KeySurf - A keyboard Web navigation system for persons with disabilities

by

Leonhard Spalteholz
B.Eng., University of Victoria, 2006

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

© Leonhard Spalteholz, 2012
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

KeySurf - A keyboard Web navigation system for persons with disabilities

by

Leonhard Spalteholz
B.Eng., University of Victoria, 2006

Supervisory Committee

Dr. Kin Fun Li, Co-Supervisor
(Electrical and Computer Engineering)

Dr. Nigel Livingston, Co-Supervisor
(CanAssist)

Dr. Stephen Neville, Departmental Member
(Electrical and Computer Engineering)

Dr. Margaret Anne Storey, Outside Member
(Computer Science)

ABSTRACT

Supervisory Committee

Dr. Kin Fun Li, Co-Supervisor
(Electrical and Computer Engineering)

Dr. Nigel Livingston, Co-Supervisor
(CanAssist)

Dr. Stephen Neville, Departmental Member
(Electrical and Computer Engineering)

Dr. Margaret Anne Storey, Outside Member
(Computer Science)

For people with motor disabilities unable to control a pointing device, there is a need for an efficient keyboard Web navigation method. The current state of keyboard navigation tools is surveyed and discussed. The standard tab key navigation system is widely regarded as insufficient for practical keyboard access to the Web. Other techniques, such as identifier navigation — where links are activated by entering their numeric code — are found to be much more efficient, but possibilities remain for improvement. A technique based on selecting links by text search provides an alternative method of keyboard navigation, but does not minimize the number of required keyboard inputs, which is important for individuals not capable of fast typing input.

A new technique called KeySurf is proposed, which aims to make text search navigation more efficient and intuitive by estimating which elements are more likely to be selected by the user, and then allowing those elements to be selected with fewer keystrokes. Web page elements without text associated with them are assigned

labels based on algorithms that make generated labels predictable to the user. Various constraints on the searching algorithm are implemented to decrease the keystroke cost of selections, such as selecting visible elements first, matching the first characters of labels, and prioritizing visually prominent elements. In addition, the user's browsing history is used to calculate a measure of page and element interest in order to make interesting elements easier to select.

KeySurf performance is examined in three experiments: an automated analysis of keystroke cost of element selection on randomly selected Web pages, a small study involving four individuals with motor disabilities to compare KeySurf and mouse use, and a study of 11 individuals browsing the Web with KeySurf collecting data passively in the background. The automated selection test calculates the number of keys necessary to activate each link for 48,182 links, resulting in a mean of 2.69 keystrokes. The study involving individuals with disabilities shows that KeySurf can be faster than mouse use if the user is able to type 2 or 3 keystrokes faster than pointing to a target using their pointing device. The study with 11 non-disabled individuals shows that for 4,601 recorded clicks, KeySurf would have required 2.38 keys per selection. Comparing mean keystroke cost for pages containing similar numbers of elements, we find that for real Web sessions KeySurf can decrease keystroke cost by 15% compared to the simulation results by anticipating which elements a user is likely to select. A keystroke level model of tabbing, ID navigation, and KeySurf indicates that the predictability of ID navigation makes it more efficient for faster typists, while KeySurf is likely to be faster for slower typists.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **v**

List of Tables **viii**

List of Figures **ix**

Acknowledgements **xi**

1 Introduction **1**
1.1 Motivation 2
1.2 What is Meant by Web Navigation? 3
1.3 Organization of this Thesis 3

2 Background **4**
2.1 Keyboard Navigation 5
2.1.1 Early Text-Based Browsers 5
2.1.2 Tabbing 6
2.1.3 Access Keys 7
2.1.4 ID Navigation 9
2.1.5 Search-Based Navigation 10
2.2 Switch Input 11

3 KeySurf **15**
3.1 Target Users 16
3.2 System Design 17
3.3 Element Labelling 18

3.3.1 Label Generation 19
3.3.2 Image Links 21
3.4 User Centric Search 22
3.5 Element Selection Shortcuts 27
3.6 Reducing Error Rate 29
3.7 Accessing Bookmarks 29
3.8 Using Web Browsing History 30
3.8.1 Inferring User Interest 31
3.8.2 Web Page Keyword Ranking 33
3.8.3 User Searches 34
3.8.4 Applying User Interest Keywords 35
3.8.5 Balancing Performance and Prioritization Complexity 35
3.9 Input Devices for KeySurf 36
3.9.1 Speech Input 36
3.9.2 Ambiguous Layout On-screen Keyboards 38
3.9.3 Switch Input 39
3.10 Limitations of KeySurf 42

4 Evaluation by Simulation and a Pilot Study 44
4.1 A Simulation to Measure Navigation Efficiency 44
4.2 Initial Performance Indications (4-User Mini-Study) 46
4.2.1 Participants 46
4.2.2 Experimental Design 47
4.2.3 Results and Discussion 48

5 Evaluation on Real-World Web Sessions 51
5.1 Study Design and a Note on Studies Involving Persons with Disabilities 52
5.2 Participants and Recruitment 54
5.3 Data Collection 54
5.4 Results and Discussion 55
5.4.1 KeySurf Performance 57
5.4.2 Use of Selection Shortcuts 59
5.4.3 Impact of History 60
5.5 Model-Based Comparison to Other Techniques 62
5.5.1 Mouse Point and Click 64

5.5.2 Tabbing 64
5.5.3 ID Navigation 65
5.5.4 KeySurf 67
5.5.5 Comparison 68
5.6 Discussion 72

6 Conclusions and Future Work **73**

Appendix A Data Logged by Passive KeySurf **75**

Appendix B List of WebSpeak Keywords **77**

Appendix C JavaScript Huffman Code Implementation **78**

Bibliography **82**

List of Tables

Table 4.1 Input devices used by test subjects. 47

List of Figures

Figure 2.1 The Lynx Web browser displaying a Web page. 6
Figure 2.2 Typical scanning order in a three column Web page [45]. . . . 7
Figure 2.3 Shortcuts in Konqueror showing visual clutter and page occlusion. 9
Figure 2.4 “Find as you type” showing unintuitive matching behaviour. . 10
Figure 2.5 Row-column scanning keyboard (from Shein [46]). 12
Figure 2.6 High frequency layout for a row-column scanning keyboard. . . 12

Figure 3.1 KeySurf system overview. 18
Figure 3.2 HTML form elements and their associated KeySurf labels. . . . 20
Figure 3.3 A Web page showing multiple links starting with the same letters. 21
Figure 3.4 Image buttons with identical alternate text in KeySurf. 22
Figure 3.5 Numbered overlays for image buttons without alternate text. . 22
Figure 3.6 Text and image link with identical targets. 23
Figure 3.7 The set of visible elements (1), and all elements on a page (2). 25
Figure 3.8 The user centric search process. 26
Figure 3.9 Web page showing multiple links with the same text. 28
Figure 3.10 Highlighted elements after typing “h”. 28
Figure 3.11 The bookmarks selection screen in KeySurf. 30
Figure 3.12 The interface of CanAssist’s Dynamic Keyboard. 39
Figure 3.13 Element highlighting showing the effect of priority estimation. 42

Figure 4.1 Average required keystrokes per element selection. 45
Figure 4.2 Penny Giles track ball [29]. 47
Figure 4.3 Mean link activation times with KeySurf and regular Web browser. 49

Figure 5.1 Number of pages loaded and links clicked by participant. . . . 56
Figure 5.2 Unique pages and domains visited by participant. 57
Figure 5.3 Mean keystroke cost by participant. 58
Figure 5.4 Elements per Web page by participant. 58

Figure 5.5 Element activations benefiting from a numbered shortcut. . . . 60
Figure 5.6 Number of elements and mean keys to select by history group. 61
Figure 5.7 Average ID length vs KeySurf selection keys. 67
Figure 5.8 Element activation time by system and typing speed. 69
Figure 5.9 Element activation time for ID navigation, KeySurf, and mouse. 70
Figure 5.10 Element activation time for KeySurf and mouse navigation. . . 71

ACKNOWLEDGEMENTS

I would like to thank:

Dr. Kin Fun Li, for your enduring patience and support on what turned into a very long process. Also for pushing me to write papers and then giving me the opportunity to present them, which helped me immensely.

Dr. Nigel Livingston and CanAssist, for inspiring me to get involved, supporting me at every step, and funding my research.

my wife for not bugging me too much about when I was going to finish.

Chapter 1

Introduction

Even with the myriad of assistive devices available to aid people with disabilities in accessing a computer, controlling the mouse pointer is difficult or impossible for many people with motor disabilities. This problem has been recognized since the beginning of graphical user interfaces, and much work has been done to facilitate keyboard accessibility for those users unable to use a pointing input device. In general, this has resulted in modern graphical desktop applications being efficiently accessible for both keyboard and mouse users. However, with the proliferation of the Web, an increasing amount of time is spent accessing Web pages and Web applications via a Web browser instead of using platform-native applications running on the user's computer. Especially for individuals with physical and mobility impairments that may have trouble accessing traditional paper-based resources, it is critical that the World Wide Web be accessible regardless of the computer access method.

For users without the option of using a pointing device, navigating Web pages with a keyboard is often very cumbersome. Although guidelines for Web accessibility — which include best practices for keyboard accessible design — have been developed by the Web Accessibility Initiative [57], adoption rates amongst Web authors are still very poor and seem to be not significantly improving [3, 23, 56, 22]. This has the effect that any tool to facilitate keyboard accessibility for the Web cannot solely rely on the implementation of existing or new accessibility standards. Currently, an alternative Web navigation system is only practically useful if it is compatible with a large majority of popular Web sites, regardless of their conformance to accessibility standards.

Guideline 2.1 of the Web Content Accessibility Guidelines (WCAG) specifies that Web authors should "Make all functionality available from a keyboard" [62]. However,

the specifications do not mention the access method that should be used to navigate the page using the keyboard, or make mention of any efficiency targets for keyboard access. Schrepp discusses the disconnect between usability for people with disabilities and conformance to Web standards, noting that Web sites can be fully compliant with guidelines and yet remain essentially unusable for people with certain disabilities [44]. Conversely, Web sites not conforming to accessibility guidelines can still be quite usable by persons with disabilities. In our experience working with persons with disabilities, many individuals unable to operate a mouse are excluded from Web access not because it is functionally impossible but because of the required effort outweighing the benefits of Web access.

KeySurf is a keyboard Web navigation system for people unable to use a pointing device. Essentially, it allows users to browse the Web with a keyboard or equivalent text input device. While there are existing methods to access the Web with a keyboard, the goal of KeySurf is to present an interface to keyboard users that gives them a similar quality of Web browsing experience as mouse users. This means that the time and effort required to navigate between and within Web pages with KeySurf should be comparable to that required by a regular mouse user. Schrepp and Fischer propose that for a keyboard access method to be a viable alternative to the mouse, the time required to navigate by keyboard should not exceed twice the time required for navigation by mouse [45]. Naturally for a user who cannot use a pointing device at all, the primary characteristic of any alternative Web navigation system is that it must be efficient enough such that the benefit of accessing the Web outweighs the effort required to do so.

1.1 Motivation

Although KeySurf is designed to be generally useful to many people with physical disabilities, the initial motivation to create the system came from a CanAssist[1] client with Cerebral Palsy. This young man is an avid computer user and very technically savvy. Although he must be strapped to his chair to control involuntary movements, he is proficient at typing on a regular keyboard supported on an angle with a specialized keyboard stand. Using his lips, nose, and chin to press keys on the keyboard, he is able to operate any desktop application that can be operated by the keyboard. As

[1] CanAssist is a non-profit entity at the University of Victoria that develops technology and devices for persons with disabilities (`www.canassist.ca`)

keyboard accessibility of desktop applications has been emphasized in user interface guidelines of most modern operating systems [28, 37], he is able to efficiently control most applications.

However, navigating on the Web was still a challenge for him, as it required him to either press the tab key many times to move from element to element, or use the "Mouse Keys" accessibility feature to move the mouse pointer with the number keys on the keyboard. Both of these methods were not fast enough to provide practical Web access. Thus the central goal of this work is to create a system that reduces the effort required to navigate the Web to a level where it is a practical tool for this user. Many of the design decisions and features of KeySurf came out of requirements for this particular user.

1.2 What is Meant by Web Navigation?

The term "Web navigation" in the context of this work, refers to the low level mechanics of moving from one hypertext page to another, manipulating the common controls found on hypertext pages, and navigating within a page. In other words, KeySurf aims to provide a keyboard alternative for actions that are usually performed with a mouse click in a regular Web browser where such an alternative doesn't already exist. The most common actions include clicking on text or image hyperlinks, activating buttons, focussing text fields and areas, and manipulating form controls of all kinds. These Web page controls are collectively referred to as "active elements" or simply page elements in this work.

1.3 Organization of this Thesis

In Chapter 2, we present the existing systems for Web navigation, focusing on work that has been done for keyboard users. Chapter 3 describes the design of KeySurf, and how we use the characteristics of Web pages and the browsing history of users to estimate which elements a user is likely to select. A small evaluation by simulation and a pilot study is presented in Chapter 4, while Chapter 5 describes our evaluation of KeySurf using recorded real Web sessions. Chapter 6 concludes this work.

Chapter 2

Background

For keyboard users, the accessibility of the Web is still inferior to that of mouse users. Schrepp surveys the accessibility of Web sites in 2006 for both mouse and keyboard users [43], with the conclusion that while the state of Web site design for mouse users is quite good, the majority of Web sites are very cumbersome to navigate for keyboard users (using tab key navigation). For popular online resources, even able-bodied keyboard users require between 4 to 10 times as much time to complete navigation tasks as mouse users. Six years later, tab key navigation remains the only widely implemented method of keyboard navigation while web page complexity has increased, leading to the conclusion that keyboard accessibility of web sites is still insufficient.

The problem of navigating the Web for people with physical disabilities has been approached from various angles. This chapter discusses research on improving Web navigation accessibility for people with disabilities, focusing on currently available keyboard access methods. We only discuss work related to accessibility of interaction with the Web rather than its presentation (such as for visually impaired Web users).

Since current Web sites are optimized for pointer based input devices, improving the accessibility of existing pointer control schemes and devices is the most direct and common method of facilitating access to the Web for people unable to accurately control a standard computer mouse. This may involve specialized alternative pointer control devices such as joysticks, track balls, head or eye tracking systems working in conjunction with software filters to improve pointing accuracy. Trewin et al. have proposed a method of steadying the mouse cursor during clicks, such that users are able to activate a target with a click after they have acquired it with the mouse pointer [54]. Bilmes at al. have provided an alternative to mechanical cursor control

entirely, by proposing a Vocal Joystick to allow users to control a mouse pointer with continuous voice control [6]. In general we feel that the variety and quality of adaptations for pointer based input is sufficient to allow users of these devices to access the Web efficiently. In the following sections, we focus instead on methods for keyboard and switch access to the Web, which are less well developed.

2.1 Keyboard Navigation

The WCAG 1.0 standard specifies that pages should be designed for device independence [61], such that functionality does not depend on a particular input device. Version 2 of WCAG clarifies this point, specifying that all functionality must be accessible with a keyboard interface [62]. A keyboard interface is any interface that provides keystroke input to the Web page. This may be a regular computer keyboard, an on-screen or soft keyboard, or voice recognition with keyboard emulation. Many alternative input devices are capable of emulating a keyboard, making a keyboard-accessible Web site also theoretically accessible to those devices. The WCAG standards do not specify the method that should be used to enable keyboard access to the functionality of a Web page, instead leaving implementation details to the browser. The following sections examine the existing keyboard navigation methods and standards.

2.1.1 Early Text-Based Browsers

In previous years, when most Web sites were written to earlier HTML standards and visually much simpler than today's graphically rich and dynamic pages, text-based browsers provided a good alternative for those people requiring keyboard or screen reader friendly access to the Web. Text-based Web browsers, such as Lynx [34] (Figure 2.1) or Links [18] and its derivatives, rendered a Web page to a terminal, omitting multimedia content and simplifying layout. Users could navigate to links on a page by moving the focus with the arrow keys, or having the browser assign numbered shortcuts to links on the page. These approaches worked well on simpler pages, but are not capable of delivering the full experience of the World Wide Web.

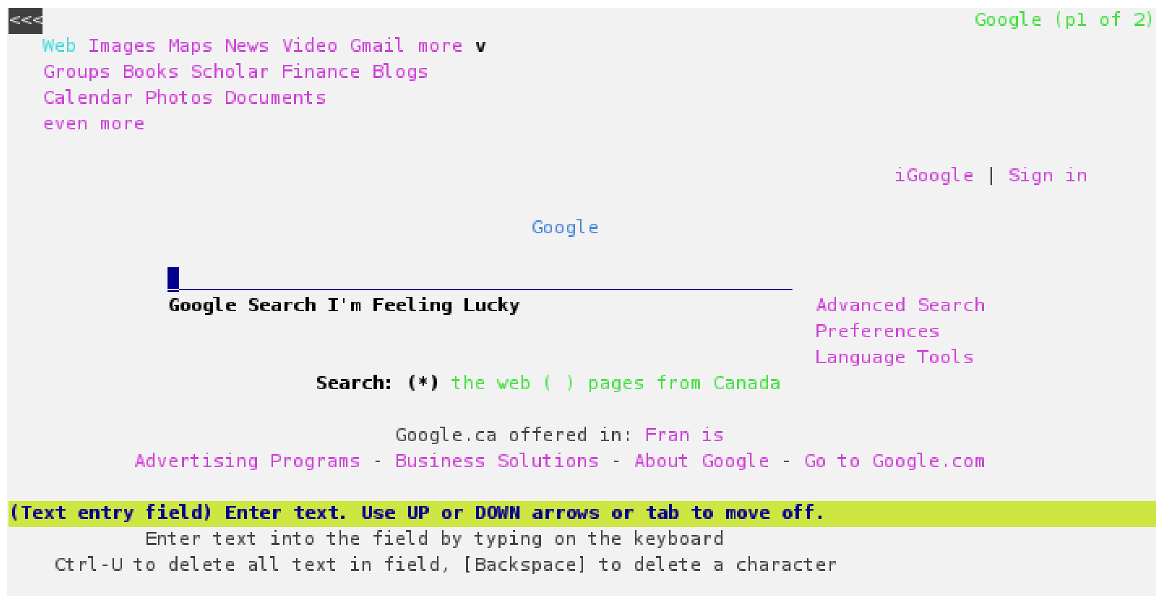

Figure 2.1: The Lynx Web browser displaying a Web page.

2.1.2 Tabbing

In the currently available general purpose Web browsers (Mozilla Firefox, Google Chrome, Microsoft Internet Explorer, Apple Safari, etc), tabbing is the only common keyboard navigation interface. As a page is loaded, active elements within it are placed into the tab chain, in the order they are encountered in the Web page source code. By pressing the tab key, focus can be moved iteratively through this tab chain, from one item to the next until the desired item is reached. The enter key activates the currently focused item.

Although scanning elements in a top to bottom order is logical for simple single column Web pages, sites with more complicated layout containing multiple separate vertical and horizontal groups of elements can make in-place element scanning very difficult to follow. Figure 2.2 shows the progression of tab focus from element to element in a typical Web page layout. The tab chain order is defined at the source level, so depending on how the Web page is written, the focus could move entirely unpredictably based on the visual design of the page. In this example, one can see that to access a link in the rightmost pane (which often contains secondary navigation elements), a user would have to scan through every element in the page's navigation and content.

HTML elements that can be focused with the keyboard also support an attribute

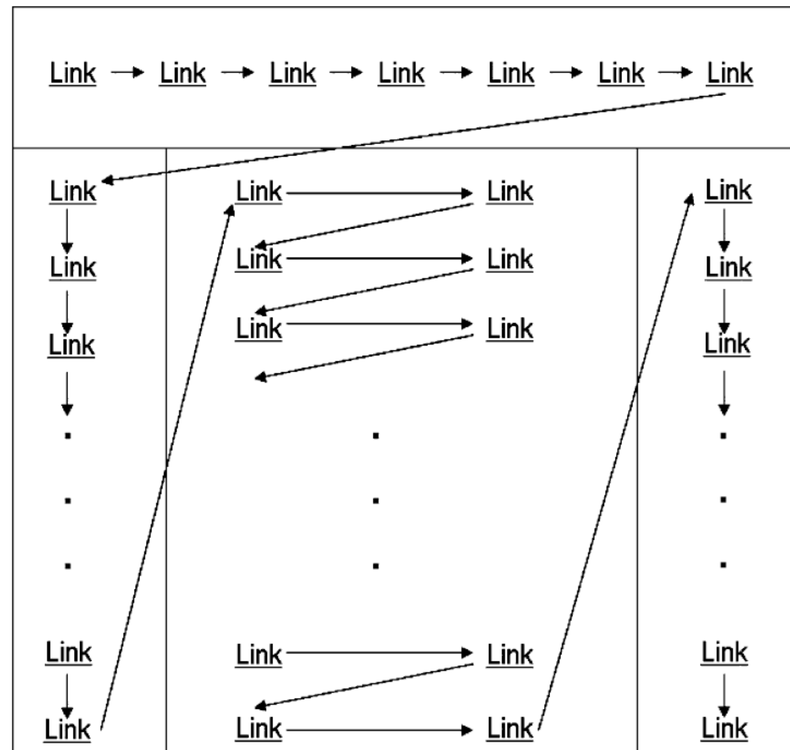

Figure 2.2: Typical scanning order in a three column Web page [45].

called *"tabindex"*, which allows the Web page author to assign elements a specific index in the tab chain. Elements without the tab index attribute are appended to the tab chain after those where it has been specified. For example, a page author could assign the main navigation links tab indices, which would allow a keyboard user to jump to them very quickly, regardless of other elements on the page. However, since the tabindex attribute is not shown visually, this can make predicting focus movement even more difficult for users who have not previously encountered the Web page.

