixels for all block-level elements an a DOM node range (sn, en).
TextAlign(a, sn, en)	Sets CSS alignment to value (a) for all block-level elements an a DOM node range (sn, en).
SpellMark(sn, si, en, ei)	wraps an adjacent set of inline-elements and text nodes in a given fragment range between a start point (sn, si) and end point (en, ei) in the DOM with a spelling error wrapper element.
SpellUnmark(n)	Removes a spelling error element that wraps node (n).
SpellCorrect(n, str)	Replaces a spelling error element (n), and it's contents, with the spelling correction text (str).

Appendix C

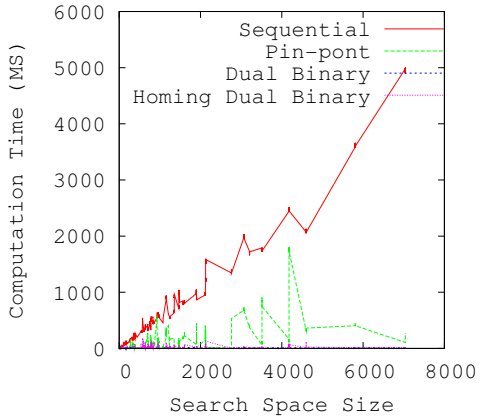
Cursor Algorithm Performance Data



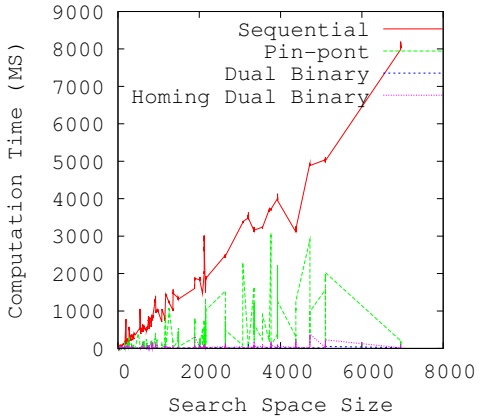
Raw small-scale test results excluded from full scale analysis. From Internet Explorer 7.0 running on a Windows XP virtual machine. Part-way through this benchmark session Internet Explorer's performance dramatically drops.

Continued next page...

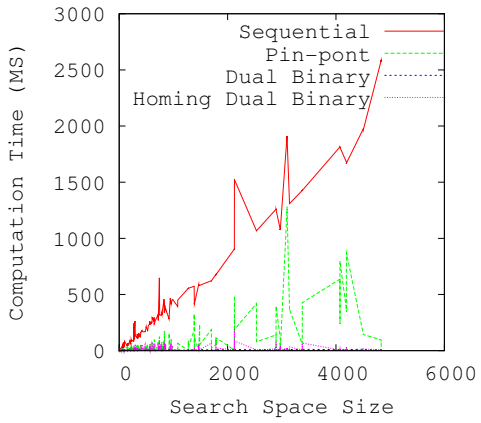
Continued...



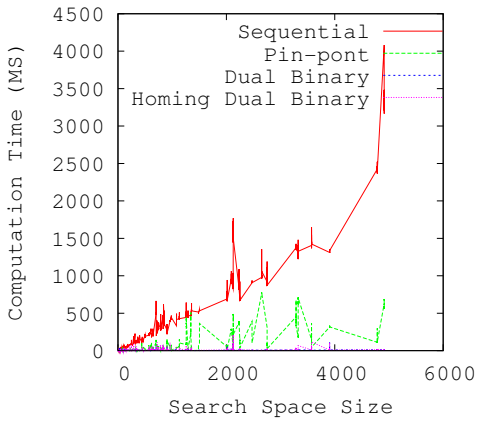
(a) A session running on Firefox 2.0 (Linux).



(b) A session running on Opera 9.8 (Linux).



(c) A session running on Safari 4.0 (Mac).

