ChibiPoint: Accessible Pointing for Web Applications

Alexander Birch

April 6, 2014

Abstract

[Revisit this once Results section is finalized, and once Conclusions section is written]

Contents

Chapter 1

Introduction

1.1 Context and Motivation

Software is moving to the web. We now rely on 'web applications' for services such as shopping, banking, and social networking.

Many of these websites are designed to be used with mice, or equivalent pointing devices (such as touchscreens).

Many users cannot use mice to point in websites, due to input restrictions:

1. Some users are too disabled (due to conditions like Repetitive Strain Injury (RSI)) to use the input device provided to them
2. More suitable input hardware, such as bespoke accessibility hardware, can be hard to prescribe (due to lack of knowledge, or access to a needs assessor) or procure (due to rarity or expense)
3. Some devices (such as Smart TVs or games consoles) inherently enforce a standard, limited, input hardware (such as a remote or gamepad) in stead of a better pointing device

Such users are denied full access to a web that relies on a robust pointing mechanism.

Attempts have been made to push a standard for mouseless navigation on the web: tabbing navigation. Unfortunately this mechanism is ineffective, unpredictable and inefficient.

It is challenging to optimise websites to accommodate tabbing navigation. In practice the web remains inaccessible, despite the availability of standards for accessible development.

A more detailed background is disclosed in Chapter [2].

1.2 Literature & Technology Review

We confirm here the need for an alternative pointing mechanim, and explore the approaches that have been attempted already.

We review what factors exist in designing for accessible keyboard control. We assess the state of keyboard accessibility, in general and for the web. We explore also general web accessibility, and how the problem it presents has been addressed up until now. We explore the vectors through which accessibility can be added to the web. We ascertain whether there exist any novel approaches to keyboard pointing. We review whether accessibility standards are capable of solving the problem of

keyboard pointing in the web, and finally look for experimental methods to evaluate the performance of keyboard pointing systems.

We conclude that the root of the problem is that standards push a navigation mechanism that is not suited to the task of pointing. We decide that a new mechanism could be delivered in the form of a web browser extension.

1.3 Problem statement and Hypotheses

1.3.1 Problem statement

Existing work focuses around making websites conform to accessibility standards. Yet even using tabbing navigation within a tab-accessible website is not an ideal browsing experience.

The pointing mechanism provided by 'tabbing navigation' is inherently limited, so **a more suitable mouseless pointing mechanism needs to be developed: one that is designed to cope with real websites, rather than idealized ones.**

To solve mouseless pointing on a wide variety of input hardware, a mechanism is required that maps well to that wide variety of input hardware. One that is effective, predictable and efficient.

1.3.2 Hypotheses

We believe that our keyboard navigation system for the web, ChibiPoint, provides an improved mechanism for pointing as compared to tabbing.

Hypothesis 1: ChibiPoint is generally more efficient at web navigation than tabbing

We hypothesise that our system, **ChibiPoint, requires significantly fewer keypresses to navigate webpages, than does 'tabbing navigation'** (the current standards-prescribed method for keyboard navigation). 'Navigate webpages' is a broad term, as there are many classes of pointing that need to be analysed. The more detailed hypothesis is that, **given several distinct classes of pointing, ChibiPoint requires fewer (or equal) keypresses, for a large majority (>80%) of these scenarios**. Since we analyse two versions of ChibiPoint, it should be understood that we require at least one of these to outperform tabbing in a majority of tasks.

Hypothesis 2: ChibiPoint's 'flyouts' feature, reduces further the required keypresses for web navigation

We hypothesise that the novel 'flyouts' feature of ChibiPoint, which assigns hotkeys to suggested buttons on the webpage, contributes to reducing the number of keypresses required by ChibiPoint for web navigation, compared to using just its standard pointing method of hierarchically drilling through the page and pointing at elements using crosshairs. In other words, we hypothesise that ChibiPoint 'with crosshairs and flyouts enabled' performs a large majority (>80%) of pointing scenarios in fewer keypresses than ChibiPoint 'with just crosshairs enabled'.

1.4 Goals and Methods

1.4.1 Goals

We intend to demonstrate that a more effective, efficient, and predictable mouseless pointing mechanism than 'tabbing navigation' can be made for the web. The mechanism needs to be capable of achieving widespread availability — across many browsers and input systems.

The input mechanism used by the web browser extension should be input hardware-agnostic: that is, it must accept as input a language that can be mapped to by a wide variety of input methods.

1.4.2 Methods

Deliverable

We will produce a proof-of-concept web browser extension (for Google Chrome), that provides mouseless pointing in a variety of real-world websites. The implementation will recruit only technologies that are available to all web browsers (i.e. HTML5, CSS and Javascript). This foundation enables it inherently to be completely cross-platform and cross-browser. Any external libraries used must be free in cost, so as not to impose a price barrier to availability.

Supported input

The input language for the system will be an instance of symbol-based input: keystroke sequences. Thus a keyboard-based interface theoretically serves as a proof-of-concept for other input systems that can produce a vocabulary of discrete inputs, for example gestures or spoken words. The more limited this proof-of-concept's vocabulary is, the more devices it allows to map to its space.

Qualitative difference

We will improve upon the efficiency of tabbing navigation's linear trawl, by using a hierarchical traversal.

We will solve tabbing navigation's effectiveness problems: we avoid falling into 'focus traps' by avoiding a relative traversal. Additionally we solve the feedback problem represented by 'lack of visual indicator of focus', by painting our own visual indicators on elements.

We will improve predictability of navigation by using a spatial traversal rather than tabbing navigation's markup-driven traversal.

1.5 Results / Contributions

Both our hypotheses are confirmed: ChibiPoint provides a significantly more efficient pointing mechanism than tabbing. The novel 'flyouts' feature contributes further to ChibiPoint's efficiency.

We contribute a widely portable mechanism — ChibiPoint recruits in its development only widely-available web technologies. The system is compatible with many 'real-world' websites that expect mice. Some implementation bugs exist, and these are felt to be fixable.

ChibiPoint demonstrates a navigation mechanism whose input vocabulary is a simple symbol sequence. By showing an instance of its application with keyboard input, we imply that the mechanism is compatible also with other point systems that can produce symbol output, such as gesture control or speech. Thus its efficient navigation could be extended to other mouseless devices like Smart TVs.

[Revisit this once Results section is finalized, and once Conclusions section is written]

1.6 Thesis overview

The remainder of the thesis is composed as follows:

Chapter [2], *Detailed Context & Motivation*, discloses more detail around the background and motivation for this work.
Chapter [3], *Literature & Technology Review*, confirms the need for an alternative web pointing mechanism, and explores the approaches recommended, as well as reviewing what solutions have been attempted already.
Chapter [4], *Requirements & Design*, details the design of our system and pointing mechanism.
Chapter [5], *Implementation*, describes how we built our web browser extension.
Chapter [6], *Evaluation*, chronicles the output of a usability study and a larger quantitative study. We determine how our web browser extension compares to tabbing navigation, and determine also the contribution of our novel algorithm for suggesting 'clickables'.
Chapter [8], *Conclusions & Future Work*, discusses our contribution to the field, and ways to extend the work.

Chapter 2

Detailed Context & Motivation

[@Fabio: this whole chapter is copypasta from the original Project Proposal. I've not proofread it, so cannot guarantee if it's consistent with the current problem I say I am trying to solve, or the solution I propose.]