2.1.3 Access Keys

Access keys are a feature of the HTML standard that allow the Web developer to mark a focusable element with the *"accesskey"* attribute. Access keys are meant to mitigate some of the issues of the tabbing approach to keyboard access, by assigning single key shortcuts to elements on a page (usually the main navigation links). For example, a link labelled *Home* might be assigned the access key *H*, while *Calendar* could be *C*. Regardless of their position in the tab chain, those links can then be

accessed by a keyboard user by typing that shortcut.

Access keys, while still a part of the HTML standard, present multiple problems for real implementations:

1. Required modified keys. While the access key itself is a single letter, browsers require a modifier key to be pressed to activate them (e.g., Alt + Shift in Firefox, Alt in Internet Explorer, Ctrl + Opt in Safari). This increases the number of key presses required per activation, and requires the user to press multiple keys at once, which is difficult for many users with motor impairments (without using additional accessibility options to allow multi-key combinations).

2. Inconsistent implementations. For example, one site might use *H* as the shortcut for *Home*, while another might assign it the number *1*. Users must remember access keys for each site, which would quickly become infeasible if they were more widely implemented. In a study of 152 Irish Web sites, Trulock found that 82% did not implement access keys at all [56].

3. Lack of discoverability. There is no standard way to indicate the presence of access keys on a regular Web browser, such that users must have pre-existing knowledge of the access keys for a Web site before they can be used. For this reason, some sites have separate pages listing their access keys, and some governments and other organizations have attempted to standardize the use of access keys on their Web sites [19].

4. Keyboard shortcut conflicts. Screen readers and other keyboard navigation software reserve a lot of keyboard shortcuts to allow the user to efficiently navigate a page. Shortcuts defined by a Web page via access keys can interfere with these application shortcuts and cause conflicts [59].

5. Not a general solution. Access keys can only improve keyboard access for the links with the *accesskey* attribute. They are not designed to be implemented on every link on the page.

Due to these issues, some authors are recommending against using access keys on the Web [60] [10]. The World Wide Web Consortium (W3C) have also deprecated the accesskey attribute in version 2.0 of the XHTML standard [63].

2.1.4 ID Navigation

Amongst alternative keyboard navigation systems, ID navigation is a relatively common technique based on automatically assigned keyboard shortcuts for elements. The Firefox extensions Hit-a-Hint [24] and Conkeror [17] are examples of such a technique. The Konqueror [33] Web browser on the Linux platform also implements a similar access method. With these systems, the user can press a key to bring up a small numbered label beside every clickable element on the page. By typing in the number code of the desired element and pressing Enter, the system "clicks" on that element, thus following the link or activating the button.

While these types of number code systems are quite effective, we believe the design imposes some limits on efficiency and adds an extra cognitive step to the navigation process. If the shortcut labels are not shown automatically after a page is loaded, an extra key press is required to show them; while if they are shown, the extra overlays cause visual clutter and occlusion problems when there are many elements on a page (See Figure 2.3), as well as disturbing the aesthetics of the Web page. In addition, since shortcuts are assigned sequentially to the on-screen elements, there is no semantic connection between the keyboard shortcut and the clickable element (for instance, the shortcut for a link labelled "Sports News" may be 42). A user must bring up the labels, make the connection between the numeric shortcut and the element, and then type in the code. These systems also do not take the relative importance of clickable elements into consideration, such that main navigation links could have longer shortcuts than less important links.

Figure 2.3: Shortcuts in Konqueror showing visual clutter and page occlusion.

2.1.5 Search-Based Navigation

Search-based navigation refers to a set of techniques in which the user selects links by starting to type the text of the link. Mozilla Firefox implements search-based navigation with a feature called “find as you type”, which searches all links on a Web page after every typed character and focuses the first matching link on the page [38]. The match is continuously updated as more characters are typed. This allows the user to select text links, but does not allow focusing of image links, text entry boxes, clicking of form buttons, or selecting other form input fields. A similar technique is also implemented in the Gleebox [2] extension for the Google Chrome browser, which additionally allows the selection of form fields and image links. While existing search-based navigation techniques represent a marked improvement over simple tabbing, based on personal experience using these systems we believe that the matched links often do not correspond well to what a user would expect.

The core reason for this discrepancy in match results and user expectations is that the link selection algorithms search with a simple substring matching algorithm: starting from the top of the document and stopping at the first occurrence of the query. This is the simplest and fastest search algorithm to implement, and thus is widely used. We call this approach *computer-centric*, as it is the easiest to implement using standard search routines, rather than being adapted to the user, or *user-centric*. For example, if a user starts a search near the bottom of a page, the likelihood of a match earlier on the page is high. If such a match is found, the browser will scroll up to that match, resulting in a jarring transition for the user and very likely a match that was not desired, since a user is unlikely to want to select an element he/she cannot see. In addition, since the location of the matched substring is not taken into account, matches are often found somewhere inside a word of link text, resulting in matches that do not necessarily correspond well with the user’s expectations. Figure 2.4 shows how a user typing the three letters ’nat’ ends up with a link labelled International instead of the expected National Science Foundation.

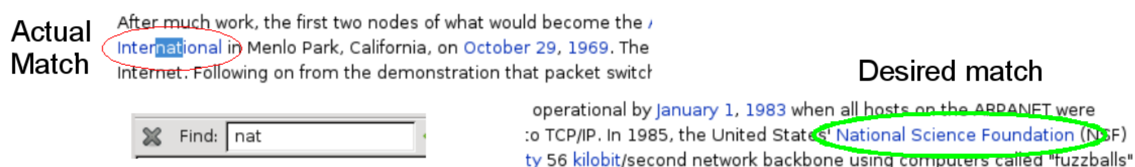

Figure 2.4: “Find as you type” showing unintuitive matching behaviour.

Search-based navigation techniques are primarily designed for proficient keyboard users, and thus have not placed as much importance on reducing the absolute number of keystrokes necessary per element activation. For the average skilled typist, additional keys have very little time cost, and typing several keys to ensure the correct link is selected is often faster than stopping after fewer keys to determine whether the desired link has been selected and possibly having to type more. For users with motor disabilities, where every key represents significant effort and time, absolute keystroke cost becomes more important.

2.2 Switch Input

People with more severe physical disabilities may not be able to control a mouse or a keyboard equivalent device at all. For these users, one or more hardware switches, activated by any body part capable of consistent, voluntary movement, may be used along with specialized software to control the computer. Many devices and systems have been developed to allow a small voluntary action to activate a switch, whether it is by changing air pressure (sip & puff switch), by measuring muscle signals, or triggering on small movements recorded with a camera. For people with severe disabilities, a single switch – although it is a low bandwidth input – is often the only reliable method of environmental control. Given that computers are designed to be operated with either a keyboard or a mouse equivalent device, the outputs of these switches must be transformed into one of these two types of inputs in order to control standard computer software.

The most common solution for typing using a single switch is the row-column scanning keyboards or variations thereof, which first appeared in research over thirty years ago [42]. These keyboards place the letters of the alphabet on a grid, which the user can then select using a combination of switch activations and delay timeouts to scan to their desired row and column. Each row is highlighted starting from the topmost until a switch is activated, then columns are highlighted starting from the leftmost until another switch activation selects a letter [52] (Figure 2.5). The most efficient implementations use a divide and conquer approach by dividing the available characters into groups and narrowing down the selection on each switch activation. These types of typing methods are commonly found in commercial switch scanning products [35].

Using this kind of layout, the letters in the top left boxes are the fastest to select,

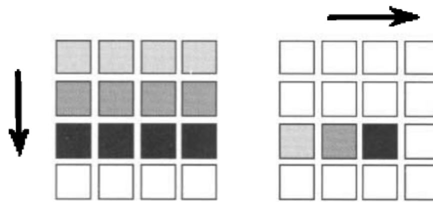

Figure 2.5: Row-column scanning keyboard (from Shein [46]).

and selection times increase towards the bottom and right. Many alternate letter layouts have been proposed to increase typing efficiency for the user's language or typing context. Figure 2.6 shows a common layout based on the frequency of the letters in the English language.

Sp	E	A	R	D	F	V
T	O	N	L	G	K	J
L	S	U	Y	B	X	Z
H	C	P	Q	1	2	3
M	W	,	“	4	5	6
Up	.	-	?	7	8	9
Del	Ent	;	‘	0	(	)

Figure 2.6: High frequency layout for a row-column scanning keyboard.

The primary method for navigating the World Wide Web has traditionally been by “point and click”. This method is not easily adaptable to binary input devices. One approach to solving this problem is to transform the binary switch input to a two dimensional pointer output by way of an intermediate system. For example, some systems use an on-screen set of arrow buttons to move the mouse pointer [47]. Focus scans through the arrow buttons until the switch is activated, at which point the mouse cursor will move in the specified direction either by a fixed amount or continuously until another switch activation. Instead of these buttons, some other systems continuously rotate a line centered on the cursor position to establish the direction of movement, and then start the cursor moving in the specified direction when the switch is activated. While these methods are a valuable fallback for applications that are not accessible via the keyboard, it is not an efficient interface for continuous use and requires precise timing control in switch activations from the user.

There are three notable Web navigation systems specifically designed to be accessible to single switch users: the AVANTI browser [51], the MultiWeb browser [41], and the Hawking toolbar extension for Mozilla Firefox [7]. All of these approaches use a linear scanning approach to link selection, with some differences in how the link scanning is presented to the user. In addition, Mankoff et al. developed a Web browser accessible to users capable of operating four separate switches using a similar linear scanning approach, with focus being advanced by switch activation instead of automatically on a timeout [36]. Hanson et al. with their accessibilityWorks project, while focusing more on adapting the appearance of Web pages, also consider the issue of control by devices with zero or one dimensional input [26]. Similar to other work in the area, they propose to extract the list of links and place them in an external list such that they can be selected by a user incapable of two dimensional cursor control. The Gnome On Screen Keyboard [21] implements a similar link extraction scheme for Web browsers that can expose Web page structure to an accessibility framework [65].

There are two general approaches to scanning through selectable elements: in-place and external scanning. In-place scanning operates the same way as tabbing in a regular browser by scanning through the list of selectable elements on the page and highlighting them within the original Web page layout. The external scanning approach works by extracting all the selectable elements in the page and displaying them in a list beside the Web page. This list is then scanned through and a link can be navigated to by activating the switch when the target is highlighted.

While the linear scanning approach is a logical extension of the scanning used in switch typing interfaces, it has some key disadvantages that limit its efficacy for navigating the Web. In-place scanning has the advantage of lending context to links with non-descriptive names. Although it is considered bad practice for Web accessibility, many hypertext links are given names such as "click here", or "read more" which are only meaningful in the context of the surrounding text. If these types of links are extracted from the Web page and scanned through in an external list, it becomes very difficult to determine which link is the desired target.

In-place scanning preserves the element context, but, similar to tabbing, comes with the disadvantage that the scanning order can be unpredictable. When using a single switch and automatic scanning, the user must quickly activate the switch when the target element is selected. This makes it even more important that the scanning order be predictable, in that the next link that will be highlighted should be known

to the user before it is reached. Techniques to decrease scanning times, like scanning ahead by several elements at once can compound the problem of unpredictability.

The primary disadvantage of both linear scanning approaches is their lack of scalability to Web pages with many selectable elements. It is very common for Web pages, especially those of news networks, search engine results pages, or site maps, to contain many tens or even hundreds of links. Linearly scanning through large numbers of elements to find a desired target is a very tedious process. The external scanning approach gives more flexibility in grouping the list of links and selecting them via a divide and conquer approach to improve performance, but is still difficult to present to the user in an intuitive way for large numbers of links. The error cost of the linear scanning approach is also high, as a missed target would, in the best case, involve having to wait for the scan to loop around, and in the worst case result in a click on an incorrect link, necessitating multiple steps to return to the previous page.

It should be noted that all of the switch scanning browsers or browsing extensions mentioned here are currently unmaintained, and either no longer operational or dependent on browsing engines that are several years out of date.

Chapter 3

KeySurf

As discussed in Section 2.1, tabbing is not efficient enough to provide practical keyboard access to the Web for users with low rate input, while accessibility features in the HTML standards do not provide a general solution to accessing all Web page elements. Advanced keyboard navigation systems like ID and search-based navigation are more efficient, but opportunities for improvements remain. In an attempt to address the shortcomings of the advanced keyboard navigation systems, KeySurf is developed based on the following design principles: efficient keyboard access, simple user interface, shallow learning curve, minimal Web page interference, and maximum compatibility with existing Web sites.

The goal of selection efficiency is naturally very important for users where every input device activation requires significant effort. While efficiency depends partially on the active elements of the Web page, KeySurf includes several features to reduce the number of required keystrokes necessary to activate an element compared with other search-based methods. However, in some cases absolute selection efficiency is traded for a more consistent and intuitive interface.

The graphical user interface of KeySurf is kept as minimal as possible. Some additional user interface elements are necessary to convey information about the current query to the user. However, a keyboard driven browser like KeySurf no longer requires the toolbar buttons (such as back, forward, home) normally visible in Web browsers, so the overall screen space dedicated to browser navigation interface is approximately equal. KeySurf provides a status bar to inform the user about its current state, and highlights matching elements inline in the page.

The learning curve of KeySurf should be as shallow as possible. This is accomplished by making the central theme of “type where you want to go” sufficient to

operate the KeySurf interface. Additional shortcuts are available to improve selection speed, but are not necessary for the majority of elements. The major downside to traditional keyboard access features (like access keys) is the lack of discoverability and the requirement for the user to memorize a set of keyboard shortcuts or commands before being able to operate the interface.

Using KeySurf, the visual appearance of Web pages should not be altered if at all possible, such that users of character-based input devices can access the same experience as users of an unmodified browser. One of the approaches to improving the accessibility of Web pages is to route them through a proxy which transforms inaccessible aspects of their design (for example, small fonts) to more suitable alternatives, or adds additional controls for switch users [36, 53]. While this improves accessibility, these kinds of automated transformations often change the design of the page and can compromise its aesthetics or functionality.

Web site compatibility is a critical measure of success for any alternative browsing method. With the majority of Web sites at best only partially conforming to accessibility standards, any alternative Web navigation cannot solely rely on those standards while expecting to provide a Web experience equal to that enjoyed by mouse users. KeySurf requires only that elements be focusable via a keyboard interface, and does not require any other special consideration by Web authors.

Materials presented in this Chapter and the next have been published in [49] (acceptance rate 44%), [48] (poster paper), and [50] (acceptance rate 10.4%).

3.1 Target Users

The KeySurf system is primarily designed for users with a physical disability who have trouble accurately using a pointing device (such as a mouse, joystick, or track ball) or are not able to use a pointing device at all. More generally, the system is suitable for any user who can type two or three characters (with a keyboard or equivalent input device) faster than they could acquire a small target (such as a hypertext link or form button) with a pointing device. For convenience, we refer to the process of entering characters into the KeySurf interface as *typing* in this work. However, KeySurf can be controlled by any input device that can function as a keyboard interface. This interface could be a regular computer keyboard, Morse code from a single switch device, any specialized hardware or software for text input, or a speech-based interface as further discussed in Section 3.9.1.

Naturally, KeySurf also has the potential to be useful to non-disabled users who are faster or more comfortable with keyboard input than mouse. With the proliferation of mobile computing devices like laptops, and the increasing availability of public wireless access points, computer users are more and more likely to be accessing the World Wide Web away from their home or work computer stations. As there is often no room to use a standard computer mouse when using these devices away from a desk, users must rely on the pointer control interfaces incorporated into these devices. On laptops, these pointing interfaces are usually small touchpads or isometric joysticks embedded in the keyboard. These pointing interfaces usually compromise either speed or accuracy compared to an external mouse, which can make a keyboard controlled Web browsing system more practical.

Although there is potential for non-disabled people to benefit from KeySurf, our research is focussed on its applicability to persons with physical disabilities, and we disregard most issues that would be specific to the non-disabled. In the majority of cases, the same concepts apply to both classes of users and the primary differentiating factor is the typing speed that a user is capable of. Certain aspects of KeySurf will be most useful to users that are limited to low-rate character input, and less so for non-disabled users capable of rapid input. This will be clarified when these features are discussed in this work.

3.2 System Design

The relationships among the major components of the KeySurf system are depicted in Figure 3.1. KeySurf is independent of the input device in the sense that it can be controlled by any input device that is capable of acting as a keyboard interface (i.e., it can produce character output). The link selection and highlighting mechanism relies on Web page elements being extracted, assigned a relative priority for the current page, and unlabelled elements being given a label before selection can be commenced.

As pages are loaded, the browsing history component keeps track of user browsing activity such as page viewing time and key terms from Web pages to form implicit indicators of a user's interests (Section 3.8.1). More explicit indicators of user interest such as a user's search terms are taken into account as well (Section 3.8.3). These history-based indicators, along with the User Centric Search algorithm (Section 3.4), influence the priority of elements on the Web page. Once a character has been entered into the browser, the highlighter module finds elements that match the query and

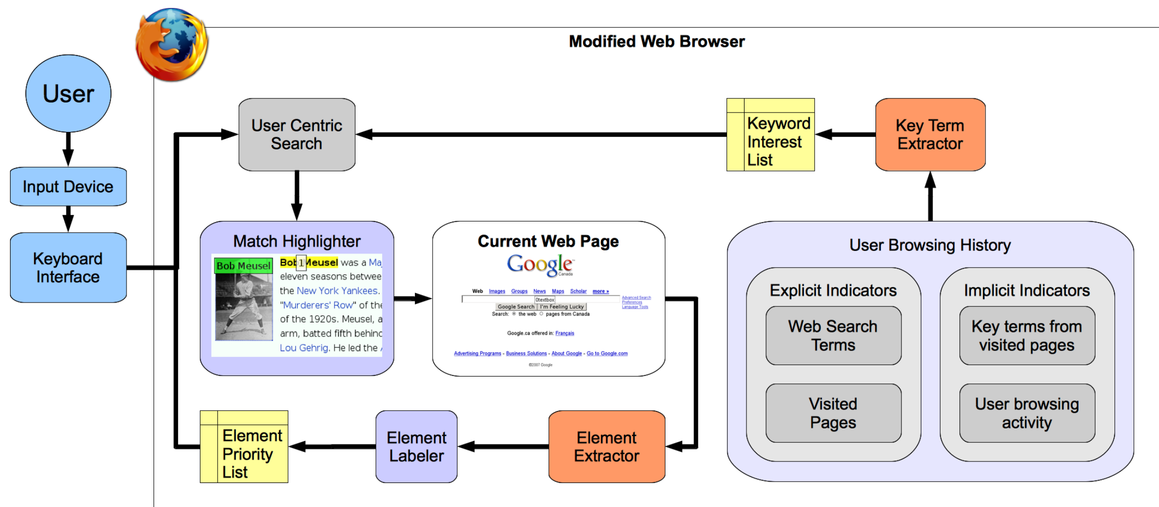

Figure 3.1: KeySurf system overview.

orders them by their relative priority. The element with the highest priority is marked with, by default, a green highlight, while a yellow highlight is used for lower priority matches.

3.3 Element Labelling

To select an element using a search-based navigation system like KeySurf, a textual label must be generated for every active element on the page. For some elements like regular hypertext links, we are provided with a logical label in the link's text content, while for others like search input boxes, we must generate a label within KeySurf. The primary goal of the element labeller is to minimize the number of labels that are not defined from the page content, since the labels provided by the page author will be much more descriptive of elements than any computer generated labels. For elements where a generated label is unavoidable, labels should be chosen to minimize the number of characters necessary to select their elements. A secondary goal is that the system should make as few modifications to the visual appearance and layout of the Web page as possible.

Initially, the system extracts the text of every active element that has been associated with text on the page by the page author (pre-labelled elements). This includes text links (where the label is the link text), form buttons (where the label is the button text), form elements with an associated `<label>` tag, image links with alternate

text, and elements with a script-defined click action (where the label is the inner text of the element). For elements without a predefined user visible label (unlabelled elements), the system must generate one. Such elements could be text input boxes, image links, or other elements that do not have associated text. The label generation algorithm is discussed further in Section 3.3.1

3.3.1 Label Generation

Since KeySurf has complete control of what characters to use for unlabelled elements, one approach to maximizing the selection efficiency of elements is to prefix generated labels with the alphanumeric characters that are not already used as starting characters for pre-labelled elements. This would ensure that the elements with generated labels would always be selectable with a single keystroke.

However, in practice this leads to unpredictable element labels on repeated visits to a page with dynamic content. For example, the popular search portal pages (Google Personalized Homepages, Yahoo!, MSN, etc) exhibit this behaviour, since they contain mostly dynamic content (news headlines and customized content) with one search text field. Since the search field is likely to be a common selection target for users, its label should be predictable every time the user loads the page, rather than possibly changing as page content changes.

To overcome this problem, the label of text input fields always takes the form {*0,1,2,3...n*}*textbox* regardless of page content. Regular text links rarely begin with numerals, making this strategy as efficient as using unused characters in most cases. Unlabelled image links are simply labelled with a number, to minimize the size of the overlaid label element and therefore the occlusion of the image by the label.

It should be noted that simple numbered labels are only constructed if no meaningful information about the element can be determined which could be used instead. For form input elements (such as check boxes, text boxes, or radio buttons), KeySurf first checks if the form contains an HTML `<label>` tag that provides a text label for the element. If a `<label>` tag exists, it is also used as the KeySurf label for that element to avoid having to generate a numbered label. If the label tag does not exist for drop down (combo) selectors, the selected text is used instead, while a numbered shortcut is used for other elements. Labelling for image links is discussed in more detail in Section 3.3.2.