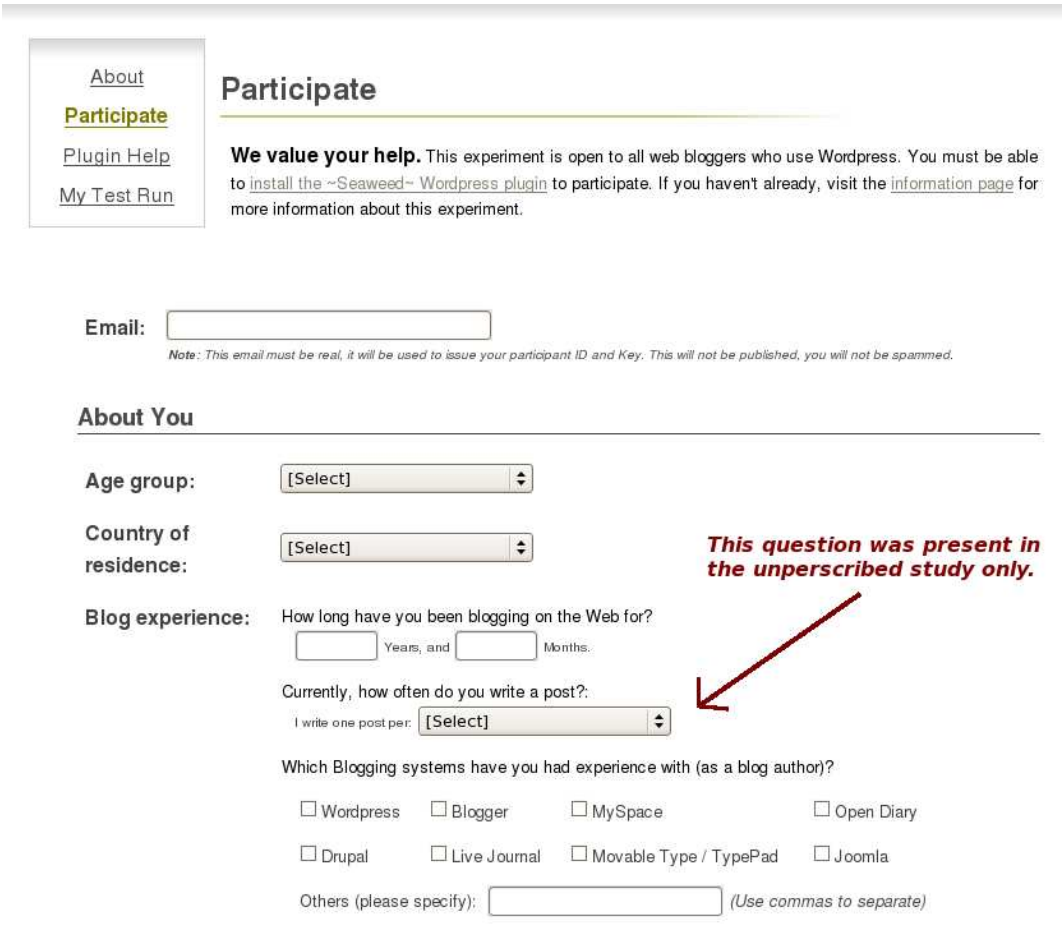


(d) A session running on Chrome 3.0 (Windows).

Raw results for a benchmark sessions.

Appendix D

Online Registration Form



Continued next page...

Computer experience:

On average how many hours a week do you spend on a computer (in your own time and at work)?

Between [ ] and [ ] hours.

Rate your experience from none to highly-experienced (5):

Office suite application experience: *(e.g. MS Office, Google Docs or Open Office)*:	○ None	○ 1	○ 2	○ 3	○ 4	○ 5
Web page editing application experience: *(e.g. frontpage or dreamweaver)*	○ None	○ 1	○ 2	○ 3	○ 4	○ 5
General visual/WYSIWYG editor experience: *(e.g. visual editors for emails or blogs)*	○ None	○ 1	○ 2	○ 3	○ 4	○ 5

Select/tick the statements which apply to you:

- ☐ I have an IT related job
- ☐ I have an IT related hobby or interest
- ☐ I have maintained/updated a website other than a blog
- ☐ I own, or have owned, a website
- ☐ I understand raw HTML syntax (for example "Hello")

Accessibility:

Do you use any software/tools which aid your web experience? If so, please detail your condition and the software/tools you use on the Web:

(for example, use of screen readers or increasing text size in browsers).

Are you human?

Please enter the words below

(This step is used to prevent malicious computer programs from spamming our research server)

Research Consent ~ MUST READ

Please **DO NOT GLAZE OVER THIS**. Once your test run begins your **ACTIVITY WILL BE LOGGED**. We also encourage you to read the experiment information page. Please read on:

1. Project Title

Seamless Editing on the Web.

2. Purpose

This experiment is conducted to both improve the quality of the Seaweed software and to progress an academic research project.