2.1 Problem

2.1.1 Problem Context

Some domain knowledge will be divulged in this section, prior to describing the problem under investigation. Namely, the case for accessible navigation alternatives' necessity, and also the relevance of the web in contributing to the size of the problem.

Need for accessibility Computer users are not uniform in their capabilities and preferences. For this reason, users differ in the input mechanisms they choose for interacting with software. Some users may prefer to perform tasks using a keyboard rather than using a mouse, for example. Commonly mouse and keyboard will be recruited together. However some users' input strategies are constrained by factors such as disability: occupational syndromic conditions such as RSI can preclude the ability to use a mouse, keyboard or both [1]. RSI and related disabilities are growing in prevalence, representing by some estimates 22% of people [2]. This disability accounts for one third of workers' compensation costs in the US private industry [3].

When the user cannot or will not use the primary input scheme of the software, it becomes necessary to provide alternative input modes. For simple interfaces that use standard components, keyboard and mouse can be both supported implicitly without additional development effort. Conversely, complex interfaces or ones involving non-standard components require explicit treatment from the developer to ensure that all funcitonality can be accessed. In cases where accessibility is not a priority or concern of the software developer, users can be left unable to use said software.

Rise of web applications Even as early as 2001, software made the leap to the web [4]. Services such as email, banking and word processing are being produced as web applications, due to a few advantages:

- high cross-platform, cross-device support & penetration

- centralized maintenance and upgrade
- centralized storage of user data
- (consensual) monetization of user information, telemetry
- scalability (down and up)
- zero end-user installation
- redundancy

Email clients such as Outlook [5] benefit from this; emails can be accessed from any web-enabled device without any need for installation, all users can be kept on the latest version without effort on their part, and user data can be analyzed to improve content filters or provide relevant advertisement. At peak times of operation, more servers can be recruited, and vice-versa, such that the cost of operations can be finely controlled based on usage.

2.1.2 Problem Description

This transition away from native applications is leading to sacrifices in accessibility. Web applications run within a native application (the browser), so there is conflict when it comes to the following accessibility paradigms:

- menu actions
- keyboard shortcuts
- tab order
- citizenship amongst other applications

Even on the conceptual level, there are problems with using the web to serve applications: a web application tries to fit a full application interface inside a web browser's (already saturated) interface. It is a second-class citizen; a user cannot 'switch application' to it, only trawl for it amongst the tabs and windows of some web browser.

Additionally, web applications are often guilty of foregoing standard UI components in favour of a bespoke UI made in JavaScript or CSS. This can be used to deliver impressive effects like transitions, or serve complicated shapes of interactive content or buttons. But this freedom makes it all too easy to produce inaccessible elements, such as buttons that can't be pressed or focused using the keyboard.

2.1.3 Existing Approaches

There exist at present a few categories of solution to the problem of web accessibility:

- following web development guidelines to produce a standards-compliant website
- using web technologies to create a non-standard interaction behaviour
- making the web browser behave in a non-standard manner

Recommended web practice

Guidance is provided on how to develop accessible web applications, in the form of the World Wide Web Consortium (W3C)'s Web Content Accessibility Guidelines (WCAG) [6] [7]. A description and analysis on its recommendations for keyboard access follows:

Pointing with access keys Access keys are one paradigm used for keyboard accessibility in the HTML5 specification [8]. Hotkeys can be assigned for the focusing of interface elements, used in conjuction with modifier keys. This system allows the user to jump instantly between distant or unrelated interface elements. However, its usefulness is predicated on the discoverability and availability of said key bindings.

It is hard to avoid conflicts with existing shortcuts used by the user's accessibility technologies, since there is no standardisation between browsers as to which modifier keys to use [9]. Recommendation exists that access keys be avoided altogether for this reason. Another problem with access keys is that they rely on explicit implementation by developers. They also require learning of a layout, and although attempts have been made at increasing standardisation [10], that standardisation is not prevalent.

Making interface tab-accessible It is difficult to develop a tab-accessible web application [11]. Since the only pointing expression assumed is the tab key, layout must be linearized. Not only is this hard to design and develop for (requiring explicit effort and markup changes), the end result is not that impressive; content that is far away in the tab order unavoidably requires repeated hammering of the tab key (making it far harder to reach with the keyboard than with a pointing device). Plus linearization doesn't suit web content, which often has navigation bars, side bars, feeds or some similar multi-dimensional layout.

Short-circuiting tab journey with skip-links An exception to the content linearization problem is skip links, which can short-circuit the tab journey around a page by allowing the user to choose alternative insertion points. These again require effort and understanding from the developer. It is hard to provide a suitable set of skip links, and again the best case is still a hamfisted pointing experience, where the user can't predict where they are going to or from.

Conclusion Even a website designed with accessibility standards in mind can be quite hard to navigate; web standards prescribe only a very limited amount of keyboard expression, so meeting these provides a very limited experience.

Using web technologies to augment accessibility

Whilst it is possible (to an extent) to create key bindings in web content (through access keys section 2.1.3, or through JavaScript event libraries like MouseTrap [12]), for the most part web applications do not make use of this. Possible reasons for this are:

- no agreed standard exists for how websites should provide keyboard shortcuts
- web application shortcuts are likely to conflict with native application's shortcuts
- existing native application shortcuts are hard to tiptoe around, since any browser or extensions could be used
- some devices (for example smartphones) do not have the same expressive keyboard capability
- would only be available on those websites that chose to implement it
- websites not guaranteed to implement in a uniform way

Changing the web browser

Adding accessibility on top of web content can be achieved through a web browser extension. This can change the way content is treated so as to work better for the preferred input mode. In some implementations, the browser is extended through the use of web technologies, which can coexist with the content of the website. One such example is Type-Ahead-Find [13], which allows users to move keyboard focus to hyperlinks by typing the content of said hyperlink. Another example is Stylish [14], which allows users to redesign a website to suit their needs.

Web browser extensions solve some of the problems associated with using web technologies alone to augment accessibility: key bindings are uniform across the user's browsing experience, so a user can configure non-conflicting shortcuts. The solution is also easy to distribute via web browser extension repositories.

2.2 Proposed Solution

[@Fabio: (again) this whole chapter is copypasta from the original Project Proposal. I've not proofread it, so cannot guarantee if it's consistent with the current problem I say I am trying to solve, or the solution I propose.]

2.2.1 Approach

Half the problem is the limited amount of expression afforded to the keyboard; if tabbing is the only navigation method offered, then there is a severe constraint on what can be achieved. A means for utilising more keys is necessary. Unfortunately websites cannot provide this, as they cannot predict which key bindings will be free on the user's computer. Nor should they try: each website would have to be learned individually.

The burden of reserving key bindings should fall on the web browser, or an extension thereof. Here it can be controlled and configured, as well as uninstalled should it cause conflicts. A novel shortcut scheme would be used to avoid conflicts, such as sequences rather than chords, or invoking the extension explictly as a precursor to action.

The rest of the problem is that accessibility is presently found only on websites that design for it. It is easy to blame developers for making websites that only work with a mouse or touchscreen, but the real problem here isn't that the website is designed wrong, it is that the keyboard isn't powerful enough to cope with these pointing situations.