Figure 3.2 shows an example of various HTML form elements and their associated

labels in KeySurf. The text field in this form does not have an associated `<label>` tag, so KeySurf generates a numbered label for it and places it over the element as an overlay. The checkbox element is correctly associated with a `<label>`, so its KeySurf label is identical to the displayed text. The drop down has no associated text, so its label is the selected text, while the label for the button is the text inside it. As much as possible, KeySurf tries to use labels that allow users to "type where they want to go".

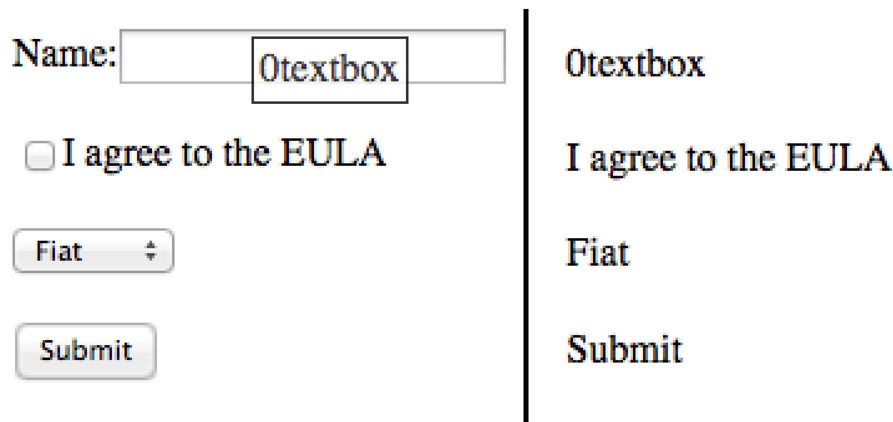

Figure 3.2: HTML form elements and their associated KeySurf labels.

The trade-off to consider when constructing labels for text links is whether to provide as much coverage of the alphanumeric character set as possible versus leaving the labels as they appear on the page. When multiple links start with the same letters, typing the first letter of a desired link selects all the links instead of just one. For example, if searching only from the start of a link, a user would have to type 8 letters to uniquely select one of the "Google" links in Figure 3.3. One solution to this problem is to perform aggressive relabelling on the conflicting link texts, by prepending a character that is not used in other elements. This ensures maximum efficiency but, like the ID navigation techniques, necessitates layout modifications and reduces the intuitiveness of the labels for links.

In our implementation, we have chosen not to interfere with the labels of any elements that have been given a label by the page author. Through informal user testing, aggressive relabelling added to visual clutter, while users expressed confusion as to why certain links appeared "broken" (with a modified label) while others were not. For elements that would not be selectable with two keystrokes using the user centric search algorithms (Section 3.4), we have provided more predictable selection

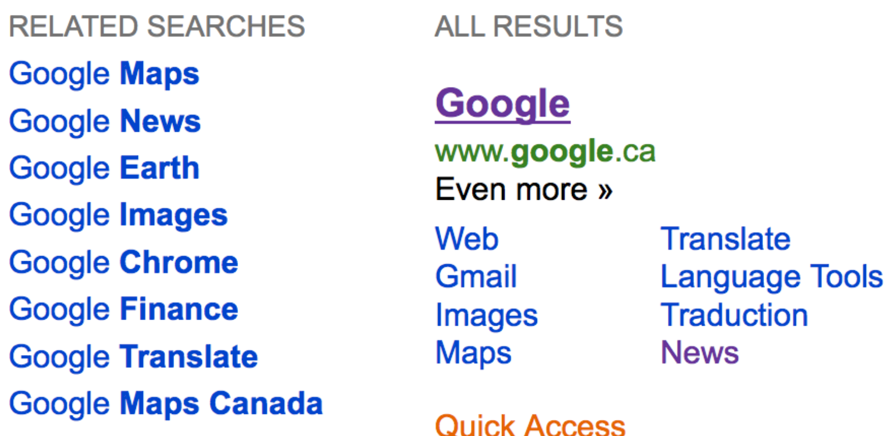

Figure 3.3: A Web page showing multiple links starting with the same letters.

shortcuts as discussed in Section 3.5.

3.3.2 Image Links

Image links on Web pages are anchor (`<A>`) tags whose sole child tag is an image tag. If a link contains text content along with an image, it is treated as a regular text link and the image is ignored. Image links are primarily used in three distinct ways on Web pages: thumbnails that link to a larger image, site navigation buttons that contain rasterized text, or image links that provide an additional click target for a text link. The former two categories of image links raise problems during labelling that require special consideration.

For images acting as site navigation buttons, overlaying a label is a potential source of confusion on poorly written Web sites. In the best case, these images will have alternate text specified which is identical to the text on the image. While this will cause duplicate labels for a single element, the overlaid label will agree with the text on the button itself (Figure 3.4). If there is no alternate text specified, or worse if the alternate text is different than that on the button, the basic tenant of "type where you want to go" is compromised, as now users must type a different label than the button text. Figure 3.5 shows image buttons without alternate text that exhibit this problem. A potential future solution to this issue is to develop an optical character recognition module that would attempt to recognize the text on image buttons for use as the KeySurf label.

When image links are used to provide a larger click target area for an existing

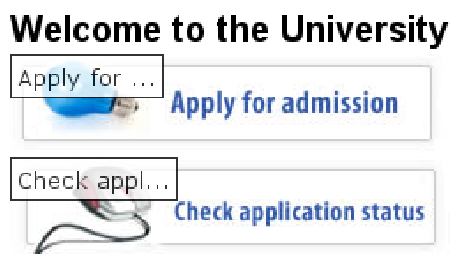

Figure 3.4: Image buttons with identical alternate text in KeySurf.

Figure 3.5: Numbered overlays for image buttons without alternate text.

text link, they can be safely ignored by KeySurf. This is often the case when a link is decorated with an image preview of the linked content as shown in Figure 3.6. In this case, both the image and the text below it are separate links pointing to the same target page. With a simple approach to image labelling, the image link would be labelled with an overlay, visually cluttering the page with unnecessary overlays. KeySurf detects that the image is linked to the same page as a nearby text link and does not generate a label for the image. Although this approach avoids unnecessary overlays, if the identical text link is not close to the image, the lack of an overlay on the image can be confusing to users as it is not clear which text link is linked to the same target as the image. To mitigate this problem, overlays on images are only displayed if either no same-target text link exists, or if the pixel distance to the identical text link is greater than a threshold.

3.4 User Centric Search

The concept of using incremental search to select links on a page is not new. As discussed in Section 2.1.5, this algorithm is implemented in the Mozilla family of

Figure 3.6: Text and image link with identical targets.

browsers under the name of "find as you type", but has several important limitations. The goal of our matching algorithm in KeySurf is to improve on the incremental search algorithm by taking a user centric approach to matching and ranking results, instead of the computer centric approach of a regular linear search algorithm.

To accomplish this, we have implemented a series of constraints on the incremental search algorithm to more accurately match a user's expectations of which link should be selected for a given query. These constraints are best summarized as a set of relations between which types of matches are prioritized over others. Note that all the following examples represent possible matches given that the user has typed an upper case "S".

1. Currently visible elements before off-screen ones. In the vast majority of cases, users will want to select elements that are currently visible, rather than elements that are off-screen. This reduces the number of elements that are searched in the common case, while still allowing off-screen elements to be selected by advanced users.

2. Same case matches before case insensitive. We assume the upper case letter was deliberately chosen, since upper case letters require additional effort to produce with most input devices.

   ex. Sybase before systematic.

3. Starting characters of element labels before others. Left to right reading order in the English language suggests that the beginning of the element text is the most logical location to start typing letters when wanting to select an element.

ex. Sports News before Download SDK.

4. Starting characters of words before other substrings. Word boundaries present a more logical starting point to begin typing than positions within words.

   ex. Download SDK before Downloads.

5. Visually prominent elements before subtle ones. Web authors often make semantically important elements (such as headlines) more visually prominent than less important ones.

   ex. Search before Sports.

6. Elements that correspond to user interest before items that do not. By estimating from browsing history what topics the user is interested in, elements that match these topics can be prioritized over elements that do not (Section 3.8.1).

The first four constraints are implemented by restricting the active search field, that is the set of elements that are selectable with the current query, while the last two constraints are used to rank matches by their probability of being activated by the user. The highest ranked match is given a green highlight along with the keyboard focus, which allows the user to immediately activate it with the enter key.

The currently active search field specifies which subset of the active elements on a page are currently being matched against the user's input. Most keyboard navigation systems operate on a fixed search field of the entire Web page. For example in tabbing, each link on the page is traversed, whether it is initially visible to the user or not. While this is simple to implement, it does not respect the fact that the probability of a user selecting any given element is not uniform. It is far more likely for a user to want to select a visible link, than one that is currently off-screen, since selecting an off-screen link would require pre-existing knowledge of the page content. Thus, one natural search optimization is to search currently visible elements before other elements on the page (Figure 3.7).

Some other keyboard navigation systems, like the Konqueror browser, use a fixed search field of only the visible elements. While this approach improves efficiency for the common case by reducing the number of digits in element IDs, it prevents expert users from selecting an off-screen link of which they may have prior knowledge. Since scrolling down a long page requires many user inputs, it would be beneficial for a system to allow selecting any link without having to scroll it into view first. The

Figure 3.7: The set of visible elements (1), and all elements on a page (2).

goal of dynamically setting the search field within the set of all elements on a page is to reduce the number of highlighted elements after every keystroke to only the most likely results, yet still allow any element on a page to be selectable. Figure 3.8 shows the overview of the flow in the user centric search algorithm.

The step labelled "Widen Search Field" requires some additional explanation. Initially the restrictions on the search field are set at their strongest to limit results only to visible elements whose labels begin with the query. Thus, if a Web page is taller than the height of the browser view, the algorithm will only match elements that are currently visible to the user. If characters are entered and there are no results with those restrictions, they are relaxed step by step until results are available. If no visible elements start with the query, KeySurf considers visible elements in which

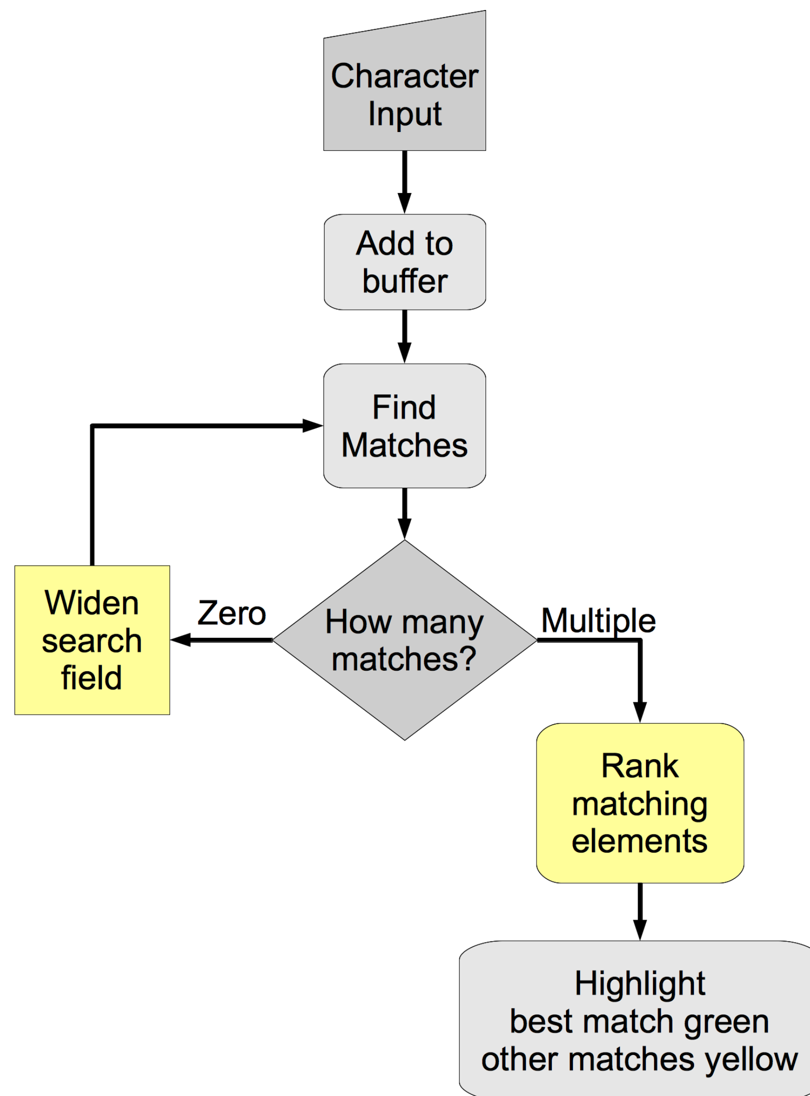

Figure 3.8: The user centric search process.

any words start with the query. If there are still no matches, off-screen elements starting with the query are added to the search field, followed by off-screen elements where any word starts with the query. In the vast majority of cases, the user will be attempting to select an element that is currently visible to them. This search fallback method increases the efficiency for the common case of a user wishing to select a visible element, while still allowing a user to select an off-screen target if he/she has existing knowledge of its label.

3.5 Element Selection Shortcuts

Although the basic user centric search algorithm usually results in few matching elements after only one key stroke (See Section 5.4 for empirical data on numbers of matching elements), certain collections of elements can create situations where selecting some elements can be very inefficient or even impossible. As previously discussed, searching for "Google" may yield results like ""Google Maps", "Google News", "Google Images", and so on (Figure 3.3). While one of these links will be selected after typing "g", the others would require 8 keystrokes to select ("Google " and one more letter), which is clearly not acceptable for users with slow keyboard input. Although users could take advantage of constraint 4 of the user centric search algorithm and start typing at the second word, this may also be inefficient if another link on the page starts with those letters (e.g., a link named "News" would be selected before "Google News" if a user typed "n").

Another relatively common occurrence in Web pages that prevents the selection of some elements is that of multiple different elements with the same labels. If there are two or more pre-labelled elements with identical labels, a search will return all these elements even if the user types the whole label. These situations occur most often for elements whose meaning is defined by their location or proximity to other elements, rather than their text alone. For example, the "Reply" and "Read More" links in Figure 3.9 take their meaning from their placement beside the message text. Although this practice is discouraged in accessibility standards due to its detrimental effect on accessibility for visually impaired users (using screen reading software), it is still commonly seen on the Web, and any usable Web navigation system must be able to select these links. Without additional consideration, the user centric search algorithm cannot uniquely select multiple links with identical labels.

If an element cannot be uniquely selected by typing two characters, we provide two shortcut methods in order to reduce the number of required keystrokes. After typing the first character, all non-default (yellow) matches are examined to determine if they could be uniquely selected by typing the next character in their label. If an element cannot be uniquely selected (i.e., there would still be multiple matches after two keystrokes), it is assigned a numbered shortcut. The shortcut is shown with a translucent overlay as seen in Figure 3.10. In this figure, the first link is the default, and the second link can be selected by typing the next key in its label ("y"). The third link starts with the same five letters as the first, so it is assigned the shortcut

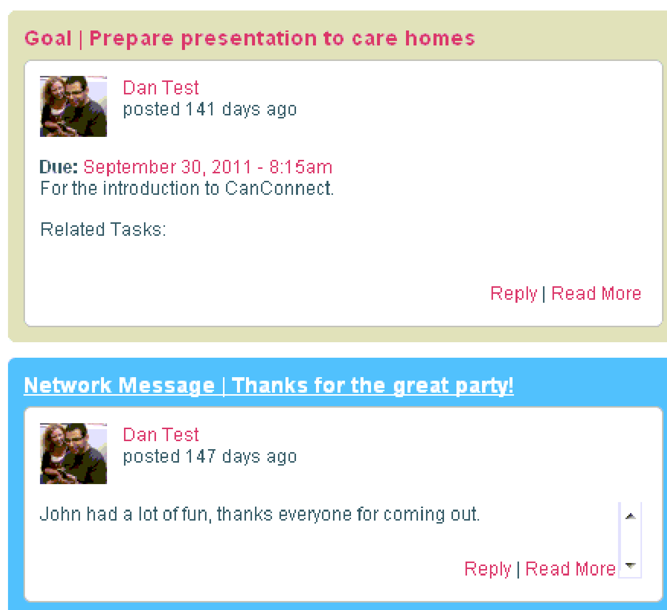

Figure 3.9: Web page showing multiple links with the same text.

of “1”. This makes it selectable with 2 keystrokes (“h1”) instead of 6 (“homini”).

- Superfamily Cercopithecoidea
  - Family Cercopithecidae: Old
- Superfamily Hominoidea: apes
  - Family Hylobatidae: gibbons ("les
  - Family Ho1nidae: great apes inc

Figure 3.10: Highlighted elements after typing “h”.

The arrow keys to move the default focus within matches was added as an alternative to the numbered shortcuts, if the desired match is close spatially to the default item. For example, in Figure 3.10, pressing the down arrow would move focus to the second link, while pressing the up arrow would wrap focus to the third link. While the arrow shortcuts do not add efficiency, they provide an alternative to selecting non-default matches that doesn’t require reading the numbered shortcuts.

These shortcuts were added primarily for users with low rate keyboard input, as it establishes an upper bound of two keystrokes to select any element (providing that the visible section of the page does not contain more than 11 elements with the same first two characters, which is a valid assumption in most cases). Users capable of faster character input retain the option of typing just the letters of their desired

element, which requires less visual feedback (no processing of number overlays) at the cost of some extra keystrokes.

3.6 Reducing Error Rate

In addition to a reduced keyboard input rate, persons with motor disabilities are also prone to a higher error rate when keyboarding. Trewin and Pain identify six keyboarding errors common for persons with motor disabilities, four of which result in unintended characters being typed [55]. The most common errors arose from holding a key too long resulting in duplicated characters, and pressing a key adjacent to the target key. In Trewin and Pain's study of 20 individuals with motor disabilities, correcting these errors added significantly to the typing time, with increases ranging from 4% to 49% depending on the user. Thus, it is important to prevent typing errors to reduce the impact of correcting them.

In KeySurf, input is filtered by the available elements on the screen. If an input would result in zero matching elements (after the relaxation of all search field constraints), it is ignored. This saves the user from having to correct some keyboarding errors. The percentage of keyboarding errors that can be prevented depends on the labels of the elements on screen. For the first key stroke, there is little effect, since it is highly likely that at least one element will match any given character input, even if it is not the desired element. For the second character, it is much more likely that a key press in error would result in zero matches, and thus be rejected by KeySurf. Our own experimental data from real Web browsing sessions shows that the median number of matches after one key press is only three (see Section 5.4.2), which indicates that most accidental characters after the first would be filtered out by KeySurf (since they would not match any elements on the page).

3.7 Accessing Bookmarks

For the most part, the major Web browsers provide adequate keyboard shortcuts for common Web browsing actions (e.g., back, forward, stop, bookmark page, etc). For these common actions, no additional support is necessary to facilitate use by people with disabilities. Accessing bookmarks, however, is usually quite cumbersome, with users having to iterate through each bookmark entry in the menu to locate their desired bookmark (with a bidirectional menu containing a single level list of n

bookmarks, activating a bookmark requires on average of $n/4+3$ keystrokes [1]. Thus, just like tabbing through links, accessing even a moderate number of bookmarks can be slow.

To address this problem, we have extended our link selection process to also select from bookmarks. Since bookmarks can be represented as simple links, the same selection process applies as to that for Web pages. If a user activates the "Bookmarks Mode" (currently via the period key), the system renders all of the user's bookmarks on a single page, grouped by folder. Individual bookmarks can then be selected by typing letters as normal. Providing that all bookmarks can be rendered on a single page, this system allows any bookmark to be activated with four or fewer keys (one key to show the bookmarks, one or two keys to select a link, and one to activate it).

Bookmarks

Get Bookmark Add-ons

Wikipedia
Qt 4.3 with Visual Studio 2008 | PcManiac
ecsflyer.pdf (application/pdf Object)
Contact Handiramp for all Ramp needs
Tips and tools for helping people with special needs
Contact-Us

Mozilla Firefox	Keyboard Navigation
Help and Tutorials	On the Net, Web Browser Keyboard and Navigation Shortcuts
Customize Firefox	Mozilla/Gecko Keyboard Navigation Proposal
Get Involved	On the efficiency of keyboard navigation in Web sites
About Us	WebAIM: Keyboard Accessibility

Figure 3.11: The bookmarks selection screen in KeySurf.

3.8 Using Web Browsing History

To improve our estimation of user interest, we go beyond the physical characteristics of matches discussed in Section 3.4 (visual prominence, match location, visibility,

[1] Bookmark access starts with a keyboard shortcut to display the bookmark menu (Alt+B), followed by the arrow keys to navigate down the list. In a menu that allows bidirectional navigation the user can press the up arrow to jump to the bottom of the list. Once the desired bookmark is located, the user presses enter to activate it

etc) by taking into account the user's browsing history. To determine what may be of interest to the user, we examine various aspects of a user's browsing history, such as pages visited, page activity, and topics searched. These interest indicators are discussed in detail in the following sections.

The most direct approach to taking advantage of user history is by directly prioritizing previously visited pages. Web browsing behaviour generally exhibits a certain degree of locality, and if a user has visited a page in the past it is likely an indication of interest in that page. When selecting a link, KeySurf checks each matched element against the user's browsing history. If the exact target page of a link has been previously visited, that link is strongly prioritized. Currently a visited link takes the highest priority, and it will be selected as the default match over other non-visited links.

If the user has previously visited another page on the target domain of a link, that link is weakly prioritized. For example, if a user has visited the page with the URL *http://coolstuff.org/monkeys.htm* and then visits another page containing a link to *http://coolstuff.org*, that link will be prioritized over other links that have no connection to the user's history.