3. What is this research project about?

To explore a new way of editing content on the Web called "seamless editing". The exploration compares seamless editing with the existing ways of editing content, and investigates how difficult it would be for the Web to shift to this new way of editing.

Print Consent Information

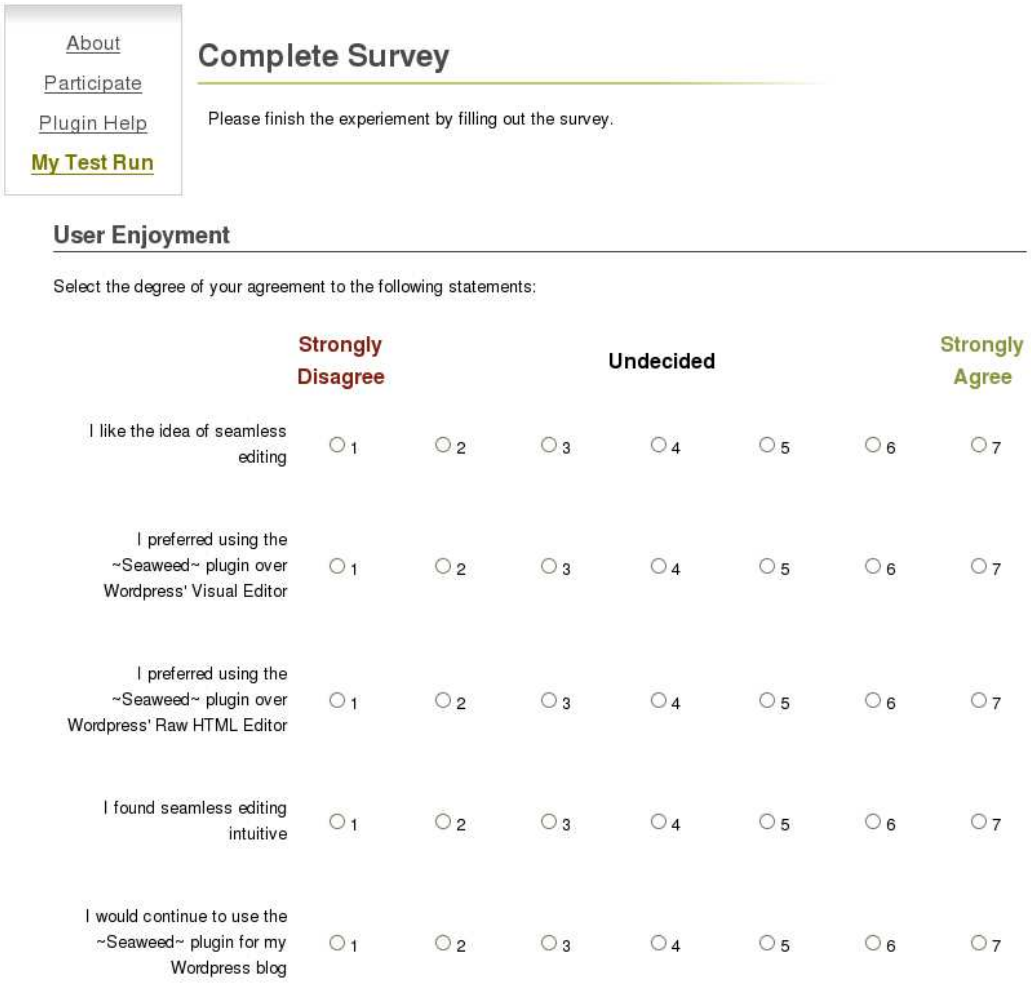
☐ **I agree to participate in this study under the terms and conditions outlined in the consent information above**

"Thank you in advance" ~ Brook Novak *(lead researcher)*

Register

Appendix E

Online Survey



About
Participate
Plugin Help
My Test Run

Complete Survey

Please finish the experiement by filling out the survey.

User Enjoyment

Select the degree of your agreement to the following statements:

	Strongly Disagree			Undecided			Strongly Agree
I like the idea of seamless editing	○ 1	○ 2	○ 3	○ 4	○ 5	○ 6	○ 7
I preferred using the ~Seaweed~ plugin over Wordpress' Visual Editor	○ 1	○ 2	○ 3	○ 4	○ 5	○ 6	○ 7
I preferred using the ~Seaweed~ plugin over Wordpress' Raw HTML Editor	○ 1	○ 2	○ 3	○ 4	○ 5	○ 6	○ 7
I found seamless editing intuitive	○ 1	○ 2	○ 3	○ 4	○ 5	○ 6	○ 7
I would continue to use the ~Seaweed~ plugin for my Wordpress blog	○ 1	○ 2	○ 3	○ 4	○ 5	○ 6	○ 7

Continued next page...