Again, a solution where the keyboard is augmented via the browser is desirable. Increasing keyboard control diminishes the number of problem situations that exist for it.

What is novel about the approach?

Previous attempts (such as suggested best practice, described in section 2.1.3) at introducing keyboard accessibility to web applications have been focused on redesigning the website. The novelty of our approach is that we redesign instead the manner of traversing the website. Pointing will be designed from the ground-up to expect and support complex non-linear layouts using non-standard components.

Why does it make sense?

The approach embraces the way the world is making web applications (complex, bespoke, mouse-optimized interfaces), and attacks the problem that necessitated non-standard interfaces: that web applications are second-class citizens, described as 'content' in a pane of a native application. The pointing system will exist inside the context of the web application rather than the native application, and will acknowledge the web interface as the highest-level citizen. This makes for a clearer paradigm.

2.3 Methodology

2.3.1 Deliverables

The goal of this work is to produce an accessibility layer to sit between a web browser and web content, which would augment the accessibility of said content. The primary purpose of the accessibility layer is to achieve effective pointing via a sequence of keystrokes.

The system will aim to solve better the pointing situations to which tab navigation [1] is less suited. Pointing contexts such as form navigation, at which tab navigation is satisfactorily effective, need not be a focus of the novel system. Ideally the novel system would co-exist with tab navigation, allowing the user to fall back on tab navigation in cases where it is suitably effective (or more effective).

Software solution

The choice of browser for implementation is immaterial; the same logic can be applied to any extensible browser. This work will concern itself with Google Chrome, but the choice is arbitrary. The accessibility layer will be implemented in the form of a web browser extension, as this meets the criteria of sitting between browser and content, persists between page navigations, and doesn't incur work on the website developer's end.

[1] Side-note: the term 'tab navigation' is used in this document always to refer to focusing interface elements via the tab key (sometimes known as 'tabbing navigation') — not to be confused with 'tab navigation', where a user switches browsing contexts represented as 'tabs' in their windowing system.

Chapter 3

Literature & Technology Review

[@Fabio: this whole chapter is copypasta from the original Literature Review. I've not proofread it, so cannot guarantee if it's consistent with the current problem I say I am trying to solve, or the solution I propose.]

3.1 Overview

The deliverable for this dissertation is anticipated to be a browser extension that adds accessibility onto web content, favouring a novel interaction approach over the experience prescribed as accessible by web standards.

The key points to explore are:

- What factors exist in accessible design for keyboard usage, in general and for the web?
- Have approaches to keyboard accessibility evolved over time?
- Has the status of keyboard accessibility evolved over time?
- Does literature support the notion that 'keyboard accessibility in the web is generally lacking'?
- Does work exist around making web content more accessible for the keyboard?
  - Has web accessibility changed over time?
  - Does work exist around adding extra accessibility to web content without the web developer's involvement?
    - Does this work aim to bring content in line with a suggested web standard, or is a novel approach preferred?
  - Does work exist around adding extra accessibility to web content via the user agent?
- Do there exist any novel keyboard pointing methods?
- Is there any support for the notion that web accessibility standards prescribe a sub-par experience?
- Are there any experimental techniques that are recruited for evaluating task performance on keyboards?
  - Are there any comparisons with mouse performance?

A review of each of these points follows. References to 'accessibility' in general will be focused primarily around keyboard accessibility. The search effort around 'keyboard accessibility' was constrained to 'pointing with the keyboard'; accessibility concerns around other keyboard roles, such as typing, will not be discussed.

3.2 What factors exist in accessible design for keyboard usage, in general and for the web?

It is useful to have an understanding of what is meant by 'keyboard-accessible' design. Deng, 2001 lists the factors involved [15] [as cited in 16]:

- Using a logical tab order (using the tab key from the keyboard to navigate from link to link) mapped to the layout of the controls and the layout of information on the screen
- Using keyboard mapping for speeding up keyboard interaction and enhancing alternative input methods
- Avoiding conflicts with the operation of assistive software such as screen readers, and exploiting the built-in accessibility features of operating systems
- Providing multiple methods for access via the tab key as well as the use of shortcut keys
- Defining hot keys for more functionality for example, allowing the user to go backwards from link to link
- Ensuring that access keys and hot keys for frequently used functionalities are reachable using one hand, for people using one hand only
- Avoiding repetitive key presses that would be uncomfortable for users with repetitive strain injuries
- Placing frequently used links and functions on the first navigation level without requiring the user to navigate a lot to reach them

This makes some good points (particularly that tabbing isn't the only way to expose functionality via the keyboard; shortcuts help too).

As for which issues were prevalent in practice, a list of 'Top 20' accessibility concerns was published in 2005, which included observations of the following keyboard concerns from accessibility tests [17]:

- Users cannot access objects by keyboard.
- Hot keys are confusing, missing, or conflict with browser commands.
- Focus does not move to the right place (so availability of target remains unknown).
- There is no visual focus on the page.
- Users cannot tab to page elements in logical order.

It must be stressed that this list is not exclusive to keyboard accessibility, and yet a quarter of accessibility concerns in this shortlist pertained to keyboard usage, and of those the majority were pointing issues. This suggests either that websites tend to be developed in a way that makes keyboard pointing difficult, or that the mechanism provided for pointing is not very effective.

3.3 Have approaches to keyboard accessibility evolved over time?

It would be interesting to explore the original role of the 'tab' key, and verify whether it is still honored today. However, its history proved to be ill-documented. Tab's nature suggests that it is for form traversal: there is no directional control, nor any promise of a particular destination beyond 'next element in markup' – these factors are not a problem in form navigation, where the intention is to travel to the next field. As such it seems well-designed for at least this, but the aforementioned shortcomings are much more relevant in pointing scenarios, where the destination is spatial, having more complexity than 'the next control', and can be unrelated to the starting position of the search. If it is the case that

tab navigation was intended for form traversal, then it would not be surprising for it to be unsuited to general pointing. It is also easy to see how its use could have evolved from form navigation to whole-interface navigation; it provides a way to move keyboard focus to controls, so the temptation exists to purpose it as a general focus mechanism. On some level, this seems like accessibility, but on another level, it can be harmful; it pushes a possibly poorly-suited mechanism as the standard problem solution.

3.4 Has the status of keyboard accessibility evolved over time?

An early view (1995) is provided on the landscape of accessible computing by Bergman and Johnson [18]. It suggests (p.9) that "most interface style guides were not written taking users with disabilities into account". Whereas today, accessibility is featured in the Design Guidelines for many prominent software platforms: Apple references accessibility in its OS X Human Interface Guidelines[19], Microsoft provides literature on Designing Accessible Applications[20], and The GNOME Project provides Human Interface Guidelines for use with its desktop environment for Unix-like OSes[21].

Hendrix and Birkmire describe how, in 1998, the Mac versions of Internet Explorer and Netscape web browsers lacked any provision for selecting or activating web links without the mouse [22]. The Mac OS as a whole was also slated for not providing out-of-the-box support for accessing menus via the keyboard. These points are in stark contrast to today, where the default Mac browser, Safari, provides tab support by default[23], and the OS provides global keyboard shortcuts to access menus[24].