The prioritization of visited pages allows a user to navigate to links they have already visited more quickly. While a link may require two keystrokes to select the first time, the priority boost from being visited is likely to make it the default match after only one keystroke when it is selected again. However, the majority of links that users select are not previously visited (See Section 5.4), and thus cannot benefit from direct history prioritization. Section 3.8.1 discusses how we use Web page keywords to predict user interest in unvisited links.

3.8.1 Inferring User Interest

As other authors have noted, while it would be more accurate to explicitly query the user about their degree of interest in visited pages, or a topic in general, the rating process is too intrusive and time consuming for most users [20]. Especially when considering the target users — for whom any selection requires significant effort — explicit ratings are not feasible. To address this problem, there has been significant prior work in the area of implicitly determining user interest based on a user's browsing history. Previous approaches have attempted to find a correlation between a user's interests and various factors such as page viewing time, scrolling time, mouse

clicks, and related activity (bookmarking, printing). In a user study comparing explicit ratings with these observed factors, Claypool et al. concluded that time spent viewing a page and the amount of scrolling provide the best indicators of interest in that page [15].

Page interest scores are calculated based on the maximum amount of time spent viewing a page in the user's Web history and the percentage of the page that was viewed. Web page viewing times do not accumulate if a user revisits the same page at a later date. This prevents often loaded pages (such as the user's home page) from dominating the page viewing times, even though each visit may only last a few seconds. Additionally, pages that are visited repeatedly often contain very dynamic content (news sites present a prime example), such that key terms may change at every visit, and the page viewing time for the previous visit cannot be meaningfully added to the current viewing time.

We define page viewing time as the time elapsed between when a page is rendered by the browser and the time just before the page is hidden from view (either by navigating to a new page or closing the browser). By only measuring time after a page is rendered, we exclude the variable loading time due to network congestion and page size. To detect when a user is actively viewing a page (versus being distracted by external events), we keep track of input activity (mouse movements, key presses, and mouse clicks) while a Web page is open. If there are no input events for 2 minutes, the page timer is paused until another input event occurs. We normalize page viewing times by defining the viewing time of 20 minutes as maximum interest, and scaling times to this maximum to obtain a view score (S_V) between 0 and 1.

To calculate page score (S_P), we combine the view score with the percentage of the page that was covered by scrolling as shown in Equation 3.1. The scroll modifier is calculated by dividing the height of the page viewed by the user (H_S) by the total page height (H_P). That is, if the user scrolled halfway down the page the view score is reduced by half to find the page score.

$$S_P = S_V \left(\frac{H_S}{H_P} \right). \tag{3.1}$$

The page score provides a measure of interest for visited pages. However, since the majority of link clicks are to previously unvisited pages, we must expand our estimation of user interest to include more general topics or keywords of interest. These keywords can then be used to calculate user interest scores for links leading to

unvisited pages. The algorithm for extracting this information from visited pages is described in Section 3.8.2.

3.8.2 Web Page Keyword Ranking

In order to turn our page interest score into a keyword interest score, we execute a keyword extraction and scoring algorithm on every Web page visited by the user. The module is triggered after the page has completely finished loading. The algorithm steps are as follows:

1. Web page finishes loading and is displayed to the user.

2. System begins recording page viewing time (See Section 3.8.1).

3. Web page is converted to plain text format, and processed by a keyword extraction algorithm. To extract keywords from the document, we make use of the Term Extraction Service, part of Yahoo! Search Web Services [64]. This allows us to send plain text documents to the Web service and retrieve a set of key terms. Although details of the algorithm are not available, the use of the Term Extraction Service has the benefit of not requiring complex term extraction algorithms and supporting data to be implemented on the user's computer. Users in need of assistive technology often do not have the resources for high performance computing hardware, and minimizing the computational requirements of our system is an important factor in its utility. An area of future work is to implement a keyword extraction component on our own servers to realize more control over the extraction process.

4. Web page is closed (by navigating to another page).

5. Page score (S_P) is calculated from the recorded user activity on the page.

6. Keywords are assigned a normalized weight based on their relative importance in the page ($W_1..W_m$ for the m keywords on the page). Relative importance is determined by the keyword extraction process, which returns results in order of decreasing relevance. Currently we weight keywords from highest to lowest in the order they are returned from the Term Extraction Service.

7. The keyword scores in the keyword list are updated as follows:

- For each keyword that appeared on the current page, a keyword score $S_K(i)$ is calculated from the current page score S_P and the keyword's relative importance $W(i)$ as follows:

  $$S_K(i) = S_P W(i)$$

  Keyword scores are added to the keyword list using the incremental mean formula:

  $$S_{MK}(n+1) = S_K(i) + \frac{nS_{MK}(n)}{n+1} \quad (3.2)$$

  Where $S_{MK}(n)$ is the value previously stored for this keyword in the list and n is the number of pages where this keyword has occurred. Note that if the keyword does not exist in the list, the second term will be zero and the keyword will be added with an initial score of $S_K(i)$. The value of n is incremented in the list for each of these keywords.

- If a keyword on the list is not present on the current page, its score in the list is reduced by a constant ageing factor A, where $0.9 < A < 1.0$. The ageing factor is introduced to bias the list of keywords to more recent user interests, and to ensure that infrequent high interest keywords do not dominate the top of the list. This factor is chosen to balance prioritizing newer interests while maintaining long term interests in the list.

3.8.3 User Searches

A more direct indicator of user interest is search terms entered into search engines or the search fields of other Web sites. As we detect these search terms, they are processed by the same keyword extraction algorithm as discussed in Section 3.8.2. We treat these keywords similarly to those extracted from visited pages. As there is no equivalent to the page score for terms extracted from search fields, we add these terms to the list with an empirically determined initial score. Since these terms are more explicit indicators of user interest than those gathered from visited web pages, we add them with a score representing close to maximum interest (a value of 1 in place of S_P).

The key to making effective use of data entered into Web forms is to differentiate between search fields and other text input fields, such as those used for Web based email interfaces or other form fields commonly found on Web pages. To de-

tect search fields, we compare the text entered into text fields with the URL of the next loaded page. Many search engines will place the search term into the URL of the next page as a parameter (e.g., searching for "dog" on Google gives the page `http://www.google.com/search?q=dog`). If the entered text is found in the next page URL, we assume that the text was a search term and process it as an indicator of user interest. Although this simple method cannot detect all search fields (some custom search fields send search terms using HTTP POST instead of GET), all major search engines use the URL parameter method and are thus supported.

3.8.4 Applying User Interest Keywords

Once our keywords for user interest are placed into the global keyword list and ranked, they can be used to estimate which links may be of interest to the user. To determine the estimated user interest for an element, we iterate through each word in the element's label, and query the keyword interest list to determine if the word exists as a keyword, and to retrieve its score. The element score, S_E, is thus the sum of the keyword score of each word in its label. Words that do not occur in the keyword list have a score of zero.

$$S_E = \sum_{i=1}^{n} S_K(i) \text{ where n is the number of words in the label}$$

After the user has typed a key, KeySurf calculates the element score for each matching element. Elements with higher score are prioritized more strongly in the matching process, and thus are more likely to be chosen as the default match that the user can immediately activate.

3.8.5 Balancing Performance and Prioritization Complexity

As the complexity of user interest prediction algorithms increases, there is a corresponding impact on browsing performance. The first five steps of the user centric search are very fast, and do not impose any significant delay to matching elements. In fact, by initially limiting the search field to visible elements, matching speed is increased compared to searching all elements on the page. Direct history prioritization also has no noticeable effect on performance, as the browser engine already tracks these data and KeySurf only retrieves it. On current desktop computers, the time from key press to highlighted elements is less than 200ms.

However, gathering indirect interest indicators can have an impact on both network and processing resources. Firstly, to retrieve page keywords, we upload the text content of the entire Web page to the online keyword extraction service, which can cause significant network traffic. Compounding the problem is that uplink speeds on consumer Internet connections are many times slower than downlink speeds, so any uploaded data have a larger effect on network congestion. Implementing the keyword extraction module within KeySurf would eliminate this network traffic at the cost of some additional local processing.

A larger concern is the cost of computing element score from the global keyword list when matching. Since the keyword list is stored in a relational database on the hard disk, querying it can take significant time. As keyword matching is performed after the user enters a key and before matches can be highlighted, the process blocks system response and magnifies the perception of any delays. While unlikely to be a problem for users with lower rate keyboard input, users capable of faster typing are likely to be negatively impacted by the delay. Some participants in the passive evaluation of KeySurf (Chapter 5) commented that the response time of the system decreased as the keyword table increased in size. Future work should examine whether this module could be optimized or selectively disabled for users with higher rate keyboard input.

3.9 Input Devices for KeySurf

Naturally the primary input device for KeySurf is the standard computer keyboard. However there are other input devices that can produce character output, and thus be used to operate KeySurf. Section 3.9.1 provides an overview of a speech input system for KeySurf, while Section 3.9.2 discusses how ambiguous layout keyboards can be modified to control KeySurf. Lastly, options for controlling KeySurf with switch input are discussed in Section 3.9.3.

3.9.1 Speech Input

For users unable to use a keyboard or pointing device but able to verbalize their intentions, an alternative is the use of a voice controlled interface. Although consumer speech recognition tools have improved dramatically in recent years, recognition accuracy for dictation is still an issue if the engine is not adequately trained or users

have difficulty with clear pronunciation. However, it is clear that speech recognition accuracy and speed can be improved by restricting the recognition vocabulary to small sets and minimizing the word length of commands [9, 40]. Given our objective of selecting page elements with KeySurf, we can constrain the recognition set to only those words appearing in element labels on the current page. Thus, a user can directly select a link or other element by speaking one or more words in its label. Several similar interfaces have been developed in the past, beginning with speech integration for the Mosaic browser over fifteen years ago [27].

While the active vocabulary generated from each page is small, any word in the page's language can still occur in this set, which dictates that users be able to clearly pronounce a very wide range of words. In a 2000 study by Christian et al. comparing the performance of a direct selection voice browser with mouse interaction for non-disabled users, few recognition errors were encountered [14]. However, the tested pages contained less than five visible links at a time, which is no longer reflective of the complexity of real Web pages. Data from our automated evaluation indicates a mean of 66 visible links per page (Section 4.1). In addition, words appearing in link text must be recognizable by the speech engine, which is not always the case for proper nouns or abbreviations. These limitations may make direct selection systems unsuitable for users with disabilities who have difficulty with clear pronunciation, or who can only pronounce a small set of words with sufficient clarity to be recognized by speech recognition engines.

An alternate voice interface is simple voice spelling. As activating any element with the KeySurf system is designed to require only two or three characters, recognizing a whole word is more information than is required to uniquely select an element. To represent the letters of the English alphabet, we use a slightly modified version of the International Phonetic Alphabet (Alpha, Bravo, etc.) which was designed with distinct sounding words that results in much more robust recognition than speaking the letters directly. This initial alphabet serves as a good starting set, but letters can be mapped to any word that is distinct from other words in the set and that a user is able to pronounce clearly. The addition of numbers and some common browser commands gives a final vocabulary size of 47 words (See Appendix B). We use the CMU Sphinx speech recognition engine [16] to recognize user commands and convert them to the appropriate character. These characters are then transmitted to KeySurf, where they are treated the same as any other text input.

We conducted a user study involving 32 users of various ages (20 to 60 years),

gender and ethnic background to assess the accuracy of the untrained Sphinx speech recognition engine on our set of words. Users achieved an accuracy of 94% on the first attempt. We anticipate that even higher recognition rates can be achieved with more advanced speech engines. This shows that the simplified spelling interaction combined with the KeySurf system has the potential to be a very robust yet efficient voice browsing alternative. The voice-enabled interface to KeySurf is discussed in more detail in [25].

3.9.2 Ambiguous Layout On-screen Keyboards

With prior knowledge of the labels of selectable elements on a page, it is possible to improve the performance of certain types of input devices by constraining their set of possible outputs, and providing information on the probability of characters and sequences. Although the additional information provided by the KeySurf system can be used in any virtual keyboard to improve word completion, this type of integration has the most benefit in ambiguous layout virtual keyboards, which seek to improve typing efficiency by dynamically adjusting their layout based on the probability of upcoming letters. These types of keyboards aim to improve typing performance by making likely letters easier to type. The Dasher project [58] and COGAIN's Gazetalk [30], as well as CanAssist's Dynamic Keyboard [11] (Figure 3.12) are examples of such applications.

Both GazeTalk and the Dynamic Keyboard present only a limited number of large buttons to the user in order to allow users to more easily click buttons with inaccurate pointing devices (such as head or eye controlled pointers). Given that there are more letters than buttons, typing a letter would normally require multiple button clicks. During normal operation, these systems use a statistical model of the English language to estimate the probability of a letter being typed, given the user's previous inputs. In order to reduce the number of necessary clicks, both systems place the most likely next characters in their own buttons, such that they can be selected with one click, while less likely characters are grouped into the remaining buttons.

Naturally a model for the entire English language is not ideal when using an ambiguous layout keyboard to control KeySurf, as the probabilities of letters depend on the labels on the current page. To facilitate the integration with dynamic input devices, KeySurf can expose the labels for all elements extracted from the current Web page, as well as their corresponding relative priority. Any virtual keyboard or

other dynamic input device may access these data to improve performance for the task of typing characters to select an element.

When the Dynamic Keyboard is used to control KeySurf, the standard language model is replaced by a language model generated from the exposed Web site text and priorities. As there are far fewer distinct letters and letter sequences on a Web page compared to the entire English language, the probability of likely letters being assigned their own button is much higher, allowing the user to produce character output with fewer clicks.

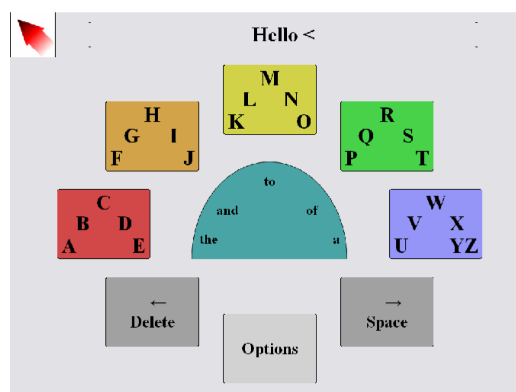

Figure 3.12: The interface of CanAssist's Dynamic Keyboard.

3.9.3 Switch Input

KeySurf can be controlled using switch input devices using an on-screen scanning keyboard as described in Section 2.2. Unlike scanning page elements directly, which is fundamentally unpredictable, re-using a standard scanning keyboard allows the user to take advantage of their knowledge of the keyboard layout and their muscle memory to operate at higher scanning speeds. Expert users with familiar scanning interfaces can use scanning rates as fast as 0.1 seconds if the scanning order is predictable [39].

One can think of the switch scanning process as a code, since each letter can be produced with a sequence of predefined inputs. These inputs can be a switch activation or an automatic delay timeout, but once the letter layout is fixed, the sequence of inputs for a given letter will not change. While a static approach promotes rote learning, this can only be taken advantage of by a user that is able to operate their switches with high accuracy and speed. The disadvantage of using a static switch scanning keyboard is that since the arrangement of letters is tuned towards

typing in the English language, it is not always optimal for selecting from the specific links on the page. It is evident that if a row-column keyboard were able to adapt its layout to the current set of elements on a page, it would require fewer inputs to achieve the same result.

To achieve maximum selection efficiency, the first letters of important elements should be the easiest to type (requiring the fewest switch activations and timeouts). Since the starting letters of important elements will change on every page, and indeed as the user scrolls down a page, the optimal code for the visible elements will need to be dynamically generated.

In order to create a system that dynamically creates an optimal code, we must either integrate KeySurf with a row-column scanning keyboard (similar to Section 3.9.2) and modify the letter layout when required by a page load or scroll event, or assign codes to the elements directly. If the code is dynamic, there is little advantage to integrating with an external scanning keyboard, as the dynamic layout changes as the user scrolls would be very disorienting. By assigning codewords directly to each element, we can bypass the need for an additional display window and show codewords directly on the Web page along with the elements themselves.

Encoding Web Page Elements

To obtain an optimal code for the Web page, we use the Huffman algorithm to assign codewords to elements. To reduce the number and length of codewords on large pages, we only consider elements that are currently visible on the page. Since the elements have already been ranked by their likelihood of selection (by the user centric search and history modules), we need to only create the code based on these probabilities. To accomplish this, the Huffman algorithm was implemented in JavaScript to run inside KeySurf. The implementation of the Huffman encoding process is shown in Appendix C.

When the page loads, the initially visible elements form the base set and are encoded into a binary code tree. As the user scrolls down, some elements may scroll out of the visible view, while others become visible. The simple solution is to recode all elements when the number of visible elements changes, however this could result in a disconcerting effect for the user, as the codewords for elements change during page scrolling.

To maintain constant codes for visible elements, the code is only partially rebuilt

when the page is scrolled. The Firefox extension is notified on each scroll event, and the new visible elements are compared to the previously visible elements to identify the newly exposed items as well as the ones that have dropped off the screen. Dropped items are first purged from the code tree and then the new elements are encoded based on the partially completed code tree. Naturally, if none of the original items are visible in the new view of the page, the code is generated from scratch in the same way as immediately after page load.

If the number of elements dropped is greater than or equal to the number of elements added, we just reassign the freed codewords to the new elements according to priority. If there are more new elements than removed ones, we change the longest codeword to open up the tree for new additions. In the future, if a high priority element is added it should take the place of an element with lesser priority.

Visualizing Element Codes

One of the difficulties of using binary input directly to select elements with alphanumeric labels is that the codewords mapped to the elements must be communicated to the user in some way. With the regular KeySurf, this is not a problem, since the input alphabet is the same as the label alphabet and thus the text of an element is also its codeword. One possibility is to represent the binary input as two symbols (such as dot/dash or 1/0), turn these codewords into labels and then overlay these labels onto the elements. Unfortunately, overlaying codewords onto elements occludes page content and creates a difficult label layout problem. These problems can be observed in other keyboard navigation extensions using the overlay technique, as shown in Figure 2.3. Binary codewords exacerbate this issue, as they are on average much longer than the numeric codes used in ID navigation systems.

Consequently, the codewords associated with elements must be communicated to the user in another way, taking care to modify the page layout as little as possible. Since we only have two symbols in our codewords, we can represent our code alphabet with colour instead of glyphs. To facilitate fast recognition and prevent problems for users with red/green or yellow/blue color blindness, we chose yellow and light red as the colours to represent the binary input. These colours are separated early in the visual system and should provide sufficient contrast difference for even users with monochromatic vision to distinguish.

After the codewords are calculated, the corresponding elements are highlighted

with the colour representation of the codeword, with the background of each letter representing one bit. An example of this highlighting is shown in Figure 3.13. On this page section, the large text of the headline has given that element the highest probability while the category links are in the middle, and the sub-heading has the lowest score. This is reflected in the codewords, where the headline was coded with the shortest codeword (just one bit), and the other elements are assigned longer ones.

Figure 3.13: Element highlighting showing the effect of priority estimation.

One of the problems evident in this screenshot is the effect from variable spaced fonts. Human perception of hue and saturation changes when the coloured area is very small [8], and with letters like the lowercase "i", the coloured background can be reduced to just a few pixels. One possible solution is to set the font of elements to a monospace family to allocate a constant area to each bit in the codeword.

3.10 Limitations of KeySurf

It should be noted that as it is described in this work, KeySurf is not compatible with all existing Web pages and content delivery technologies. While this is certainly a longer term goal, technical limitations related to accessing the internal structure contained within external plugins make achieving this goal difficult. In the scope of this work, we do not attempt to make the content contained within plugin-controlled areas (such as Flash [2], Java [3], or Silverlight [4]) accessible from KeySurf. For keyboard access to this content, we must rely on the keyboard support provided by the plugin. Fortunately for keyboard users, the increasing capabilities of newer HTML standards have reduced the use of browser plugins to deliver dynamic content.

[2]www.adobe.com/products/flashplayer/
[3]www.java.com
[4]www.silverlight.net

In addition, Web pages incorporating complex client-side dynamic content or script-defined actions are not fully supported in KeySurf. KeySurf does recognize some script-defined actions in Web pages (i.e. the onclick attribute), but does not support all dynamic elements within Web pages. However, complete support of these script-defined elements should be technically possible and remains an important area of future work.

Another limitation, shared with all keyboard navigation methods, is that KeySurf does not have an equivalent for mouse actions other than a left click. Elements that react to the mouse hovering over them (popup menus, button highlights, etc) or dragging (online mapping, Web applications, etc) cannot be manipulated by KeySurf without additional support from the Web page author. As a workaround for Web pages that provide their own extensive keyboard shortcuts, KeySurf can be temporarily disabled to prevent conflicts.

Chapter 4

Evaluation by Simulation and a Pilot Study

For the performance of KeySurf, we are primarily concerned with the number of characters necessary to activate a link. A subset of this is how often the default link suggested by KeySurf matches the user's intention, and how often the user must resort to using a numbered shortcut to select their link.

This chapter describes the results of a simulation of keyboard navigation using KeySurf on real-world Web sites, as well as the results from the small pilot study involving 4 persons with disabilities.

4.1 A Simulation to Measure Navigation Efficiency

Using the user centric search method with the numbered selection shortcuts (Section 3.5), we determined the upper limit of keys required to uniquely select any element to be two (three to activate). However, the actual number of keys necessary on typical Web pages depends on the distribution and text of visible elements. As the number of visible elements increases, the probability of the first character of a desired element being unique amongst the starting characters of all visible elements on the page decreases. Due to the default selection that allows the activation of one (green highlighted) element even with multiple matches, if there are n visible elements starting with the same character, then one element will be uniquely selectable with one keystroke, while $n-1$ elements will require two keystrokes (assuming that not more than 11 links start with the same two characters).