Plugin Design

Detail any bugs / technical problems that you encountered:

Detail any feature(s) would you *add* to the Seaweed plugin:

Detail any feature(s) would you *remove* from the Seaweed plugin:

Thoughts / Opinions

List web-sites / web-systems where you think Seaweed (seamless editing) would be useful (other than blogs):

General comments:

Credits

You have achieved "tester" status in the ~Seaweed~ project. If you want to be recognized you can provide your name or an alias to be listed as a "tester" in the the ~Seaweed~ project. Credits will be forever listed on the project website, help documentations and within the project source distributions.

Name/Alias: *(Note: this will not be stored with your logged data).*

Your Blog

In case your activity was not logged correctly, please enter the URL of your blog's main/index page.

Blog URL: *E.G: http://myblog.com.*

Submit Survey

Appendix F

Quirks in Seaweed During Both Studies

Prescribed Study	Unprescribed Study
Pasting for the first time in Internet Explorer, an extra 'v' would add on the end of the pasted text.	Inserting images would sometimes place the image on a new line below the cursor (instead of on the actual line where the user intended it to be placed).
Pasting into the URL-inputs for the link editor did not work in Safari.	In some themes/plugin-configurations, images would only align left despite being aligned right/center via Seaweed's image editor.
Copy and pasting in general sometimes would not work in Safari browsers.	In some cases, the spacebar on the keyboard would not work when creating new posts via Seaweed.
In some cases inserting space after an existing whitespace (that is, creating double whitespaces) removes both of the whitespace instead.	Shortcodes transformed directly by plugins rather than via WordPress' shortcode API were not being preserved by Seaweed.
Assigning categories to new posts would not work.	Clashes with the *ajax-edit-comments* WPMU plugin. That is, the Seaweed plugin does not work at all when ajax-edit-comments on WPMU is enabled.
Paragraphs sometime would loose all line wraps when using undo and redo in Firefox.	

References

[1] History of wikipedia. Web page. Retrieved from http://en.wikipedia.org/wiki/History_of_Wikipedia on 16th of December 2009.

[2] Metaphors for interface design. April 1987. Paper presented at NATO-sponsored workshop on Multimodal Dialogues Including Voice (Venaco, Corsica, France, September 1986).

[3] Robert M. Akscyn, Donald L. McCracken, and Elise A. Yoder. KMS: a distributed hypermedia system for managing knowledge in organizations. *Commun. ACM*, 31(7):820–835, 1988.

[4] Paul Bausch, Matthew Haughey, and Meg Hourihan. *We Blog: Publishing Online with Weblogs*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[5] Tim Berners-Lee and Mark Fischetti. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Harper San Francisco, 1999. Foreword By-Dertouzos, Michael L.

[6] Eric Bier and Ken Pier. Sparrow web: group-writable information on structured web pages. In *CHI '03: CHI '03 extended abstracts on Human factors in computing systems*, pages 634–635, New York, NY, USA, 2003. ACM.

[7] Rebecca Blood. Weblogs: A history and perspective. *Rebecca's Pocket*, September 2000. Retrieved from http://www.rebeccablood.net/essays/weblog_history.html on December 11th 2009.

[8] Bob Boiko. *Content Management Bible*. Wiley Publishing, Inc., Crosspoint Boulevard, Indianpolis, second edition, 2005.

[9] Paul Browning. Get tooled up: Through the web authoring tools. *Ariadne*, (39), April 2004. Retrieved from

http://www.ariadne.ac.uk/issue39/browning on 17th of December 2009.

[10] Vannevar Bush. As we may think. *SIGPC Note.*, 1(4):36–44, 1979.

[11] Bay-Wei Chang. In-place editing of web pages: Sparrow community-shared documents. *Computer Networks and ISDN Systems*, 30(1-7):489 – 498, 1998. Proceedings of the Seventh International World Wide Web Conference.

[12] Brenda Chawner and Paul H. Lewis. WikiWikiWebs: New ways of interacting in a web environment. 1999. Handout prepared for the LITA Forum 2004, St.Louis, Missouri.