A landmark legal case in accessibility, Maguire v The Sydney Organizing Committee for the Olympic Games (SOCOG) [25, 26], 2000, made it clear that laws such as the Commonwealth Disability Discrimination Act 1992 (Cth DDA) could be used to enforce web accessibility, and that websites lacking provisions for the disabled could be defined as discriminatory, and thus be penalizable. SOCOG's defence, that the cost of accessible development constituted 'unjustifiable hardship', was rejected. Not only did the defendant have to redesign their website, they also had to pay damages to the plaintiff on top of this. This revelation that a lack of accessibility could be more costly than the perceived 'price' of accessible development.

3.5 Does literature support the notion that 'keyboard accessibility in the web is generally lacking'?

Literature suggests that "the current design of most Web sites makes ... efficient keyboard navigation nearly impossible" [27, p.1], and that "the design of ... web applications is highly optimized for users who navigate with a mouse" [27, p.1]. This could be due to lack of standards-compliance, as a 2004 study [28] revealed a large majority (81%) of websites fail to satisfy even the most basic checkpoints of the WCAG [6, 7]. The same study [28] showed through interviews that disabled users believed that most web sites of the time did not consider their specific needs. The problem of Web designers lacking knowledge of the requirements of disabled users has been reported for years [28] [29] [30]. This is understandable; the development of keyboard-accessible websites is considered to be challenging [11].

3.6 Does work exist around making web content more accessible for the keyboard?

As explored previously in section 2.1.3, guidance is provided on how to develop accessible web applications, in the form of the W3C's WCAG [6, 7]. However, the content itself is not the only factor; content is accessed via a user agent (ie a web browser), and the user experience of that agent is important too. The W3C provide recommendations on how to develop accessible user agents for browsing web applications[31, 32]. The User Agent Accessibility Guidelines recommends navigation methods for users, that could reduce the journey time for tab navigation. For example, Guideline 8 of the version 1.0 document, 'Provide navigation mechanisms', suggests that "User agents should allow users to configure navigation mechanisms (e.g., to allow navigation of links only, or links and headers, or tables and forms, etc.)" [31, p.17]. Certainly allowing a filter on which elements are navigated would reduce the amount of navigation actions. Indeed, as will be discussed in section 3.7, the Safari web browser allows the user to filter which types of focusable controls are traversed during tab navigation. However this filter choice is limited (only two modes are offered, and neither may be what the user really wants), and the feature is not equally available in other mainstream browsers (it is missing in Firefox, and Chrome's implementation buries it inside a preference [33]).

3.6.1 Has web accessibility changed over time?

Suggested standards have evolved, with two versions of the WCAG being published [6, 7], in 2001 and in 2008. Some noteworthy changes to keyboard use include [34]:

- The recommendation for provision of access keys has been redacted in the newer version.
- Focus traps (where the user can't move keyboard focus out of some interface area, such as an 'infinite-scrolling' list) are recognised to be a problem.
- Visual indicator of focus is now required for keyboard navigation.

Web accessibility as a research field is now growing quickly and increasing in diversity, despite few studies having been published until 2002 [35]. Freire, Goularte, and Mattos Fortes claim that this interest has been stimulated by a need for developers to address accessibility requirements. This chronology could perhaps be explained by the Maguire v SOCOG [25, 26], 2000 lawsuit referred to in section 3.4, after which accessible development gained some visibility.

3.6.2 Does work exist around adding extra accessibility to web content without the web developer's involvement?

In cases where the web developer is uninterested or uninformed about accessibility, they cannot be relied upon to introduce accessibility to their website. Kouroupetroglou, Salampasis, and Manitsaris presents a framework for annotating web-pages with semantic information about the role of content, to aid information access for disabled users of the Worldwide Web [36]. The paper is based on the idea of the Semantic Web [37], where structure and relationships of content are declared. The annotation framework relies on a community of users proposing markup for inaccessible web-pages, and uploading the annotation file to a public storage server. A bespoke web browser for blind users, SeEBrowser, loads the pages alongside their annotations, and uses the extra markup to provide structure-aware browsing shortcuts.

The advantage of divorcing the responsibility of accessible development from the web developers is that the onus is not imposed on someone who may have an incomplete understanding of the domain, and also that stakeholders are given the power to add accessibility themselves, should the official design be unsatisfactory. This is, of course, predicated on having an active community (since without annotations, no accessibility can be added). The approach may also be relevant to keyboard-accessibility, as blind users navigate via the keyboard. Certainly, providing shortcuts that allow a variety of manners in which to traverse content, gives keyboard users more control over their navigation. However, in mainstream browsers, the extra semantic markup would be less useful, as the only keyboard navigation method available is tabbing, which is considered to be inefficient even for well marked-up pages (discussed later in section 3.8).

3.6.3 Does work exist around adding extra accessibility to web content via the user agent?

Introducing accessibility via the user agent removes the need to change the way websites are developed. If existing websites do not meet content accessibility standards, the user agent may be able to compensate, by modifying the presentation of inaccessible content, or providing novel navigation methods that are robust to inaccessible page structure. For example, SeEBrowser (described previously in section 3.6.2) is a user agent that provides browsing shortcuts to aid blind users in navigating efficiently through various web page elements (such as content areas, navigational aids and functional elements) [38].

User agent customization has been used to modify Internet Explorer [39] to fit the needs of older users. This work aimed to aid vision impairments by zooming content and speaking text, aid cognitive impairments by simplifying layout, and help dexterity issues by providing large buttons, and an easy interface to change keyboard settings. The reason it was chosen that the work be an extension on Internet Explorer was that users preferred to use a standard browser with accessibility adaptations, rather than a specialized one with a limited set of features. Content transformations were performed in the user agent, as opposed to the original design which proposed to transform pages in a proxy placed before the client (though this would make the featureset available on all web browsers, it was found to be a non-viable architecture [40, 41, 42]). User preferences were stored server-side, so that they would be obtainable from different computers. Ultimately it was found that most features offered were used, but the majority used were those that made only minor presentation adjustments.

A subsequent study built on this work by implementing a similar user agent transformation for Firefox [43], making the improvements available cross-platform. It describes the software as being purposed for changing the user experience, rather than the web page compliance.

3.7 Do there exist any novel keyboard pointing methods?

A patent exists [44] describing an ‘intuitive’ method of focusing elements using the keyboard. It is spatial in nature, and allows the user to specify the direction of focus travel via the keyboard (rather than leaving this decision to the arbitrary directional order prescribed by page markup). Whilst this improves over standard tabbing by allowing a choice of direction, it still does not give an indication in advance of what will happen after the keypress.