To determine the average number of required keystrokes on real Web pages, we implemented a modified version of the KeySurf system that automatically loads randomly selected Web pages, and calculates the keystroke cost for each visible element on that page. Web pages were chosen by starting the system at various popular Web portals, and KeySurf calculated and recorded the keystroke cost for each visible link on the page. Then KeySurf would select a random link on the page and follow it to a new page, where the process is repeated. The system analysed a set of 726 unique Web pages, and calculated the keystroke cost for 48,182 links. Average keystroke cost per page is shown in Figure 4.1.

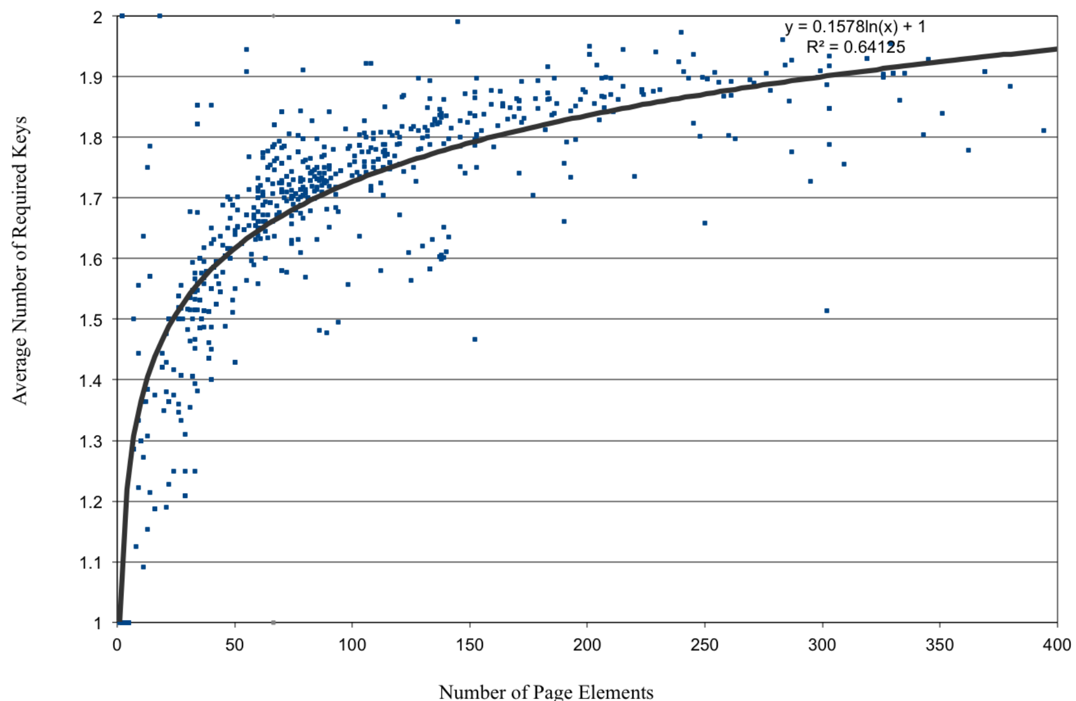

Figure 4.1: Average required keystrokes per element selection.

The mean number of elements per page was 128.5, with 66.4 elements visible when the page loaded. The overall mean keystroke cost for all pages was 1.69. Activating an element after it has been selected adds one key (enter), giving an average of 2.69 keys required to follow a link or activate another Web page element. This provides the baseline performance of the incremental search algorithm used in KeySurf. Since all links on a page are selected, this simulation does not benefit from the user centric algorithm described in Section 3.4, nor does it benefit from history or user interest

indicators. We anticipate that performance with real, user generated Web sessions will improve further over these results, as these modules come into effect and decrease selection cost for elements that users are more likely to select.

4.2 Initial Performance Indications (4-User Mini-Study)

To determine the usability of the system for those with motor disabilities, we conducted preliminary testing to gather evidence on how KeySurf performs for some representatives of the target population. This initial testing aimed to compare our system to the browsing method that study participants are most familiar with. At this point we do not aim to compare KeySurf to other advanced keyboard navigation systems, as none of the participants were using such systems, and we want to initially determine if KeySurf would be an improvement over their usual browsing method. Participants are described in Section 4.2.1, the experimental setup is detailed in Section 4.2.2, and results are presented and discussed in Section 4.2.3. All user testing was approved by the University of Victoria Human Research Ethics Board.

4.2.1 Participants

We conducted our test on four persons with Cerebral Palsy. Participant age varied from 17 to 35, with all participants having experience using computers and navigating on the Web. Three participants used pointer based input devices to navigate the Web at home, while one used a regular keyboard. One participant is deaf and non-verbal but is able to communicate by using sign language and reading lips. One other participant is also non-verbal and communicated with a text to speech assistive device.

The test consisted of participants performing a Web navigation task with a pointer device using a regular browser as well as the same task with a keyboard device using KeySurf. Table 4.1 shows the input devices used during the test by each user. Both the pointer and keyboard devices were chosen individually for each user to match the device they used at home as closely as possible.

Subjects A and C used a large Penny Giles track ball (Figure 4.2) designed specifically for people with disabilities to control the mouse cursor. Although pointing

Subject	User's Accustomed Device	Keyboard Device
A	Penny Giles Track Ball	Regular keyboard
B	InfoGrip Joystick Plus	Intellikeys USB
C	Penny Giles Track Ball	Regular keyboard
D	Regular keyboard	Regular keyboard

Table 4.1: Input devices used by test subjects.

accuracy was a problem for subject A due to difficulty with fine motor control, subject C was quite accurate at the expense of movement speed. Subject B used a joystick to control the mouse cursor with good accuracy and speed. Subject D was not able to use any pointing device and used a keyboard for both tasks.

Figure 4.2: Penny Giles track ball [29].

For interfacing with KeySurf, subjects A and C used a regular computer keyboard, on which they typed letters with one finger, while subject B used a larger, pressure sensitive keyboard designed for people with disabilities (Intellikeys USB). The plastic key guard for this custom keyboard which subject B was accustomed to was not available during testing, which slightly impaired his/her typing performance. Subject D used a regular computer keyboard for both tests, using his/her mouth to press keys. With an unmodified browser, subject D presses the Tab key to advance the element focus on Web pages.

4.2.2 Experimental Design

We designed tests to measure the time necessary to select a visible link using each subject's accustomed input device and compared it to the time required using KeySurf with a character based input device (such as a keyboard). We picked two sets of 12 Wikipedia articles with similar layout such that one can navigate from page 1 to page

12 in each set by following links. The two sets were chosen such that the spatial location of each link leading to the next page was similar between sets, but different in successive pages. In other words, if L_{Ai} represents the location of the target link in the ith page of set A, then $L_{Ai} \approx L_{Bi}$ and $L_{Ai} \neq L_{Ai+1}$. Page content was not considered in our choice of articles.

Prior to commencing the test, operation of our system was verbally explained to users. As some subjects have communication difficulties, the time required for this varied, but did not exceed ten minutes. The steps required to select a link with KeySurf were explained as follows:

1. Type the first letter of your link.
2. If your link turns green, press "Enter" to activate it.
3. If your link turns yellow with a number beside it, type the number and press "Enter".
4. If your link turns yellow without a number, type the next letter in the link and press "Enter".

Users were given time to practice selecting links on several Wikipedia pages of their choice with both input methods until they felt comfortable with both. Two random subjects were assigned to start with KeySurf, while the other two started with their usual method. Times for the selection process on each page were recorded separately.

To test selection performance, the first page in a set was loaded and the desired link was pointed out to the user (by physically pointing at the link on the screen). At the same time, a timer was started which measured the time the user required to follow the link. The procedure was repeated on all pages in the set and again using the other selection method on the other set of pages.

4.2.3 Results and Discussion

Mean link activation times and standard deviation for each user are presented in Figure 4.3.

For users accustomed to using a pointing device to navigate on the Web, the resulting times show that the relative performance of KeySurf depends on the user's typing rate versus their ability to accurately control the on-screen pointer. To determine the significance of the differences in mean selection times, we performed

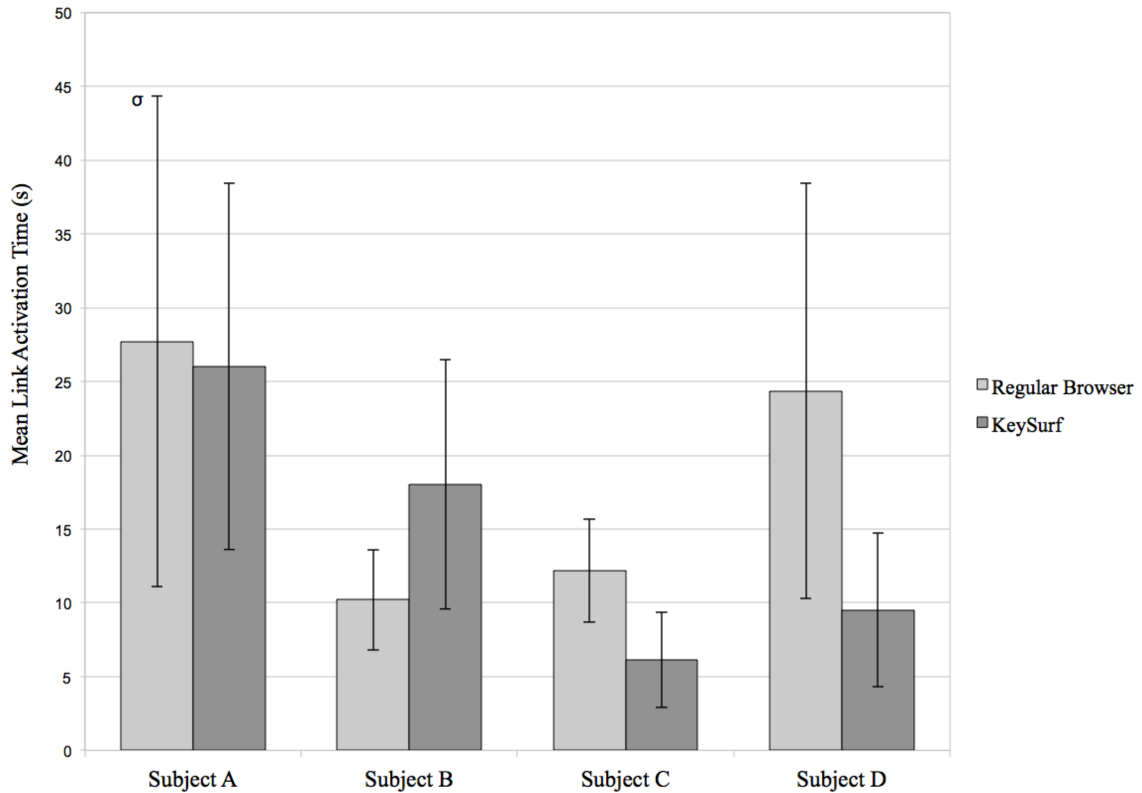

Figure 4.3: Mean link activation times with KeySurf and regular Web browser.

independent T tests assuming unequal variances. We hypothesize that there is a difference between the mean link selection times between the two navigation methods, with the null hypothesis of no difference in means. We assume that user times are normally distributed. P values for subjects A through D were 0.79, 0.014, 0.0042, and 0.0060, indicating that the null hypothesis could be rejected with high confidence for all subjects except A.

While the mean selection times for subject A were not significantly different between methods, the range of values is more constrained using KeySurf than with the pointing device. This is due to the fact that subject A frequently had problems with involuntary movement when controlling the track ball. This led to highly variable selection times, as some links could be selected very quickly, while others required several attempts. In contrast to this, subject A's typing performance was quite constant, making selection times with KeySurf much more predictable. However, more study is necessary to determine if the use of KeySurf would be beneficial to subject A.

Although subject C was accurate with the track ball device, performance with the KeySurf was significantly better, with a mean time of 6 seconds versus 12 for the track ball selection. In an informally extended testing session with subject C, the advantage of KeySurf was found to increase further for selection tasks that required scrolling, since with the track ball the subject has to move to the scroll bars to scroll down, and then move back to the Web page content to make a selection.

Subject B did not benefit from the keyboard selection process used in KeySurf. In fact, mean link activation time with the joystick was significantly lower than with the keyboard. This result stems from the fact that subject B was very accurate with the joystick, but had trouble accurately pressing keys without the help of the key guard that he/she was accustomed to. Although KeySurf performance for subject B may increase with the use of a better keyboard, we anticipate that for this subject the regular pointing interaction is more suitable.

Since subject D already used a keyboard to navigate the Web, it was expected that any improved keyboard navigation system would have a large effect on element activation times. This was verified by our testing, where mean selection time using KeySurf decreased by 60% over this subject's accustomed method. Subject D was very pleased with KeySurf and has begun using the system at home.

Chapter 5

Evaluation on Real-World Web Sessions

An issue that surfaced with the limited pilot study (Section 4.2) is that since the performance of KeySurf is highly dependent on the content of page elements, the user's history, and the target link, any controlled study will be highly biased by the choice of navigation path. If the navigation path is chosen by the study authors, it is highly likely to be somewhat biased towards the navigation method under investigation. Although random navigation paths could be tested, this would require a very large number of testing sessions to effectively reduce the impact of the random selection on results, rendering it infeasible for a lab setting.

While the automated selection simulation discussed in Section 4.1 provides a good baseline for KeySurf performance, its random nature discards many of the KeySurf algorithms designed to reduce the selection effort for links that users are more likely to select. The baseline keystroke cost of 2.69 keys/element represents the performance of only the simple search characteristics of KeySurf, and does not provide any insight as to the effect of the user centric search model.

The goal of this study is to evaluate the performance of KeySurf on real Web sessions, recorded with a version of the KeySurf extension modified to work in a passive mode. A secondary objective is to gain some insight into how users surf the Web, and whether techniques like prioritizing previously visited pages can be useful. In addition, the collected data is used to provide a more accurate basis of comparison to other Web navigation methods.

For the performance of the KeySurf system, we hypothesize that:

> The consideration of user centric factors in KeySurf will reduce the number of required key presses per element activation.

The study design and why it was chosen are discussed in Section 5.1. Participants and recruitment are described in Section 5.2, while subsequent sections discuss data collection (Section 5.3) and results (Section 5.4). Section 5.5 compares different Web navigation techniques using keystroke level modelling and discusses the application of study results to these models.

5.1 Study Design and a Note on Studies Involving Persons with Disabilities

For many studies in software engineering, the goal is to assemble a group of people that have as many characteristics in common as possible to form the pool of study participants. For example, a group of participants may be drawn from all employees of a particular company with the job title ”Developer”, or all graduate students in computer science. Such groups of subjects can be reasonably assumed to have comparable physical and mental capabilities, and similar experience and education. This property allows experiments to be designed with the assumption that the results from different subjects can be meaningfully combined to form a more complete picture of the results, as well as being generalizable to a wider population.

This situation falls apart when studying people with disabilities. The term “disabilities” is very wide and includes everything from complete physical immobility, to mental and cognitive disorders, to mild learning disabilities. With this general definition, statistics show that approximately 18 percent of the US population have some sort of disability [5]. The target audience for KeySurf is constrained to people with a physical disability which prevents them from accurately using a mouse or other pointing device. However, even with this constraint there is still a large variability in the ability of people that could potentially benefit from KeySurf, and that could be selected to participate in the study.

This situation affects almost all studies done with people with disabilities. The common result is either that researchers attempt to find subjects with similar conditions (for example, people in late stage Amyotrophic Lateral Sclerosis) and end up with very few subjects or researchers use people with varying abilities and end up with very large variances between the results of different subjects. Our own pilot study

suffered from both of these problems, in that while subjects all had Cerebral Palsy, their level of motor impairment was highly variable. This makes between-subject experimental designs very difficult to conduct in studies involving people with more severe or complex disabilities.

One other factor working against the involvement of large numbers of disabled subjects in studies is simply the logistics of organizing study participants. Subjects with the severe physical disabilities that we are interested in face many problems with time constraints of when they feel well enough for an experiment, constraints in organizing accessible transportation, and requirements for specialized computer interface devices. Setting up a customized test environment that works for all subjects without affecting the results is sometimes impossible.

Our original study design for real-world evaluation involved participants with physical disabilities using KeySurf for a 2-week period as their Web browsing tool with passive data collection and a post-test questionnaire to gather qualitative data on the usability and performance of KeySurf. Unfortunately, it was not possible to recruit enough participants for this study, partially due to the constraints mentioned. Most importantly, the number of people with physical disabilities which significantly affect their operation of a computer and who are also Web users is small. For those able to use a pointing device, switching to a keyboard to browse the Web for two weeks is a large burden as there is no personal benefit. Conversely, those more able to use a keyboard and not a pointing device are less likely to be Web users, and the effects of learning to use the Web and a new navigation method at the same time would confound the study data or result in very few Web sessions over the testing period.

Due to these complications with the initial study design, the evaluation was performed on users without disabilities to gather a greater volume of Web session data. To maximize the data collected, participants accessed the Web using regular mouse interaction, while a modified version of KeySurf ran in the background and calculated the keys necessary to perform the same element activations as were activated with the mouse. Given the sensitive nature of recording information about Web sessions (despite that Web addresses and user inputs are not recorded), it remained somewhat difficult to find participants for the modified study design. However the volume of data collected from most participants was sufficient for the purposes of our statistical analysis on the performance of KeySurf.

5.2 Participants and Recruitment

The study involved 11 participants (8 male, 3 female) consisting of adult Web users recruited via poster advertisement in a university setting. Study participants were not directly compensated but were entered into a draw for a set of movie tickets as incentive for participation. The participants were instructed to use a modified passive version of KeySurf on the Firefox Web browser to replace their customary browser for a period of at least two weeks. Participants were asked to continue using the Web as normal, however since their regular browser was still available, participants who were concerned about monitoring of their browsing chose to use the instrumented browser only for some of their Web browsing during the evaluation period.

The study took place in a work or home setting, depending on where participants chose to install the KeySurf extension. Six individuals used the software in a work setting, while five used it at home.

5.3 Data Collection

The passive KeySurf is a version that removes all user interface and cannot be operated with the keyboard directly. All the same algorithms are present as in the regular version, however selection with the keyboard is simulated after the user clicks a link or other Web page element using the mouse. For example, if a user clicks a link to another page, KeySurf calculates the sequence of characters that would have been necessary to select that same link using the KeySurf algorithms.

During the study, KeySurf recorded information about the browsing session in the background which allows us to gather information about how KeySurf would perform for the same Web navigation path, and what page characteristics influence KeySurf's performance. KeySurf records when Firefox starts, which marks a new Web browsing session. Subsequently, each page load is recorded, as well as every mouse click, link activation, scrolling, and keyboard event. Full information about what events were recorded and the parameters of these events are detailed in Appendix A. To protect participant privacy, page keyword data was not collected (although it was considered by KeySurf as part of the navigation process), and thus the correlation of gathered keywords with user interest was not evaluated as part of this study.

The user interaction data recorded by KeySurf is analysed to extract components of the user's browsing patterns such as their degree of browsing locality and number

of elements per page, as well as performance measures for KeySurf such as keystroke cost per activated element and when shortcuts are required to achieve maximum efficiency.

5.4 Results and Discussion

During the course of the study, KeySurf recorded 441 browsing sessions, encompassing 5654 pages loaded. Of the pages loaded, 3159 or 56% were unique URLs, while 44% were repeat visits to the same pages. Of the pages loaded, 774 were on unique domains. KeySurf recorded 4601 clicks on links and other active elements on Web pages.

An additional 1878 clicks were recorded where the action could not be reproduced using the KeySurf algorithms. That is, an element was clicked with the mouse that could not have been clicked if a user were using KeySurf with a keyboard. Clicks that cannot be reproduced by KeySurf and thus fall into this category can come from either:

1. Clicks on elements that are not active. These clicks could be due to other interaction with the page (selecting text or clicking a page to clear focus from a form field), or unintentional clicks (e.g., a click that was aimed at an active element but missed, or an element that appeared to be active but was not).

2. Clicks on elements that KeySurf cannot determine are active (i.e., certain elements that have their behaviour defined by JavaScript in in a manner not recognized by KeySurf).

Most commonly, clicks in the second category occur within complex Web applications such as online mapping, some modern Web mail clients, or other pages where functionality is delivered using JavaScript rather than traditional HTML elements. All the links that could not be reproduced in KeySurf are not included in the analysis of Web browsing behaviour.

Figure 5.1 shows the number of page load events and link clicks that were recorded by KeySurf during the study. Participant 5 did not use the tool extensively during the study and did not produce sufficient Web browsing data to produce reliable statistics (49 page loads and 18 link clicks) and was thus removed from further analysis. The

number of page loads is greater than the number of link clicks for all participants except participant 6 and 7. A greater number of page loads than link clicks is expected, given that aside from being the result of a link click, page loads can also result from starting the browser (and loading the home page), following a bookmark, or reloading a page.

Participant 6 shows an opposite relationship which is due to the browser being used primarily to access a Web application on an Intranet which uses internal frames and JavaScript to present information without changing the URL of the Web site. The data in Figure 5.2 also illustrate the narrow pattern of this individual's Web browsing, as all 443 page loads occurred on only 19 unique domains. Data for participant 7 shows an approximately equal number of page loads and link clicks, which would suggest use of a page with a similar internal frame structure, however this was not confirmed.