[13] Jake Crosby. What makes a CMS useful? Degree report, University of Waikato, 2009.

[14] Ward Cunningham. Wiki history. Web page. Retrieved from http://c2.com/cgi/wiki?WikiHistory on 16th of December 2009.

[15] Nicole B. Ellison Danah M. Boyd. Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication*, 13:210–230, 2008.

[16] Alana Dix, Janet Finlay, Gregory D. Abowd, and Beale Russell. *Human-Computer Interaction*. Pearson Education Ltd, 3 edition, 2004.

[17] James Gillies and Robert Cailliau. *How the Web was born : the story of the World Wide Web*. Oxford University Press, 2000.

[18] M J Harris and R Rosenthal. Mediation of interpersonal expectancy effects: 31 meta-analyses. *Psychological Bulletin*, (97):435–693, 1985.

[19] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Hum.-Comput. Interact.*, 1(4):311–338, 1985.

[20] Angelo Di Iorio and Fabio Vitali. Web authoring: A closed case? *Hawaii International Conference on System Sciences*, 4:95b, 2005.

[21] Jeff Johnson, Teresa L. Roberts, Us West, Advanced Technologies, William Verplank Idtwo, David C. Smith, Charles H. Irby, Marian Beard, Metaphor Computer Systems, and Kevin Mackey Xerox. The Xerox Star: A retrospective. *IEEE Computer*, 22:11–29, 1989.

[22] J. Kolbitsch and H. Maurer. The transformation of the web: How emerging communities shape the information we consume. *Journal of Universal Computer Science*, 12(2):187–213, 2006.

[23] Bo Leuf and Ward Cunningham. *The Wiki way: quick collaboration on the Web*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[24] Charlie Lindahl and Elise Blount. Weblogs: Simplifying web publishing. *Computer*, 36(11):114–116, 2003.

[25] Brad A. Myers. A brief history of human-computer interaction technology. *interactions*, 5(2):44–54, 1998.

[26] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[27] Ted Nelson. Xanadu: document interconnection enabling re-use with automatic author credit and royalty accounting. *Information Services and Use*, 14, June 1994.

[28] Tim O'Reilly. What is web 2.0: Design patterns and business models for the next generation of software. Technical report, March 2005. Retrieved from http://oreilly.com/web2/archive/what-is-web-20.html on December 11th 2009.

[29] James Robertson. In-context versus back-end authoring, April 2008. Online article. Retrieved from http://www.steptwo.com.au/papers/cmb_incontext/index.html on 16th of December 2009.

[30] B. Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, 1983.

[31] D. C. Smith and Et. Designing the Star user interface. *Byte*, 7(4):242–282, 1982.

[32] John C. Thomas and Wendy A. Kellogg. Minimizing ecological gaps in interface design. *IEEE Software*, 6(1):78–86, 1989.

[33] Fabio Vitali. Creating sophisticated web sites using well-known interfaces. In *Proceedings of HCI International 2003 Conference*, Crete (Greece), June 2003.

[34] Fabio Vitali and Angelo Di Iorio. A xanalogical collaborative editing environment. In *In Proceedings of the second international workshop on Web document Analysis*. University of Liverpool, 2003.

[35] Fabio Vitali and Angelo Di Iorio. Writing the web. *Journal of Digital Information*, 5(1), May 2004.

[36] Fabio Vitali and Angelo Di Iorio. From the writable web to global editability. In *HYPERTEXT '05: Proceedings of the sixteenth ACM conference on Hypertext and hypermedia*, pages 35–45, New York, NY, USA, 2005. ACM.

[37] Gottfried Vossen and Stephan Hagemann. From version 1.0 to version 2.0 : a brief history of the web. Technical report, Dept. of Management Systems, University of Waikato, 2007.

[38] Jenna Wortham. After 10 years of blogs, the future's brighter than ever, December 2007. Online article. Retrieved from http://www.wired.com/entertainment/theweb/news/2007/12/blog_anniversary on 16th of December 2009.

[39] Ka-Ping Yee. Pyxi: A browser and editor for Xanadu hypertext. Retrieved from http://lfw.org/repo/pyxidemo-last-last.pdf on Decemeber 11th 2009.

[40] Jeffrey Zeldman. *Designing with Web Standards (2nd Edition) (Voices That Matter)*. Peachpit Press, Berkeley, CA, USA, 2006.