The default Mac browser, Safari, supports multiple granularities of interface tabbing, differentiated by whether Alt is held down with Tab[23]. This allows a choice of whether to navigate between all focusable elements, or just form elements. This essentially gives the user a filtering mechanism, and

can shorten tab journeys where the only controls of interest are form controls (ie, forms). Chrome also provides two modes for tabbing, but they are accessessed via a setting toggle rather than given separate keybindings [33]. The result is that switching between them on a whim is not possible, so combining the use of the two modes in navigation is denied. The Mac OS itself also provides a global 'granularity' setting for tabbing, which can be toggled by shortcut between 'text boxes and lists only', or 'all controls'[24]. Again, the effect is that the tab journey can be made finer or coarser via a toggle, shortening journeys to certain elements. This system-wide toggle stacks with the effects of Safari's toggle, allowing for complex filters to be created. However, shortcomings are that the current state of the system-wide toggle has to be memorized to be able to predict effects, and as ever with tab navigation, the destination is unknown until after the action. Additionally, this setting is not honored by all browsers; the Chrome web browser, for example does not observe this preference[33]. As a solution, there are other impracticalities to consider: the setting affects interactions in all aspects of the OS interface, rather than just the application at hand, which may be undesirable. It is also questionable whether repeatedly changing system preferences just to complete navigation tasks within an application, is sensible from a design point of view; changes made in 'System Preferences' should be just that: preferences. A user can't 'prefer' both modes – rather, the choice is made based on current context, so a better implementation for these purposes is to allow both modes of navigation at all times, mapped to different key bindings (rather than toggling the mode used by one single key binding). In essence, this is what is achieved by Safari's 'Alt-Tab' shortcut.

MouseKeys is a feature that exists in Windows, Mac OS X and X Windows-based workstations. This is a system-level alternative pointing mechanism, that moves the mouse cursor and performs clicks, using keypresses. Naturally, being system-level allows it to work for all applications, and being an interface for operating the mouse cursor allows it to be treated by applications in the same way a mouse is. However it is considered time-consuming in comparison to keyboard navigation because it provides 'relatively crude directional control', and inefficient support for continuous motions like drag & drop [18].

Switch Scanning is a paradigm whereby input options are scanned in front of the user, who activates a switch whenceupon their desired option cycles into focus[22]. By allowing one key to perform many roles (dependent on time elapsed), it increases the expression of a keyboard shortcut. It is described as slow (naturally; it introduces a time factor to an otherwise-instant action). However, where time is not a concern, it could be harnessed to reduce the repetitive motor strain associated with repeated keyboard tabbing (a user could initiate cycled tabbing with one keypress, then stop it with one more, regardless of distance). For the purposes of this work though, which aims to improve on the speed of tabbing (not just the repetitive strain), switch scanning is likely too slow to be a solution.

TypeAheadFind (now Find As You Type), referred to previously in section 2.1.3, is not just a plugin for the Chrome browser, but also a feature distributed with Firefox as standard [45]. It streamlines for the user the process of selecting hyperlinks; any time keyboard focus is not within an input field, key input is directed to an as-you-type search, which focuses any matching string in the page. A selected hyperlink can easily be followed by pressing Enter. This is an efficient method for focusing hyperlinks, as text search can narrow down the target quickly, and without aiming. However the string needs to be sufficiently unique, to reduce cycling through options. Additionally, it is no use for selecting interface elements that are not defined by text, such as divs. It can also be a bad citizen if the web application in question expects to receive keyboard input when the user is not within a form (for example, in a game). This can be improved optionally by setting a single, uncommonly-used key to be used for invoking the feature.

One promising approach is 'MouseGrid', recruited by many speech-control systems. It exists within Windows Vista's speech recognition, is available as a plugin for Dragon NaturallySpeaking (made by Hippocampus), and is an official feature in Dragon Dictate. Once invoked, a grid is drawn, dividing

the screen into segments. Each segment of the grid is labelled with a number the user can say to 'drill down' into that segment. Upon doing so, a smaller grid is made within that grid segment, and the user is invited again to specify a segment to navigate towards. Ultimately the user tries to steer the grid closer to some button they wish to press. At all times, a click can be effected in the center of the most specific grid. This recursive split of the screen divides the search space of the screen logarithmically, and so enables the user to express interest in a huge variety of screen locations, using very few lookups. This approach uses a small vocabulary of speech inputs (just a grid of numbers 1-9, plus a 'click' command, and on/off). As such, it could be easily mapped to the keyboard. It could well provide an efficient means for specifying buttons on the web.

3.8 Is there any support for the notion that web accessibility standards prescribe a sub-par experience?

Current accessibility guidelines do not address the problem of efficient keyboard access [27]. Keyboard inefficiency can be severe enough that even a standards-compliant website may be, for practical purposes, unusable by keyboard users[27, 29, 46]. However, as discussed previously in section 3.6, current user agents do not necessarily comply with user agent standards as wholly as they could; the problem could therefore lie with the existing implementations of web browsers, rather than the standards prescribed.

3.9 Are there any experimental techniques that are recruited for evaluating task performance on keyboards?

Schrepp, 2006 models keyboard performance using the Goals, Operators, Methods, and Selection rules (GOMS) technique described by Card, Moran, and Newell [47]. It is used for predicting how long an experienced user needs to complete interface tasks. The technique has been used to compare efficiency of mouse and keyboard techniques [47, 48]. An overview exists of the different GOMS models [49]. GOMS reduces human-computer interaction to the sum of elementary actions (either motor, cognitive or perceptual). For example, pressing of a button, moving a cursor to a target, or perceiving the current position of the mouse cursor. Schrepp freely admits that some of the factors in keyboard navigation are hard to model with GOMS; accounting for the cognitive demand of orientation (that is, ascertaining the current position of keyboard focus) is challenging. Though measurements for this time penalty are known (1.35s) [50], the frequency with which this penalty is incurred is harder to estimate (in fact, no guess was ventured on this by the paper). The ultimate conclusion was that, irrespective of this unaccounted extra time penalty, mouse speed still far exceeded keyboard speed. This further supports the notion that there is a wide discrepancy between mouse and keyboard efficiency. However the author concedes that application of GOMS for disabled users was questionable to begin with, as standard predictions of motor time may be too optimistic; this has been estimated as high as 0.6s [51], rather than the 0.2s [50] predicted for able-bodied experts. For some cases, such as repetitive strain, it is possible that motor problems would worsen with use, making the time factor worsen as the experiment progresses. This increases the challenge of modelling times.

For the purposes of evaluating the planned deliverable (an alternative keyboard pointing mechanism), the main comparison to be made is whether the novel system improves upon tabbing; for this it suffices to count keypresses. This avoids the difficulty associated with modelling usage times for disabled users, since the number of keypresses is still a useful indicator of effort, which is agnostic of physical

condition. Time-based comparisons could still be made between keyboard and mouse, but perhaps selection of able-bodied users would help to increase the validity of the modelling.

3.10 Conclusions

Keyboard navigation of websites is a problem. Correcting the way people develop websites is challenging, and unrewarding; many websites don't observe standards (because the web developers don't attempt it, or even know how), and even those that do, do not necessarily provide an ideal experience. This is not necessarily a fault of the content guidelines; rather, it seems to be the fault of the navigation method prescribed by user agent guidelines, or the interpretation of said guidelines. A different user agent that provides more manners in which to traverse content with the keyboard, could be the real solution to accessible keyboard pointing. Annotations used by blind users could be used also by sighted keyboard users to inform of content structure, and by extension, navigation possibilities. User agent extension has been pursued before as a solution in favour of other mechanisms (such as content modification by proxy server). Thus it seems like the original proposal of augmenting keyboard pointing in the user agent via a browser extension, is indeed an approach worth pursuing.

Chapter 4

Requirements & Design

4.1 Requirements

Here we capture the main engineering goals of the software system to be produced. Its purpose is to paint a picture of our initial intentions; it is not intended as a rigid specification. An attempt is made to refrain from prescribing a specific implementation.