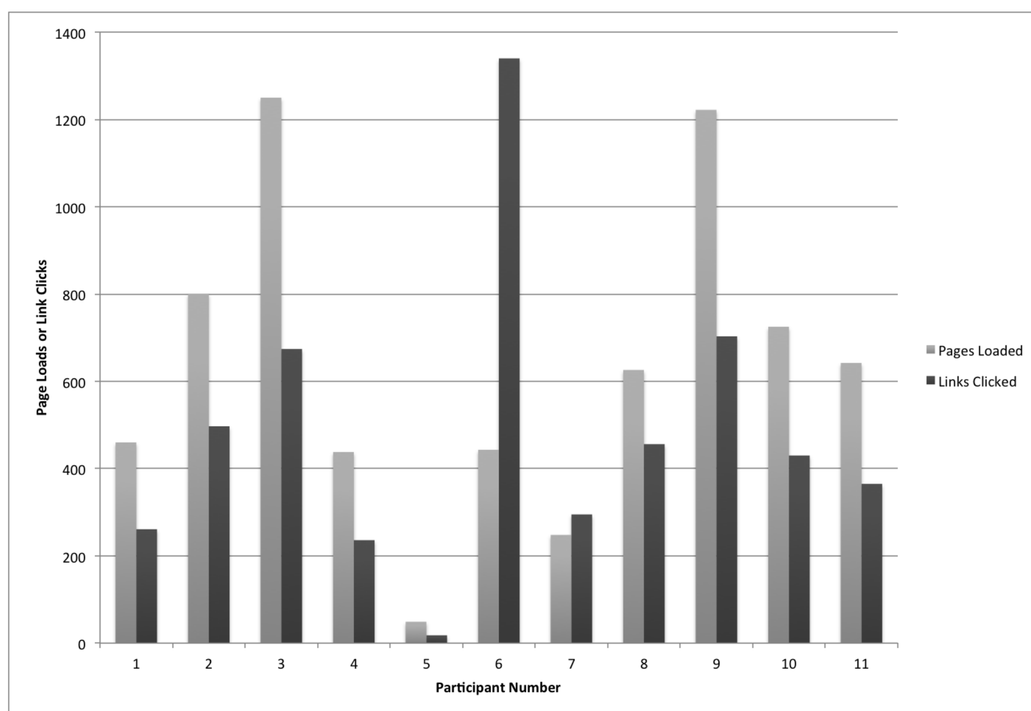

Figure 5.1: Number of pages loaded and links clicked by participant.

Figure 5.2 shows how often participants tended to visit new sites and Web pages, versus visiting existing Web sites. Note that page uniqueness was defined by the full site URL, so changes in parameters in URLs are counted as separate pages. This can sometimes cause two identical pages to be counted as different. For example, the

google search page for "sports" `http://www.google.ca/#q=sports` is visually the same as when another parameter is added to the URL (e.g., `http://www.google.ca/#q=sports&fp=e86c`) but will be marked as a unique page load by KeySurf.

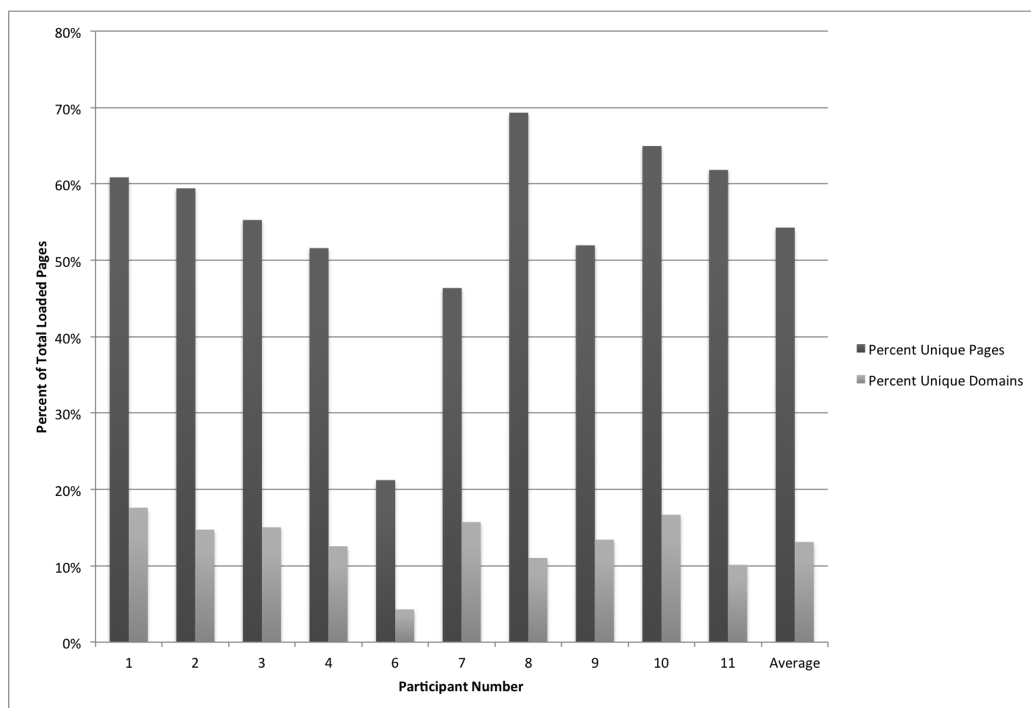

Figure 5.2: Unique pages and domains visited by participant.

5.4.1 KeySurf Performance

For each participant, the average number of keys necessary to select an element with KeySurf is shown in Figure 5.3. The overall weighted average for all participants is 1.38 keys per selection (2.38 keys to activate or click the element with the Enter key). This compares positively to the results from the simulation (2.69 keys to activate), where the number of keys necessary to activate was calculated for all elements on each page. However, since keystroke cost is influenced by the number of elements on the Web page, we must normalize this variable before being able to compare the keystroke costs directly.

The weighted mean number of elements per page for this study across all participants was 155 compared to the average page in the simulation, which had 128.5 (see Figure 5.4). To compare these two results, we find the mean keystroke cost in the simulation for pages containing a mean of 155 elements using two methods. As

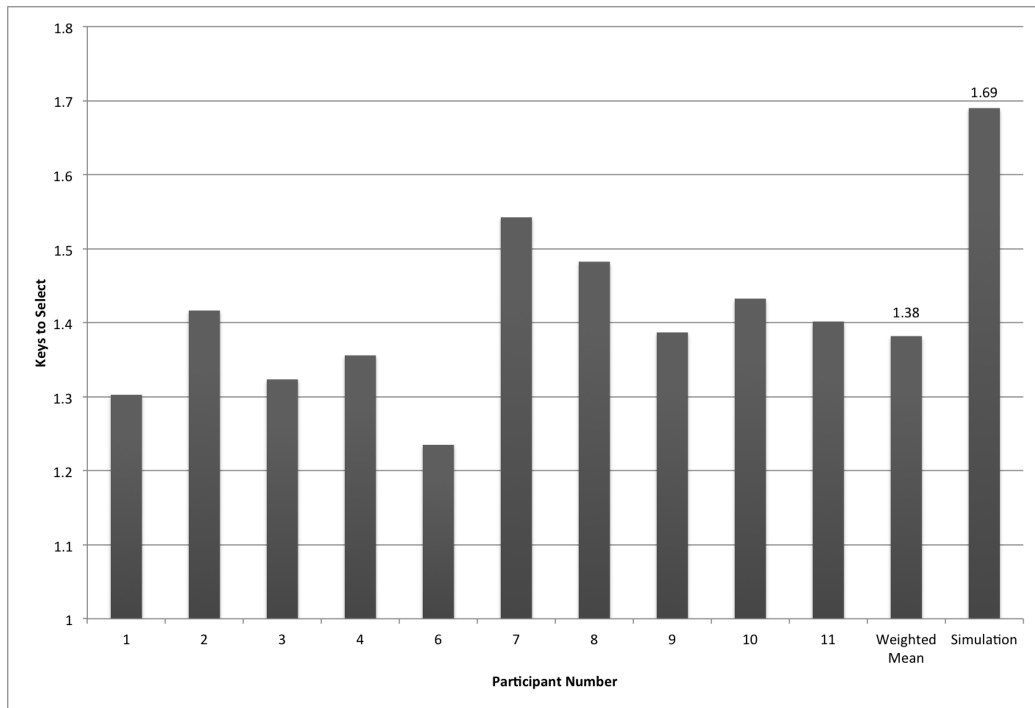

Figure 5.3: Mean keystroke cost by participant.

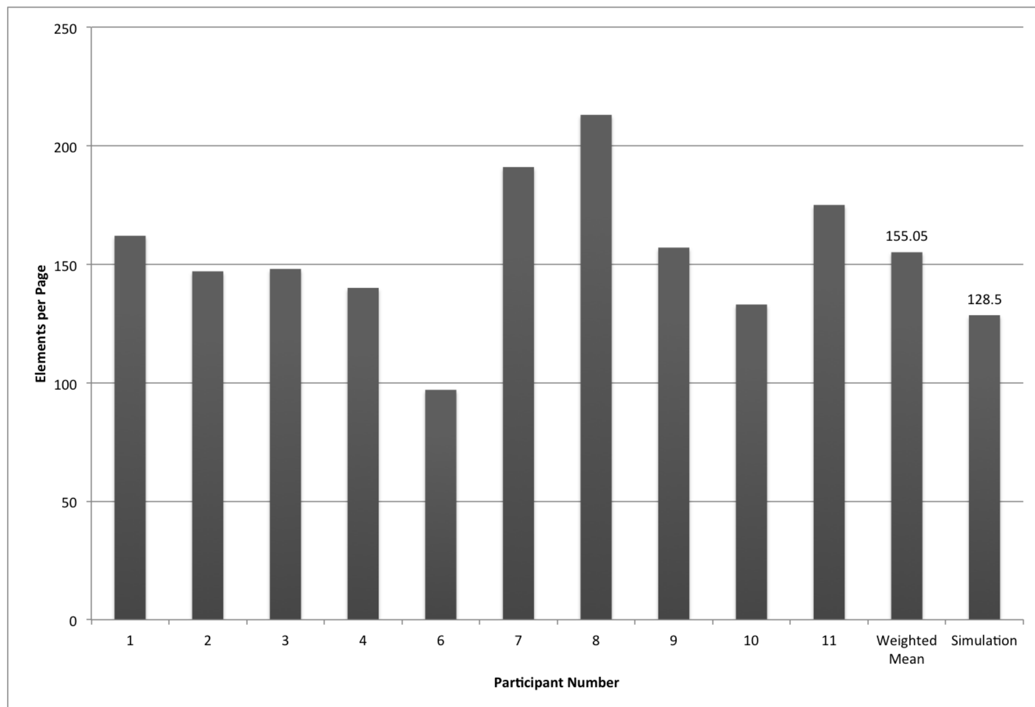

Figure 5.4: Elements per Web page by participant.

shown in Figure 4.1, there is a roughly logarithmic relationship between the number of page elements and the keystroke cost of each link. Although the fit is not strong ($R^2 = 0.64$), we can approximate the keystroke cost as shown in Equation 5.1 (where n_K is the mean keystroke cost of elements on a page with n_P elements). For $n_P = 155$ we get $n_K = 1.796$ keys/element. To confirm this result, we return to the simulation data and determine the mean keystroke cost for all pages where $n_P = 155 \pm 20$, which yields $n_K = 1.800$ keys/element.

$$n_K = 0.1578 \ln(n_P) + 1 \tag{5.1}$$

Based on these data, we can conclude that the user centric search and consideration of browsing history as employed in KeySurf reduce the mean keystroke cost per element by 0.42 keys or 15% compared to simulated results for page sizes of approximately 155 elements per page. Naturally any reductions in keystroke cost vary based on browsing patterns and types of Web sites visited, however for all users in this study, the mean keystroke cost was less than that of the random browsing simulation, demonstrating that there is some correlation between predicted element selection probabilities, and the real browsing patterns of users.

5.4.2 Use of Selection Shortcuts

As described in Section 3.5, numbered shortcuts are assigned to elements that cannot be uniquely selected with two or fewer keys. While the element may be selectable without the shortcuts by typing more keys, in our simulation we assume that shortcuts are always used when they would make selection more efficient. Since one of the design goals of KeySurf is to reduce the visual clutter and page occlusion present in ID navigation, it is interesting to examine how often it is still necessary to show shortcuts over an element. Figure 5.5 shows the percentage of element activations by participant that benefited from a numbered shortcut (i.e., where a shortcut decreased the keystroke cost for the element). For all participants, 22% of element activations benefited from a shortcut.

Although almost a quarter of element activations benefit from the use of a numbered shortcut, it must be taken into account that — unlike in ID navigation — those shortcuts are not shown for every element on the Web page. In fact, for those elements that require two keys to select, the median number of matches after one keystroke is just three. Considering that one of those matches will already be focused

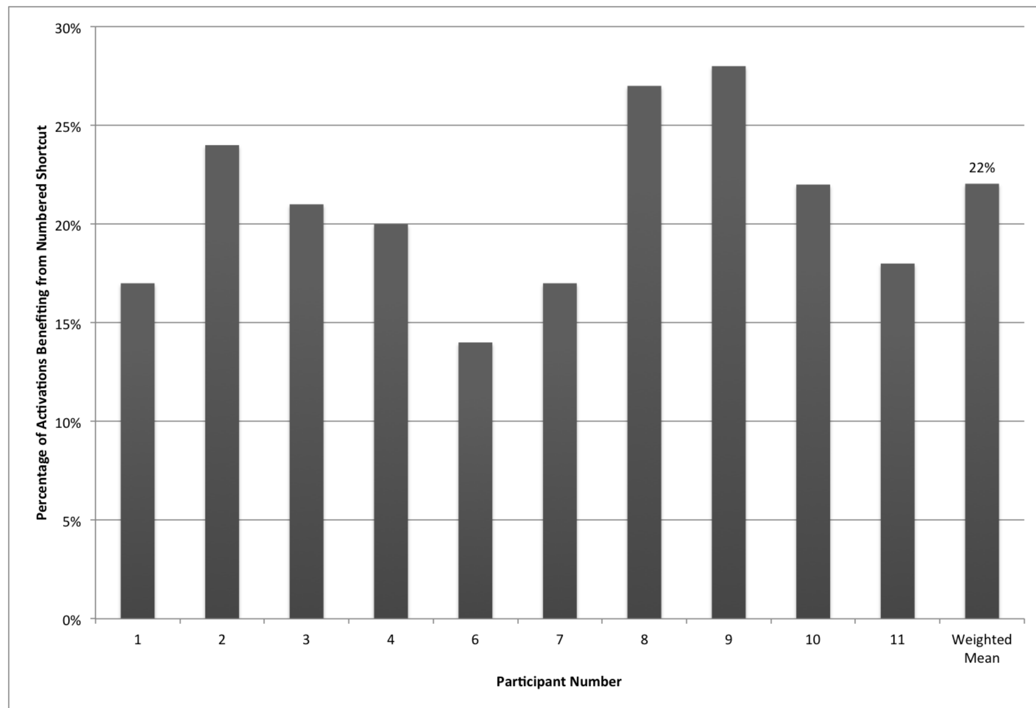

Figure 5.5: Element activations benefiting from a numbered shortcut.

and thus does not require a shortcut, and others may be selectable by typing the next letter in their label, the number of shortcut overlays that are shown is generally very small (most often just one or two). Visual clutter and occlusion by labels is significantly reduced compared to assigning a shortcut to every element.

5.4.3 Impact of History

To examine the impact of history prioritization and our user interest keywords, we extract four subsets of element activations to separate elements that have been influenced by each technique. Our four groups are defined as follows:

1. Previously visited elements that have positive keyword score. These elements are prioritized directly since they have been visited, and indirectly because the element's label has a positive user interest score (one or more of the keywords in the label match the keyword interest list).

2. Previously visited elements that have zero keyword score. These elements are prioritized directly since they have been visited, but their label does not match any of the keywords in the global keyword list.

3. Unvisited elements that have positive keyword score. These elements have not been previously visited, but one or more keywords in their label match the global keyword list.

4. Unvisited elements that have zero keyword score. These elements have not been previously visited, and none of the keywords in the global keyword list match the label.

Elements in the first group are given the highest priority as they are associated with both an explicit user interest indicator (previously visited) and implicit ones (matching keywords from web pages). The fourth group is not directly influenced by these modules at all, as they do not match any user interest keywords. Figure 5.6 shows the numbers of elements and the mean keystroke cost per selection for elements in each group. As expected, the keystroke cost increases for elements where no history or user interest information is known. All differences between means are significant at the 99% level ($p \ll 0.01$).

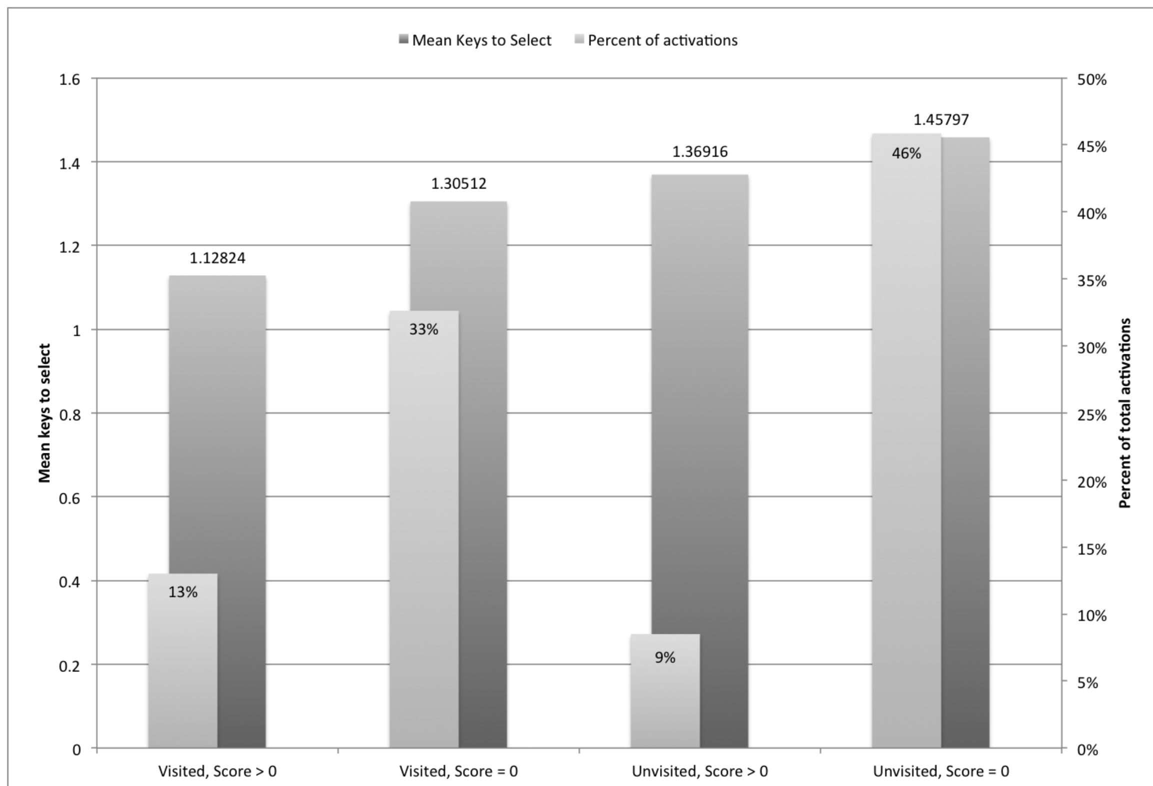

Figure 5.6: Number of elements and mean keys to select by history group.

It is clear that user history increases the selection efficiency of elements that are linked to it. For elements that are both visited and have a positive user interest score, the average keystroke cost per selection is 1.13 keys, or 0.25 keys less than the overall average of 1.38. However, the majority of links activated do not have any user interest score (79% of activations) from keywords contained in the link label. In addition, since control data is not available (KeySurf performance without any history or user interest influence), we cannot definitively conclude that the history modules provide a net benefit to browsing efficiency.

5.5 Model-Based Comparison to Other Techniques

To provide a basis for comparison with other keyboard navigation systems, we construct a model of each system using a Goals, Operators, Methods, Selection Rules (GOMS) Model by Card, Moran, and Newell [13]. Modelling is useful to approximate a human computer interaction task as a series of elementary actions. We use Card et al.'s simplified Keystroke Level Model (KLM) [12] as the navigation systems under investigation can be readily broken down into individual keyboard or mouse operations. The KLM represents a computer based task as a sequence of elementary operations such as key presses, mouse movements, or mental preparation for an action.

The elementary operations of the KLM that apply to our evaluation are listed below.

K Key press and release on the keyboard.

P Point the mouse on an object on the screen.

B Button press or release on the mouse. A button press and release (BB) is often set equal to a key press and release (K).

M Mental preparation or routine thinking. This represents the time required to perform a routine mental task such as finding an element on screen or preparing to perform an input based on information on screen. This does not represent complex mental or creative tasks.

The standard operators W(t) (waiting for the system to respond) and H (switch between mouse and keyboard) are not used in our comparison as none of the navigation systems under comparison involve significant system delays or switching between

keyboard and mouse as part of the navigation process. In the comparisons of navigation systems, we define the task of following a link as starting after the user has identified the link they want to activate since this step is common across all techniques.

Of the KLM parameters, the mental preparation time (M), is the most difficult to estimate, as the time required for this parameter can vary depending on the exact mental task. In a small study conducted by Schrepp and Fischer [45], various tasks falling under mental preparation according to the KLM require highly disparate times for participants to complete. For example, finding a link on a Web page was found to take 9.6 seconds, while regaining lost cursor focus was only 2.61 seconds, and confirming the correct link took 0.93 seconds. However, since mental preparation steps used in our navigation models are all of the simpler type (e.g., preparation to enter a key or confirming that an element is focused), we can assume that the lower times are more appropriate. Kieras [32] gives the average value of M as 1.2 seconds, which we will also adopt as a baseline in this comparison.