4.1.1 Non-functional requirements

These requirements have a high priority:

R1 High availability
Neither price nor platform should be a barrier to adoption of the software. A solution that works in many browsers would be most available.

R2 Good citizenship (input)
Bindings reserved for interaction with software should not conflict with those required by web browser, or (within reason) bindings that webpage will use.

R3 Good citizenship (output)
Interface displayed by software should not prevent reading of webpage.

R4 Predictability
User should be able to anticipate the outcome of their actions in advance.

R5 Intuitivity (output)
User should be able to understand the output of the system.

R6 Intuitivity (input)
User should be able to understand what to input into the system.

R7 No time constraints
There should exist no time-sensitive interactions; user might be disabled in a way that makes it difficult to input quickly.

R8 Independent of markup
It is wishful to expect all visited websites to have sane markup, so semantics should not be relied

upon. We point at the things that we can see, so a pointing system should be based around the visual layout of the page, rather than the semantics.

R9 Resilient to change
Many modern websites have changing interfaces, such as content being loaded in dynamically, or elements changing state (ie collapsible components).

Some of these requirements are inherently at odds with the choice to use a web extension hosted within the content of the page under navigation; necessarily, the interface of the software displayed in the webpage will need to overlap content on said webpage. Likewise there is no telling in advance which keybindings will cause conflicts with an unknown webpage. The hope is that a solution can be found that works with a usefully large portion of the Web.

These requirements have a low priority:

R10 Co-citizenship with tabbing
Existing tabbing navigation should still be available to user, as it is still useful for some tasks (e.g. form navigation).

That is to say, they are desirable but not essential.

4.1.2 Functional requirements

These requirements have a low priority:

R11 Efficient navigation (semantically unrelated elements)
Navigation to elements that exist in separate visual containers to the currently focused element, should be possible in less than 4 keypresses.

R12 Efficient navigation (semantically related, but distant elements)
Navigation to elements that exist in the same visual container as the currently focused element, should be possible in less than 4 keypresses.

R13 Efficient navigation (arbitrary position)
Navigation to elements that exist in the same visual container as the currently focused element, should be possible in less than 4 lookups.

4.1.3 Anti-requirements

The following are explicitly *not* aims of the system, so performance in these areas need not be measured.

R14 Efficient navigation (semantically related, nearby elements)
Navigation to elements that exist in the same visual container as the currently focused element, and are nearby: tabbing navigation already performs this very efficiently (albeit unpredictably). The more pressing concern is how to navigate between visual containers.

R15 Navigation to off-screen elements
In this pointing-based interface, it is assumed that the user will only ever be trying to navigate to elements that they can point at; thus off-screen elements need not be captured.

R16 Fast navigation
The goal is 'efficient' navigation, not 'fast' navigation. There can exist a correlation (less inputs will, from a GOMS perspectively, necessarily reduce task time also, provided cognitive and perceptual factors are negligible). But efficiency is the more important battle, as the system is targeted at low-input-bandwidth users, such as those with RSI (who need to avoid repetitive motor activity).

R17 Better performance than mouse pointing
This keyboard pointing method should be compared primarily to other keyboard pointing methods (ie tabbing navigation). It is for users who cannot use a mouse to point, i.e. due to physical reasons like disability, or interface constraints such as using a Smart TV Web browser requiring remote control input.

4.2 Design

4.2.1 Problem domain

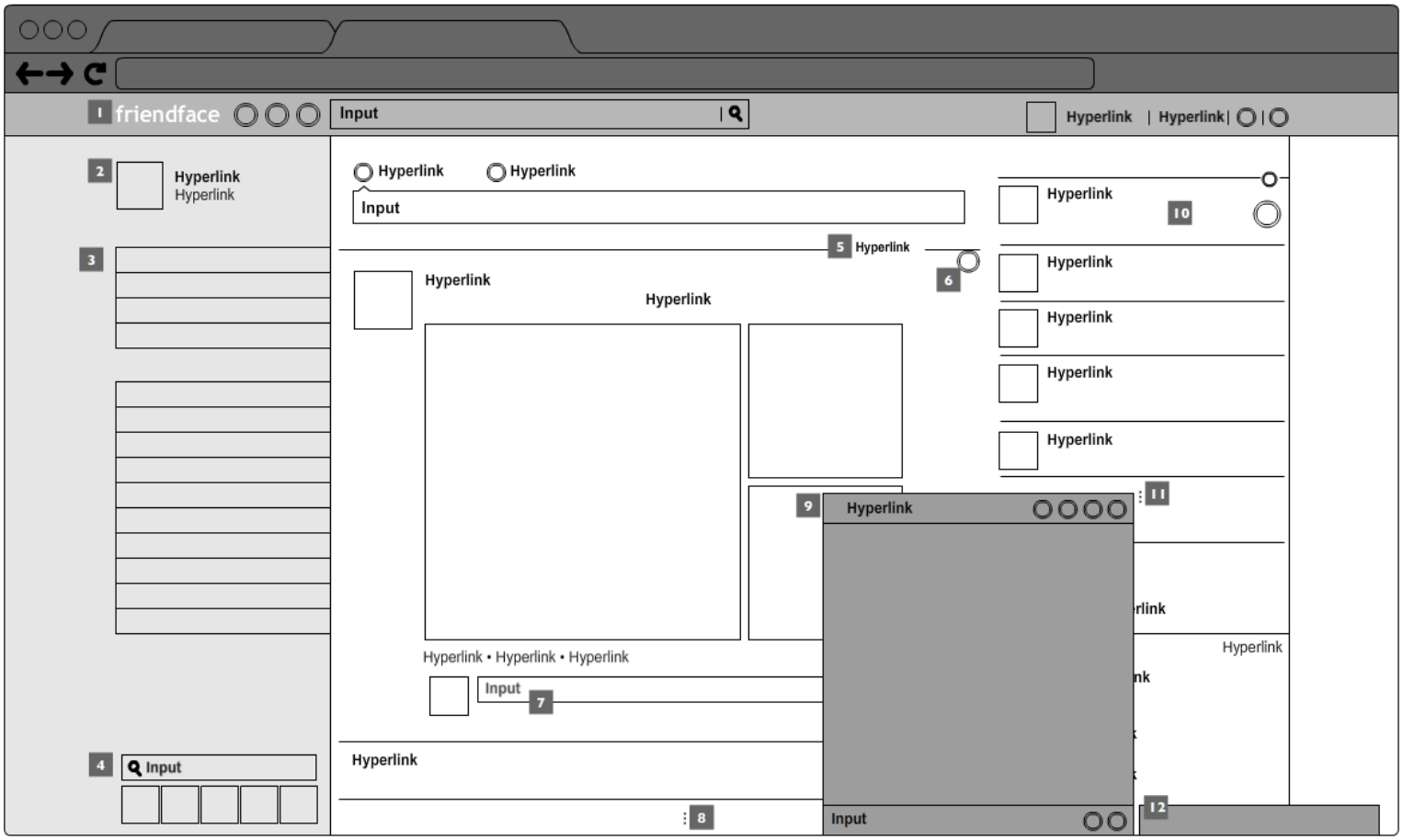

Figure 4.1: A typical complex web application layout.

Our software solution will need to cope with the many and varied pointing scenarios presented in typical web applications like Figure [4.1]. Some discrete cases of interest have been labelled; a discussion follows, regarding what is significant about these cases, and how we could consider optimizing our software for such cases.

1 — Navigation bar floats above content
This is a 'web application' version of interface 'chrome' (application controls around the content). Intuitive pointing needs to respect the current visual position of the content, and understand that the content can scroll under the floating chrome. Tab navigation ignores the presented position of the content, and can focus off-screen elements (despite the user's not being able to see such elements). It would be more efficient to restrict the search space to visible, on-screen elements (i.e. things that can be 'pointed' at).

2 — Elements can duplicate each other's actions
For example, a hyperlink could be the caption to some icon, and clicking either would produce the same result. In a poorly-marked-up page, a tab journey would traverse both of these elements, despite their function being equivalent. A more efficient mechanism would be one that allows the user to skip past duplicates (by combining them conceptually into one control, or otherwise by allowing the user to jump over multiple elements in a traversal).

3 — Repeating units of repeating units
These sets of buttons are grouped into categories. If the user can express that they are interested in a category, it could eliminate other categories entirely from the search space. Similarly, if repeating units could be detected, then an explicit list traversal mechanism could be offered to the user. Detection of such things is likely to be difficult though, as it requires sane markup.

4 — Unskippable lists
Similar to the previous point, a list of elements can be marked up without the use of an explicit list component (for example, a drop-down). In tab-navigation, skipping past the list is impossible; the user must tab past each individual element. A better traversal method would not rely on explicit list markup for entering and escaping lists.

5 — Rarely-used elements
Though it needs to be possible to point at all elements, it is not ideal for obscure elements to appear in the majority of tab journeys (the user only wants to traverse them in exceptional cases). Our system needs to give the user control over the journey they take to point at an element, such that they can decide whether to traverse toward elements they would otherwise avoid.

6 — Invisible elements
Some elements only present themselves upon mouseover or keyboard focus. Similarly, components like menus might expand as an option is touched on mouseover. Our pointing system might need to provide the user with an equivalent mechanism to 'mouseover' to probe for invisible elements. In particular, actions other than 'click' should be made available for interacting with the environment.

7 — Many forms
A page can contain many small forms that are unrelated to each other. Since tab navigation is relatively good at traversing forms, browsers like Safari allow the user to traverse just form components. But it is hard to be sure whether pressing tab will move to the next element in some form, or to a different form altogether. It needs to be clear to the user how far they will move when they interact, so a visual indicator of the constraints of the search would be useful.

8 — Infinite Scrolling
Many websites with dynamic content recruit 'infinite scrollers' to procedurally load in more content as the user scrolls the page. These are dangerous to tab navigation, which traverses depth-first; once it is in an infinite feed, it cannot escape, since as focus approaches the 'end' of the content, further content is loaded indefinitely. Thus elements after an infinite content feed are rendered unreachable. This is another case where our traversal system will absolutely need to provide a skipping mechanism, or choice of route.

9 — Layout Occlusion
Floating elements can occlude those elements behind them. Thus it becomes impossible to point at the occluded elements. Actually, tab navigation copes very well with this problem (as it traverses by markup, rather than visual position). Our system will not try to solve this problem though, since even mouse pointing suffers from this. Since websites are mouse-optimized, it is expected that they would avoid creating problematic occlusion scenarios, or at least make them reversible (for example, some occluding content can be collapsed).

10 — Repeated units of varying size
Identification of repeating units based on a visual analysis might be difficult, as the visual difference can be non-trivial; some elements can be larger than others. At any rate, our system will not attempt anything as complex as computer vision for understanding page layout. Ideally the system should be designed to not need an understanding of the page layout (like mouse navigation, which simply points at coordinates specified by the user).

11 — Multiple Infinite Scrollers
A page can recruit infinite scrollers for content other than just the main feed. Thus assumptions like 'any infinite scrolled content is likely to be the main focus of the page' can't be so easily made. The main takeaway here is that our system should not optimize interaction based on some imagined understanding of the purpose of the page structure; it could easily make incorrect assertions, thus would not be resilient to bad or obscure web design.

12 — Multi-modal Elements
Elements can change size or state, for example collapsible components. Any representation the system stores about the layout of the webpage should be sensitive to this possibility of change.

4.2.2 Design Criteria

We intend to cope with the aforementioned minefield of design traps, by prescribing the following design criteria:

1 — Navigation bar floats above content
Our system will only point within the current viewport of the browser. As for elements hidden in collapsed menus, we should not offer clicking for these until such time as that menu is opened — in other words, we must offer pointing only for currently visible elements.

2 — Elements can duplicate each other's actions
We will use a hierarchical traversal — that is, dividing the search space each time, in a logarithmic manner — rather than the linear mechanism used by tabbing navigation. This allows groups of content to be efficiently skipped. We will attempt to provide a variety of pointing choices at every stage of navigation, so that duplicates do not overwhelm the list of choices.

3 — Repeating units of repeating units
Rather than assuming we can make informed simplifications based on markup, we will simply side-step this using a hierarchical traversal; the user can efficiently skip past large portions of the page.

4 — Unskippable lists
Similar resolution to above.

5 — Rarely-used elements
Hierarchical traversal helps here also; access to a large search space can be made possible in very few lookups. If important elements are more prominent on the page (say, larger), then a spatial hierarchy creates a bias towards important elements first, but provides also the power to point at smaller, more obscure elements with a bit more work. This allows the system to be quick in the general case, and

almost as quick for the obscure case.

6 — Invisible elements
We will attempt to emulate the 'mouseover' event of a mouse cursor. For this, our system needs certainly to be able to be aimed at targets. We will present crosshairs to achieve this.

7 — Many forms
Our system needs to express the boundaries of its search — we will display borders describing the region in which search is directed, as well as where it will move on the next lookup. This allows the traversal to be predicted.

8 — Infinite Scrolling
Our system needs to provide absolute traversal, so that it doesn't get stuck in focus traps like infinite scrolling content.

9 — Layout Occlusion
Our system will call the same lookup as a mouse does when clicking, so that occluded elements are handled in the same manner. That is, we will lookup the frontmost element at the inspected coordinates.

10 — Repeated units of varying size
Our system will not attempt repetition analysis, but provide through its hierarchical pointing a robust way to aim at elements in irregular lists.

11 — Multiple Infinite Scrollers
Our system will avoid optimizing based on assumptions of page markup — it will be entirely visually aimed, like with the mouse, so it can cope in a familiar way to content that flows off-screen, or has no fixed size, or has a complex role — and it does not need to know that these factors defined those page elements.

12 — Multi-modal Elements
The system will not rely on a cached representation of the page. Should suggestions be made that have gone out of date, re-scanning should be possible.

4.2.3 Suggested Solution

These criteria suggest a system that:

1. Directs navigation spatially
2. Simulates at least the pointing behaviour of a mouse cursor (but not limited only to this)
3. Constrains search to a bounded, predictable space
4. 'Drills down' through search space hierarchically
5. Makes a variety of suggestions at each stage of traversal

4.2.4 Mechanism 1: Pointing at coordinates with 'crosshairs'

Most of these design criteria are fulfilled by a mechanism akin to that used by 'MouseGrid' (see Section 3.7). That is: the viewport of the browser is divided into a (3×3) grid, and each cell assigned a keyboard shortcut. The user specifies by keypress which cell contains the element they are interested in. Then a new, smaller grid is constructed within that cell, and the process repeats itself. Grids are constructed within grids, and a set of 'crosshairs' (like on the sight of a gun) can follow the center

of the most specific grid, and provide mouse behaviour (sending 'click' and 'hover' events) at those coordinates.