Card et al. also provide the estimates for other parameters based on their studies with able-bodied individuals. The key press and release, K, falls into the range of 0.08 to 1.2 seconds depending on the typing skill of the individual. Little research has been done into what times may be realistic for persons with motor impairments, however Keates, Clarkson, and Robinson measured K times of up to 0.6 seconds in a study of 8 individuals with disabilities [31]. It should be noted that data from our pilot study (Section 4.2) indicate that for individuals with more severe disabilities, the time required to type a key may be significantly higher. In that test, mean task completion times (which involved 2 or 3 keystrokes) ranged from 6 to 26 seconds depending on participant. Using our KLM of KeySurf usage (section 5.5.4), this gives K values between 0.8 and 7.5 seconds. Keates et al. propose that the KLM and experimental times for its parameters are still applicable, however for individuals with motor impairment, a key press is actually comprised of multiple cognitive and perceptual cycles rather than a simple automatic motor function as is the case for able-bodied individuals.

Similar factors also apply to the time estimate of pointing a mouse, which Card et al. state falls in the range of 0.8 to 1.5 seconds, with an average of 1.1. Naturally an individual using a slower input device (such as a joystick) would require significantly longer to point to a target on screen. Again, the data from our pilot study shows that link selection times with pointing devices (represented as MPBB in KLM terms)

ranged from 4 to 60 seconds. These numbers show that for users with significant disabilities, the time cost of the motor function components of a KLM can be many times greater than that for able-bodied individuals, while inaccurate pointer control can make pointing times highly variable. By modelling Web navigation methods with the KLM, we attempt to compare mouse and keyboard Web navigation without making assumptions about the values of these parameters.

5.5.1 Mouse Point and Click

The KLM for regular mouse navigation is simple and consists of only three steps: mental preparation to start moving the mouse, pointing the mouse at the link, and then pressing and releasing the mouse button as given in Equation 5.2.

$$\begin{aligned} T_{Ex} &= M + P + B + B \\ &= M + P + K \text{ assuming } K = B + B \end{aligned} \tag{5.2}$$

5.5.2 Tabbing

For the simple linear tabbing method of navigation as implemented by major browsers, we start with the GOMS model representation presented by Schrepp and Fischer [45] as shown in Equation 5.3.

$$T_{Ex} = t_1 + n_{tab}t_4 + n_{tab}pt_5 + t_6 \tag{5.3}$$

where

n is the target element's position in the tab order.

t_1 is the time to find the target element.

t_4 is the time required to press the tab key.

p is the probability (0 to 1) of losing the cursor focus while tabbing.

t_5 is the time required to regain the cursor focus.

t_6 is the time required to confirm and activate the final target.

Since we are interested in modelling activation time after the target element has been found, we can remove t_1.

This model assumes that the user does not overshoot their target, and thus does not have to traverse the tab chain in reverse order (via Shift-Tab). However, it does include another common source of errors, which is the loss of cursor focus while tabbing. Since the visual keyboard cursor is a subtle outline by default, and moves unpredictably relative to the visual layout of the screen, losing the cursor focus is a large component of the inefficiency of tabbing as a method of keyboard navigation. Schrepp and Fischer observed that although the probability of a loss of cursor focus is quite low (for non-disabled users), it comprises a large proportion of the navigation time (over 20% of total task time in their tests). However, since the errors in typing or mouse selection were not modelled, and other sources of error in tabbing (tab overshoot, activating an incorrect target) were not included, we remove this representation of one of the errors involved in tabbing and show the KLM of error-free tab navigation in Equation 5.4.

$$T_{Ex} = M + n_{tab}K + M + K \tag{5.4}$$

5.5.3 ID Navigation

For ID navigation we base our model on that proposed by Baradaran [4] and shown in Equation 5.5. In this model, each element in the Web page is assigned a unique, sequential ID. The IDs are shown on demand while a modifier (i.e., the Alt key) is pressed. The user then types the ID of their target element and releases the modifier key to activate the element.

$$T_{Ex} = T_M + n_{ID}T_K + T_R \tag{5.5}$$

where

T_M is the time required to mentally prepare for entering the ID.

n_{ID} is the number of digits in the element's ID.

T_K is the time required to press a key.

T_R is the time required to activate an element. Baradaran evaluated two alternatives, one where the activation was a timeout (the system automatically activates the

element after T_R seconds) and another where the activation was a keystroke (Enter), in which case $T_R = T_K$. We only evaluate the second option.

However, as many users with disabilities are not able to press multiple keys at once, we use the more common variation of ID navigation where the user presses and releases a modifier key to show element labels, and presses enter to activate an element after typing its ID. Note that unlike the other models, there is no additional mental preparation inserted before the final activation. This can be justified from Card et al.'s guidelines for placement of M operators which state that if an operator can be *fully anticipated* in a previous mental preparation step, it can be left out. Since the entire key sequence to activate an element is known when the ID is shown, the entry of the code and the enter key can be prepared for in a single step. This model is shown in Equation 5.6.

$$\begin{aligned} T_{Ex} &= M + K + M + n_{ID}K + K \\ &= 2(M + K) + n_{ID}K \end{aligned} \tag{5.6}$$

The number of digits in an element's ID depends on the position of that element in the page and the exact algorithm used to assign IDs. For the purposes of this evaluation, we did not reproduce the algorithm for ID navigation in KeySurf, and instead estimate the ID length based on the number of elements in a page.

For a page with n_P elements, the number of digits in the ID (n_{ID}) for any given element can be estimated by summing the probability of the element being assigned an ID of each length possible on the page (logarithmically related to n_P). Equation 5.7 shows the relationship between the number of page elements and the expected mean length of its IDs for a sequential ID numbering within pages containing fewer than 1000 elements.

$$n_{ID} = \begin{cases} 1, & n_P < 10 \\ \frac{9}{n_P} + 2\left(\frac{n_P - 9}{n_P}\right), & 10 \geq n_P < 100 \\ \frac{9}{n_P} + 2\left(\frac{90}{n_P}\right) + 3\left(\frac{n_P - 99}{n_P}\right), & 100 \geq n_P < 1000 \end{cases} \tag{5.7}$$

Using this formula, we can calculate the predicted keystroke cost using ID navigation for the recorded elements activations gathered in this study. Figure 5.7 shows the mean keystroke cost of ID navigation, beside the KeySurf data repeated from

Figure 5.3 for reference. The mean of all participants is 2.07 keys per selection, or 3.07 keys per element activation.

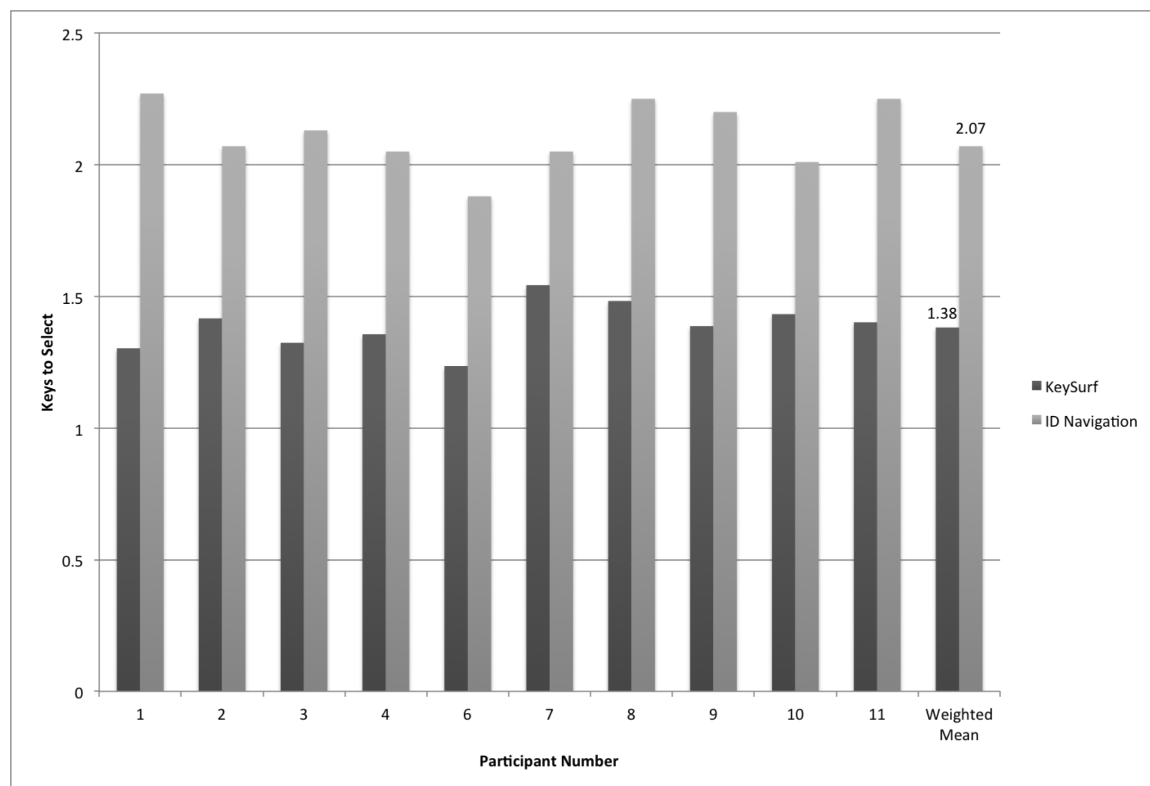

Figure 5.7: Average ID length vs KeySurf selection keys.

5.5.4 KeySurf

To model the KeySurf navigation process, it needs to be determined whether mental preparation steps are necessary between key inputs. Unlike with the more predictable ID navigation, the keys necessary to select a link using KeySurf cannot be assumed to be part of a single cognitive unit as defined by Card et al. Since the necessary key combination for the most efficient code is only revealed after the user presses the first key, the user must re-evaluate their selection after every key press. Although it is possible for fast typists to minimize these re-evaluations by typing multiple characters at a time before stopping to evaluate the selection, this is unlikely to be the case for a KeySurf user with a physical disability. We model the case of slow typists where the effort to re-evaluate is much smaller than the effort of typing additional keys.

In KLM terms, navigation starts with mental preparation to type the first charac-

ter of the target link. After entering the character, the user decides whether his/her target has been acquired based on the highlight color. If the element is highlighted in yellow, the user decides whether he/she should type the next character in its label or use a numbered shortcut to select it. After the element is selected, the user presses the enter key to activate it. The KLM representation of this process is shown in Equation 5.8.

$$T_{Ex} = M_p + n_K(K + M_e) + K \tag{5.8}$$

where

M_p is the time required to mentally prepare for typing the first character.

n_K is the number of characters necessary to focus the element.

K is the time required to press a key.

M_e is the time required to evaluate whether the target element is focused.

The time required to mentally re-evaluate the selection will vary depending on whether the user is able to operate their keyboard interface without shifting their attention from the screen, as well as whether or not the element requires a shortcut to activate. In addition, since the best first key to press is generally the first letter in the link's text, one could argue that mental preparation for the first key should be faster than that for ID navigation, where the code has no natural relationship to the element. However, for the purposes of this evaluation we assume that $M_p = M_e$, which is equal to the standard 1.2 seconds.

5.5.5 Comparison

Given the mean page from our study (155 elements), we can compare the time necessary to activate an element with each navigation technique using these models. For mouse navigation, we use a constant P and set the button press time (BB) equal to key typing time (K). Figure 5.8 shows how activation time is predicted with varying K over the standard range of 0.08 to 1.2 seconds.

Clearly, tab navigation is not an efficient system on sites with a large number of elements. As typing speed decreases, tabbing quickly becomes an infeasibly slow access method for practical Web access. To more closely examine the other systems,

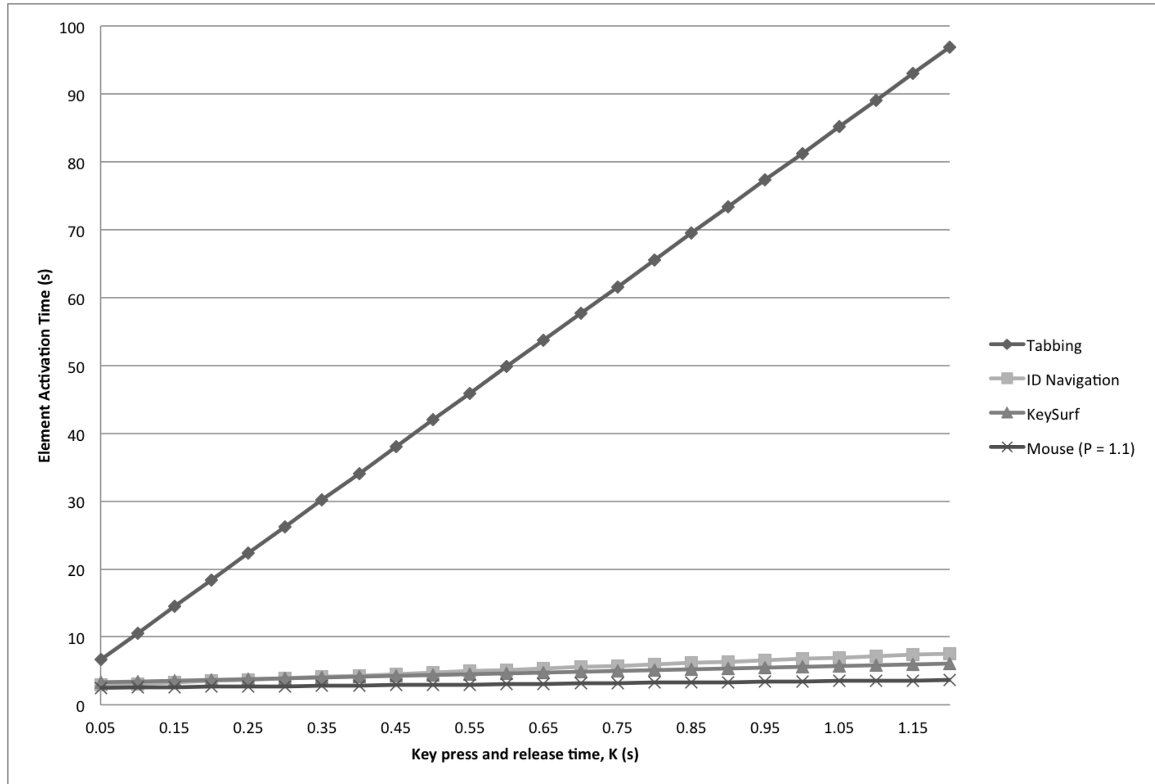

Figure 5.8: Element activation time by system and typing speed.

we remove tabbing from the comparison and re-graph (Figure 5.9). With Card et al.'s suggested average pointing time of $P = 1.1$ seconds, the mouse remains considerably faster than both KeySurf and ID navigation. However, it should be noted that both KeySurf and ID navigation fall within the acceptable range of less than twice the time necessary for mouse navigation as proposed by Schrepp and Fischer.

Comparing ID navigation and KeySurf, we find that element activation time with KeySurf is slower for proficient typists, but increases more slowly with decreasing typing speed. To calculate at what point KeySurf becomes more efficient than ID navigation, we solve the inequality given by the two models (Equation 5.9). Substituting the values of n_K and n_{ID} from our experimental data, we have the relation shown in Equation 5.10. This can be interpreted as follows:

> KeySurf is likely to be faster than ID navigation for users where the time to type one key is greater than one quarter of the time required for mental preparation.

Using the standard $M = 1.2$ seconds, KeySurf becomes more efficient than ID

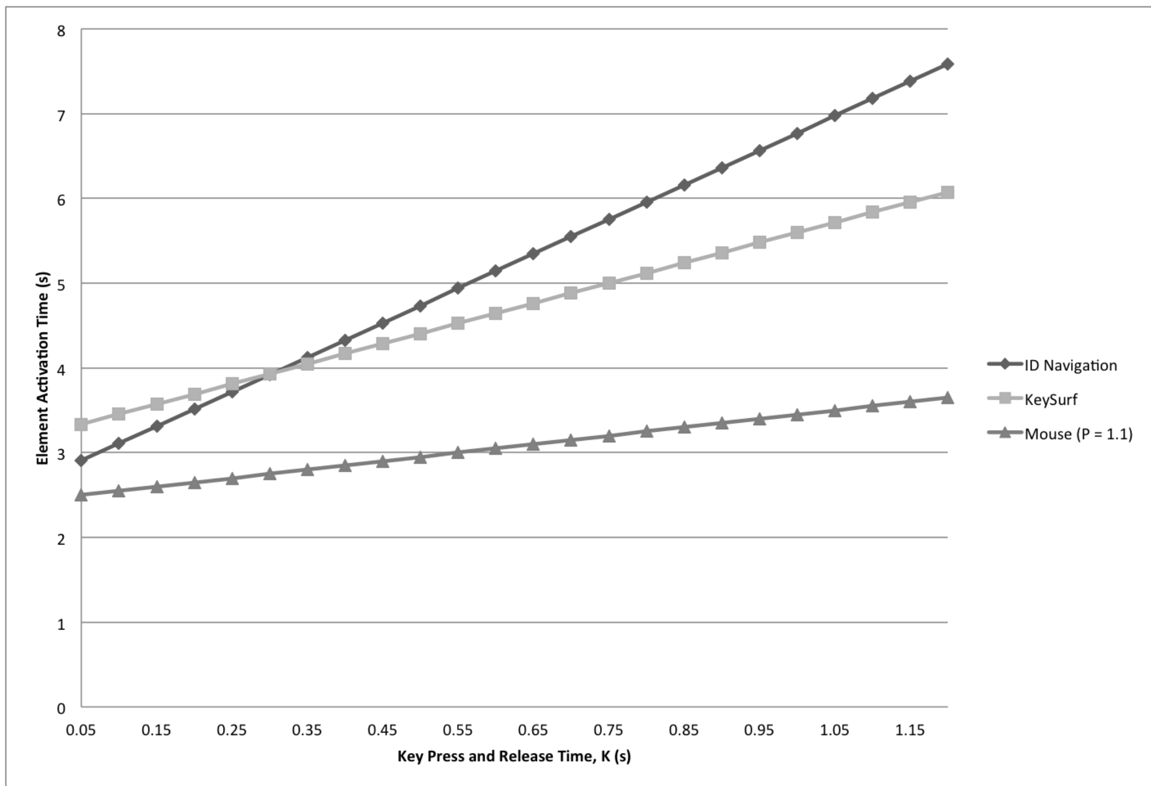

Figure 5.9: Element activation time for ID navigation, KeySurf, and mouse.

navigation when $K > 0.27$ seconds. In other words, KeySurf is predicted to be faster than ID navigation for typists that are slower than average ($K > 0.28s$).

$$\begin{aligned} M + n_K(K + M) + K &< 2(M + K) + n_{ID}K \\ n_K K + n_K M &< M + K + n_{ID}K \\ (n_K - 1)M &< (1 + n_{ID} - n_K)K \\ K &> \frac{(n_K - 1)M}{(1 + n_{ID} - n_K)} \end{aligned} \tag{5.9}$$

$$\begin{aligned} K &> 0.225M \\ &\mid n_K = 1.38, n_{ID} = 2.07 \end{aligned} \tag{5.10}$$

Of course, changing the models will move the intercept point. Specifically one might argue that for ID navigation, the first M operator, where the user is preparing

to press the modifier key, is a very simple cognitive process and should fall at the lower end of the range for mental preparation (0.6 to 1.35 seconds). If this is the case, KeySurf is only more efficient when $K > 0.52M$, or for users with a typing speed slower than 0.62 seconds using standard M.

When compared with using the mouse, we again solve the inequality to determine under what condition KeySurf will be a faster navigation method (Equation 5.11). Substituting in experimental data, we get Equation 5.12. It is more difficult to describe this relation in plain terms, so Figure 5.10 shows the performance of KeySurf compared to using the mouse with various pointing speeds.

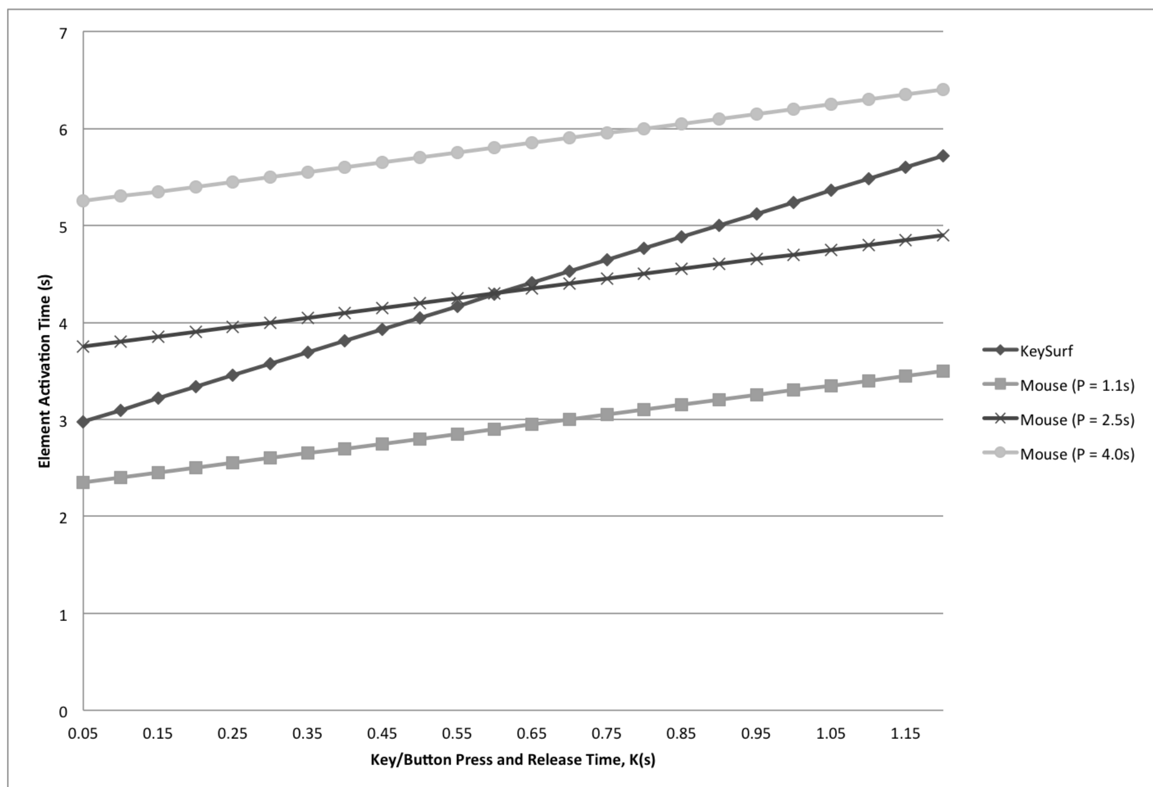

Figure 5.10: Element activation time for KeySurf and mouse navigation.

$$\begin{aligned} M + n_K(K + M) + K &< M + P + K \\ P &> n_K(K + M) \end{aligned} \tag{5.11}$$

$$\begin{aligned} P &> 1.38(K + M) \\ &\mid n_K = 1.38, n_{ID} = 2.07 \end{aligned} \tag{5.12}$$

It can be seen that as typing (and button click) time varies along the standard range, KeySurf performance decreases more quickly than the performance of mouse navigation. For a pointing time of 2.5 seconds, mouse navigation is less efficient until typing requires longer than 0.6 seconds, after which the mouse performs better.

5.6 Discussion

The underlying assumption in our study design is that the Web browsing patterns of non-disabled study participants are similar to those of people with physical disabilities. If this assumption does not hold, the generalizability of results to KeySurf users with a disability may be limited. However, we believe that if browsing behaviour is indeed different for persons with motor disabilities, it may be due to the limitations of their Web navigation systems themselves, rather than a difference in how they would like to browse the Web.

By modelling the available keyboard navigation systems and informing those models with results from our study, we can gain insight into what users may benefit from a given system. For users with slower typing rate, KeySurf's low absolute keystroke cost will provide an advantage, while users capable of faster input may be better served by an ID navigation system requiring fewer cognitive steps. In future work, a direct comparison between these two systems by persons with motor disabilities should be performed.

Chapter 6

Conclusions and Future Work

Computer users with a disability should be able to access the Web regardless of what input device they are able to operate. Users with more severe motor disabilities may require several seconds to press each key, thus requiring that a keyboard Web navigation method must not only be functional, but also efficient in order to reduce the effort of accessing the Web to a manageable level. In this work I have presented a new keyboard navigation system for the Web called KeySurf. I have described how KeySurf is efficient in terms of keystrokes per selection (33% fewer keystrokes per selection than ID navigation), intuitive ("type where you want to go"), largely independent of Web authors (only keyboard focus required), and minimally impacting on Web page design or aesthetics. It is detailed how the user centric search algorithm is used in combination with user history and interest indicators to estimate which page elements are more likely to be selected by the user, and then make these elements easier to select.

Through evaluation by simulation, I have shown that even for random element activation, KeySurf is on par with other efficient keyboard navigation techniques such as ID navigation. When evaluated on real Web sessions, it was observed that the user centric search technique reduces the mean number of keys required to select elements, and the consideration of user history reduces it further. The evaluation with people with disabilities, though limited, shows that the operation of KeySurf can be learned quickly, and that for users with slow or inaccurate cursor control, KeySurf can be faster than selecting elements with a pointing device.

It would be of interest to directly compare ID navigation and KeySurf for people with disabilities. The keystroke level models that have been presented of each keyboard navigation system represent an error-free model, which is not realistic for

many persons with motor disabilities. Comparing the types of errors that occur with each technique, the cost of error recovery, and whether the KeySurf features for error prevention are effective should be the focus of a future study.

A beta version of KeySurf has been posted on the CanAssist Web site since June 2009. This version does not use the keyword indicators of interest for privacy reasons, but retains the user centric search and history prioritization features. As of Dec 2011, KeySurf has been downloaded approximately 250 times. The individual who inspired KeySurf has been using it to access the Web since 2007. In order to move beyond a research tool and support more dynamic Web applications, the key features of KeySurf are now being ported to an open source keyboard navigation system called Gleebox [1].

The following items are left for future work:

- Develop an optical character recognition module to recognize the text on image buttons for use as the KeySurf label.
- Implement a keyword extraction algorithm either within KeySurf or on an external server in order to realize more control over the keyword extraction process.
- Optimize the calculation of the element score, or provide an option to disable it for users with higher rate keyboard input.
- Continue development of the switch controlled variant of KeySurf with directly assigned binary codes (from Section 3.9.3).
- Expand KeySurf's support of third party browser plugins and script-defined actions.

Appendix A

Data Logged by Passive KeySurf

Data logged by the instrumented passive KeySurf during a browsing session contains the following types of events and associated information.

Browser Started The start of a Web browsing session.

- **Time stamp** - Milliseconds since epoch.

Page load Occurs when a Web page is fully loaded in the browser.

- **Time stamp**
- **URL** - One-way hash of the full URL that was loaded.
- **Domain** - One-way hash of domain that was loaded.
- **Number of elements** - Number of active (selectable) elements on this page.
- **Window dimensions** - Size of the browser window.

Key event - Any keys pressed during the browsing session.

- **Time stamp**
- **URL**
- **Key type** - One of:
  - **Control** - Such as Backspace, Tab, Ctrl, Alt, etc.
  - **Alphanumeric** - Any alphanumeric character.

Scroll event - Scrolling action on the page.

- **Time stamp**
- **URL**
- **Scroll distance** - How far down the page is scrolled.
- **Page height** - The height of the entire page.

Mouse click - A mouse click on the page.

- **Time stamp**
- **URL**
- **Position** - X and Y coordinates of the click.
- **Tag name** - The HTML name of the element that was clicked.

Link click - A link was clicked.

- **Time stamp**
- **URL**
- **Position** - X and Y coordinates of the element.
- **Size** - Width and height of the linked element.
- **On-screen elements** - Number of elements on-screen when the click occurred.
- **Number of matches** - Number of matched elements when selecting this element with KeySurf after the first, second, and subsequent characters.
- **Shortcut used** - Whether a numeric shortcut was necessary to select this element.
- **Total score** - Total user interest score of this element.
- **Visited** - Whether this link has already been visited by the user.

Appendix B

List of WebSpeak Keywords

The recognition set for WebSpeak consists of the NATO Phonetic Alphabet for letters A through Z with some modifications (in bold), the numbers zero through nine, and some additional browser commands as listed below.

Letters Alpha, Bravo, Charlie, Delta, **Elephant**, Foxtrot, Golf, Hotel, India, Juliet, **Kingdom**, Lima, Mike, November, Oscar, **Piano**, **Question**, Romeo, Sierra, Tango, Uniform, **Victoria**, Whiskey, Xray, Yankee, Zulu.

Numbers Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine.

Commands Yes, Delete, Select, Spell, Space, Scroll Up, Down, Back, Forward, Favourite.

Appendix C

JavaScript Huffman Code Implementation

```
// object representing one node in the binary tree
function huffmanNode(element, prob) {
  this.elementRef = element;
  this.probability = prob;
  this.parent = null;
  this.zeroChild = null;
  this.oneChild = null;
  this.code = computeCode;

  // compute the codeword for the current node
  function computeCode() {
    var codeword = Array();
    var parentNode = this.parent;
    while(parentNode) {
      if(parentNode.zeroChild == this)
          codeword.push(0);
      else
        codeword.push(1);
      parentNode = parentNode.parent;
    }
    codeword.reverse();
```

```
    return codeword;
  }
}

// gets the root node with the smallest probability
// as long as that node is not excludeNode
function getSmallestProb(baseNodes, excludeNode) {
  var min = 1;
  var minNode = null;
  for(var i = 0; i < baseNodes.length; i++) {
    var rootNode = getRootNode(baseNodes[i]);
    if(rootNode != excludeNode && rootNode.probability < min) {
      min = rootNode.probability;
      minNode = rootNode;
    }
  }
  return minNode;
}

// gets the root node for the current subtree
function getRootNote(baseNode) {
  if(!baseNode.parent) {
    return baseNode;
  }
  else {
    return getRootNode(baseNode.parent);
  }
}

// remove purged elements from the tree
// nodes is an array of coded huffmanNode objects
// toRemove is an array of element references
function purgeElements(nodes, toRemove) {
  for(var i = 0; i < toRemove.length; i++) {
    var removeNode = null;
```

```
    var removeIndex = -1;
    // find the element in the node list
    for(var j = 0; j < nodes.length; j++) {
      if(nodes[j].elementRef == toRemove[i]) {
        removeNode = nodes[j];
        removeIndex = j;
        break;
      }
    }
    if(removeNode) { // break ties
      if(removeNode.parent.zeroChild == removeNode) {
        removeNode.parent.zeroChild = null;
      }
      else if(removeNode.parent.oneChild == removeNode) {
        removeNode.parent.oneChild = null;
      }
      nodes.splice(removeIndex, 1);
    }
  }
  return nodes;
}

// build the code tree on the array of huffmanNodes
function huffmanencode(elementNodes)
  var done = false;
  // build the code tree
  while(!done) {
    var firstSmallestNode = getSmallestProb(elementNodes, null);
    var secondSmallestNode = getSmallestProb(elementNodes, firstSmallestNode);
    if(!secondSmallestNode) {
      done = true;
    }
    else {
      var parentNode = new huffmanNode(null,
        firstSmallestNode.probability+secondSmallestNode.probability);
```

```
      parentNode.zeroChild = firstSmallestNode;
      parentNode.oneChild = secondSmallestNode;
      firstSmallestNode.parent = secondSmallestNode.parent = parentNode;
    }
  }
  return elementNodes;
}
```

Bibliography

[1] Ankit Ahuja and Sameer Ahuja. Gleebox release notes. http://thegleebox.org/releases.html, Dec 2011.

[2] Ankit Ahuja and Sameer Ahuja. Gleebox. http://thegleebox.org, Sept 2011.

[3] Dey Alexander and Scott Rippon. University website accessibility revisited. http://ausweb.scu.edu.au/aw07/papers/refereed/alexander/paper.html, Dec 2011.

[4] Hooman Baradaran. Using IDs to improve web navigation with a keyboard. Master's thesis, York University, 2009.

[5] Robert Bernstein. More than 50 million Americans report some level of disability . http://www.census.gov/Press-Release/www/releases/archives/aging_population/006809.html, Aug 2008.

[6] Jeff A. Bilmes, Xiao Li, Jonathan Malkin, Kelley Kilanski, Richard Wright, Katrin Kirchhoff, Amarnag Subramanya, Susumu Harada, James A. Landay, Patricia Dowden, and Howard Chizeck. The vocal joystick: A voice-based human-computer interface for individuals with motor impairments. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, Vancouver, Oct 2005.

[7] Brett Clippingdale. The Hawking toolbar. http://www.clippingdale.com/accessibility/hawking/status.html, Dec 2011.

[8] Robert W. Burnham and Sidney M. Newhall. Color perception in small test fields. *Journal of the Optical Society of America*, 43(10):899–902, Oct 1953.

[9] Ron Van Buskirk and Mary LaLomia. A comparison of speech and mouse/keyboard GUI navigation. In *Conference Companion on Human Factors in Computing Systems*, CHI '95, page 96, New York, NY, USA, 1995. ACM.

[10] Alastair Campbell. Access keys. `http://www.nomensa.com/blog/2004/access-keys/`, Dec 2011.

[11] canassist.ca. CanAssist Dynamic Keyboard. `http://canassist.ca/dynamickeyboard/`, Oct 2007.

[12] Stuart K. Card, Thomas P. Moran, and Allen Newell. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23:396–410, Jul 1980.

[13] Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1983.

[14] Kevin Christian, Bill Kules, Ben Shneiderman, and Adel Youssef. A comparison of voice controlled and mouse controlled web browsing. In *Proceedings of the 4th International ACM Conference on Assistive Technologies*, Assets '00, pages 72–79, New York, NY, USA, 2000. ACM.

[15] Mark Claypool, David Brown, and Makota W. Phong. Inferring user interests. *IEEE Internet Computing*, Nov-Dec 2001.

[16] cmusphinx.org. CMU Sphinx: The Carnegie Mellon Sphinx Project. `http://cmusphinx.org`, Oct 2007.

[17] conkeror.mozdev.org. Conkeror. `http://conkeror.mozdev.org`, Feb 2008.

[18] Cliff Cunnington. The Links WWW text browser. `http://links.sourceforge.net`, Aug 2008.

[19] e-Government Unit. Building in universal accessibility + checklist. `http://archive.cabinetoffice.gov.uk/e-government/resources/handbook/html/2-4.asp`, Aug 2008.

[20] Jeremy Goecks and Jude Shavlik. Learning users' interests by unobtrusively observing their normal behavior. In *Proceedings of the 5th International Conference on Intelligent User Interfaces*, IUI '00, pages 129–132, New York, NY, USA, 2000. ACM.

[21] gok.ca. Gnome on-screen keyboard. `http://www.gok.ca`, Nov 2006.

[22] Morten Goodwin, Deniz Susar, Annika Nietzio, Mikael Snaprud, and Christian S. Jensen. Global web accessibility analysis of national government portals and ministry web sites. *Journal of Information Technology and Politics*, 8(1):41–67, 2011.

[23] Stephanie Hackett, Bambang Parmanto, and Xiaoming Zeng. Accessibility of internet websites through time. In *Proceedings of the 6th International ACM SIGACCESS Conference on Computers and Accessibility*, Assets '04, pages 32–39, New York, NY, USA, 2004. ACM Press.

[24] hah.mozdev.org. Hit-a-Hint. `http://hah.mozdev.org`, Feb 2008.

[25] Foad Hamidi, Leo Spalteholz, and Nigel Livingston. Web-speak: A customizable speech-based web navigation interface for people with disabilities. 23rd International Technology and Persons With Disabilities Conference, Mar. 10-15, 2008.

[26] V. L. Hanson, J. P. Brezin, S. Crayne, S. Keates, R. Kjeldsen, J. T. Richards, C. Swart, and S. Trewin. Improving web accessibility through an enhanced open-source browser. *IBM System Journal*, 44(3):573–588, 2005.

[27] Charles T. Hemphill and Philip R. Thrift. Surfing the web by voice. In *Proceedings of the 3rd ACM International Conference on Multimedia*, MULTIMEDIA '95, pages 215–222, New York, NY, USA, 1995. ACM.

[28] Apple Inc. Accessibility averview. `https://developer.apple.com/library/mac/\#documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXDeveloping/OSXAXDeveloping.html`, Dec 2011.

[29] Infogrip. Roller trackball. `http://www.infogrip.com/products/mice/trackballs/`, Dec 2011.

[30] Anders Sewerin Johansen, John Paulin Hansen, Dan Witzner Hansen, Kenji Itoh, and Satoru Mashino. Language technology in a predictive, restricted on-screen keyboard with dynamic layout for severely disabled people. In *Proceedings of the 2003 EACL Workshop on Language Modeling for Text Entry Methods*, TextEntry '03, pages 59–66, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[31] S. Keates, P. J. Clarkson, and P. Robinson. Developing a methodology for the design of accessible interfaces. In *Proceedings of 4th ERCIM Workshop on User Interfaces for All*, pages 1–15. ERCIM Press, 1998.

[32] David Kieras. Using the keystroke-level model to estimate execution times. Technical report, University of Michigan, 1993.

[33] konqueror.org. Konqueror web browser. `http://konqueror.org`, Nov 2006.

[34] lynx.isc.org. Lynx source distribution directory. `http://lynx.isc.org`, Aug 2008.

[35] madentec.com. Discoverpro. `http://www.madentec.com/products/discoverpro.php`, Nov 2006.

[36] Jennifer Mankoff, Anind Dey, Udit Batra, and Melody Moore. Web accessibility for low bandwidth input. In *Proceedings of the 5th International ACM Conference on Assistive Technologies*, Assets '02, pages 17–24, New York, NY, USA, 2002. ACM Press.

[37] Microsoft. Accessibility. `http://msdn.microsoft.com/en-us/library/windows/desktop/bb545462.aspx`, Dec 2011.

[38] mozilla.org. Find as you type. `http://www.mozilla.org/access/type-ahead`, Nov 2006.

[39] Paul Nisbet and Patrick Poon. Special access technology, 1998.

[40] Sharon Oviatt. Interface techniques for minimizing disfluent input to spoken language systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, pages 205–210, New York, NY, USA, 1994. ACM.

[41] Janet Owens and Susan Keller. Multiweb: Australian contribution to web accessibility. In *Proceedings of the 9th Australasian Conference on Information Systems*, 2000.

[42] D. Rowell, G. F. Dalrymple, and J. Olsen. UNICOM: A universal communication and control system for the non-verbal motor impaired. *SIGCAPH Computers and the Physically Handicapped*, (24):56–59, 1978.

[43] Martin Schrepp. On the efficiency of keyboard navigation in web sites. *Universal Access in the Information Society*, 5(2):180–188, 2006.

[44] Martin Schrepp. Goms analysis as a tool to investigate the usability of web units for disabled users. *Universal Access in the Information Society*, 9:77–86, Mar 2010.

[45] Martin Schrepp and Patrick Fischer. A GOMS model for keyboard navigation in web pages and web applications. In Klaus Miesenberger, Joachim Klaus, Wolfgang Zagler, and Arthur Karshmer, editors, *Computers Helping People with Special Needs*, volume 4061 of *Lecture Notes in Computer Science*, pages 287–294. Springer Berlin Heidelberg, 2006.

[46] G F Shein. *Towards Task Transparency in Alternative Computer Access: Selection of Text Through Switch-Based Scanning*. PhD thesis, University of Toronto, 1997.

[47] Sensory Software. The Grid - computer control. `http://www.sensorysoftware.com/computercontrol.html`, Dec 2011.

[48] Leo Spalteholz, Kin Fun Li, and Nigel Livingston. Generating efficient labels to facilitate web accessibility. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 1319–1320, New York, NY, USA, 2007. ACM.

[49] Leo Spalteholz, Kin Fun Li, and Nigel Livingston. Efficient navigation on the world wide web for the physically disabled. In *Proceedings of the 3rd International Conference on Web Information Systems and Technologies*, pages 321–326, March 3-6, 2007.

[50] Leo Spalteholz, Kin Fun Li, Nigel Livingston, and Foad Hamidi. Keysurf: a character controlled browser for people with physical disabilities. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 31–40, New York, NY, USA, 2008. ACM.

[51] C. Stephanidis, A. Paramythis, C. Karagiannidis, and A. Savidis. Supporting interface adaptation in the AVANTI web browser. In *Proceedings of the 3rd ERCIM Workshop on User Interfaces for All*, pages 17–24, Alsace, France, 1997. ERCIM.

[52] Constantine E. Steriadis and Philip Constantinou. Designing human-computer interfaces for quadriplegic people. *ACM Transactions on Computer-Human Interaction*, 10(2):87–118, 2003.

[53] Hironobu Takagi and Chieko Asakawa. Transcoding proxy for nonvisual web access. In *Proceedings of the 4th International ACM Conference on Assistive Technologies*, Assets '00, pages 164–171, New York, NY, USA, 2000. ACM.

[54] Shari Trewin, Simeon Keates, and Karyn Moffatt. Developing steady clicks:: a method of cursor assistance for people with motor impairments. In *Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility*, Assets '06, pages 26–33, New York, NY, USA, 2006. ACM.

[55] Shari Trewin and Helen Pain. Keyboard and mouse errors due to motor disabilities. *International Journal of Human-Computer Studies*, 50:109–144, Feb 1999.

[56] Vivienne Trulock. A comparative investigation of the accessibility levels of Irish websites. Master's thesis, Napier University, 2006.

[57] w3.org. Web accessibility initiative. `http://www.w3.org/WAI/`, Oct 2007.

[58] David J. Ward, Alan F. Blackwell, and David J. C. MacKay. Dashera data entry interface using continuous gestures and language models. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*, pages 129–137, New York, NY, USA, 2000. ACM Press.

[59] wats.ca. Accesskeys and reserved keystroke combinations. `http://www.wats.ca/show.php?contentid=43`, Aug 2008.

[60] wats.ca. Using accesskeys - is it worth it? `http://www.wats.ca/show.php?contentid=32`, Dec 2011.

[61] World Wide Web Consortium. Web content acccessibility guidelines 1.0. `http://www.w3.org/TR/WCAG10/#gl-device-independence`, Aug 2008.

[62] World Wide Web Consortium. Web content acccessibility guidelines 2.0. `http://www.w3.org/TR/WCAG20/#keyboard-operation`, Aug 2008.

[63] World Wide Web Consortium. XHTML 2.0 hypertext attributes module. http://www.w3.org/TR/xhtml2/mod-hyperAttributes.html#s_hyperAttributesmodule, Jul 2008.

[64] yahoo.com. Content analysis web services: term extraction. http://developer.yahoo.com/search/content/V1/termExtraction.html, Oct 2007.

[65] Louie Zhao, Jay Yan, and Kyle Yuan. Mozilla accessibility on Unix/Linux. In *Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A)*, W4A '05, pages 90–98, New York, NY, USA, 2005. ACM Press.