MouseGrid on its own is not, in itself, entirely novel. More unique about this application of it, is that it is hosted within a webpage, rather than being a first-class OS citizen. A web-based implementation can recruit information from the context of the page, for example responding to content updates, or changes in zoom or scroll. Styling could be programmed to respond to the content underneath, or even be customized by the user in CSS. Though we may not pursue all of these options, it is certainly a flexible foundation that can be developed to meet emergent user needs.

'Crosshairs' pointing can be made more predictable by 'painting' a highlight on the element it targets. In this way, the user knows, before they send a click, which element will be clicked.

This accounts for all design criteria except for the production of 'suggestions'.

4.2.5 Mechanism 2: Pre-empting user needs by suggesting a shortlist of 'clickables'

Mechanism

Our provision of 'suggestions' is the key to the novelty that our pointing system will deliver.

We introduce the term 'clickable' — a button, hyperlink or other element, which performs an action upon being 'clicked'. Any 'clickable' is potentially the target of a user's pointing. Having expressive access to the structure of the page is an advantage the web gives us: unlike with desktop applications, we do not require the application layout to expose accessibility information. We do not need developers to disclose which elements are clickable, or expose explicity access to them. However an obstacle exists: many things on the web can behave like buttons, so we need to provide our own robust mechanism for classifying which elements are clickable.

The smaller a 'clickable' is, the more precision is required to specify its location, and consequently the more lookups must be specified by the user. However, this search can be shortened if the user's intentions can be pre-empted. Allowing the user to choose from a set of suggestions, can spare them from having to specify the exact location of their target. Especially in the case where few 'clickables' exist within the search space, suggesting even a small subset of the clickables in that search space enables the user to potentially forego a longer location-based search.

Suggestions can complement the grid-drilling approach. They can be re-evaluated and constrained to the space of the latest grid, every time the user 'drills down', allowing increasingly more informed suggestions to be made. These suggestions will be keyboard-activated, and thus need to display labels of the shortcut that invokes them, as well as indicating which element, upon invocation, will be clicked.

Visiblity Concerns

In order to allow the clickable underneath the suggestion to be seen clearly, the label that designates the shortcut for activating the clickable should 'fly out' from that element. An arrow will denote the relationship between the label and the element it 'flys out' from. Such suggestions will be referred to henceforth as 'flyouts'. Their presence provides necessarily a visual indicator of what the user can click — one which is not guaranteed in tabbing navigation.

If many flyouts are densely-packed, there exists a risk that flyouts will cover each other up, or the elements that they are responsible for. Hence we will implement some avoidance strategy to make

them fan out from their respective targets in separate directions, allowing the clickables beneath to be seen. Additionally we shall endeavour to limit the number of flyouts.

There are further advantages to using a smaller number of flyouts: the input vocabulary of the system is kept small. We will use one flyout per grid segment — a total of nine. This also allows some symmetry with the vocabulary size for drilling; perhaps a modal input system could take advantage of this, and map at any time to either drilling shortcuts or flyout shortcuts.

Metaphor

Incidentally: one metaphor for flyouts, is a space-constrained, constant-time tab navigation. That is: if tabbing could be directed to only traverse elements within a specified screen portion, and also if instead of highlighting a single element for clicking, several were suggested (any able to be activated in a single keypress), then it would somewhat resemble this 'flyouts' system.

4.3 System Name

The chosen name for the pointing system was 'ChibiPoint' (stylized in Japanese as ちび点). 'Chibi' (ちび) was felt to describe the youth of our pointing system, as it is used to refer to small, cute things. The chosen English pronunciation is 'chibby', although this is an oversimplification. The use of the '点' kanji to refer to 'point' (as in point score, rather than spatial pointing) is a deliberate pun: an allusion to our hopes that the system will perform well.

Chapter 5

Implementation

5.1 Chapter Purpose

The primary objective of this chapter is to disclose and attribute the external software that we relied upon. It is not the purpose of this dissertation to explain web development in detail: we require the reader to have a working knowledge of web development and technologies.

Much of the engineering involved in ChibiPoint is not of academic interest. We focus here on non-trivial engineering problems, or ones that required novel solutions.

5.2 Primary Technology

We decided (as per Section 2.1.3 and Section 2.3.1) that ChibiPoint would be implemented as a browser extension, for the arbitrarily-chosen browser of Google Chrome. It was felt that a proof-of-concept with any browser would validate the approach for use in other browsers, especially if no browser-specific code was relied upon.

To keep the system browser-agnostic, we ensured that all code necessary for proving our concept, recruited only standard, widely-available web technologies. That is: HTML5, CSS and JavaScript. Respectively these provide any modern web browser with page markup, presentation, and interaction. This is more of a theoretical taxonomy than a practical one, as in reality the lines between their roles are heavily blurred.

Google Chrome's browser extension architecture offers various ways to extend the browser. The relevant method for our needs was a 'content script'[52]. Google defines these as: "JavaScript files that run in the context of web pages. By using the standard Document Object Model (DOM), they can read details of the web pages the browser visits, or make changes to them". Thus a content script has the power to present a user interface within an existing webpage, as well as capture and effect interactions. This gave us all the power we needed to:

1. Capture input from user
2. Draw the ChibiPoint interface
3. Effect interactions within the page, at the user's request

Additionally, since 'content script' development asks only that the entire system to be comprised of web technologies, the same system could just as easily be embedded into a webpage. This gave two benefits:

1. The system could be tested independently from the context of a web extension; packaging problems such as privileges and permissions need not obstruct development.
2. The system gains a delivery option: it can exist within a website, so that users who have not installed it, can still enjoy its functionality. Perhaps this provides a cheap solution for people who are prepared to develop their website for accessible pointing.

5.3 External Libraries

5.3.1 Libraries Required for General Operation

We utilised, as a foundation for our key-capture and display of interface, code from the open-source Google Chrome browser extension Type-Ahead-Find [13, 53] (which we describe in Section 3.7). The source code to Type-Ahead-Find is made available under the GNU GPL v3[54] license.

Side-note: The GNU GPL v3 license dictates that derivative works, such as ChibiPoint, must make their source code available under the same terms. Thus ChibiPoint, too, is unavoidably licensed under the GNU GPL v3.

Our JavaScript code was to be split into many modules. There existed a need to manage dependencies, and their loading order, because otherwise code could be invoked that had not yet loaded. We wanted to avoid solutions that relied upon the browser extension's packaging system, since such solutions would be browser-dependent. Ultimately we used RequireJS[55], an open-source Javascript module loader. RequireJS is available under the 'new BSD' or MIT license[56, 57].

To make RequireJS loading compatible with browser extensions, we used a RequireJS extension provided in an example by 'nonowarn'[56]. No license was attributed to this code.

Our system needed often to select page elements and modify them: for example applying highlights to code on the webpage that was being pointed at, or else just changing ChibiPoint's own interface. We used JQuery[59] for much of our querying and selecting of page elements. JQuery is distributed under the MIT license[57].

Often there was a need to search for page elements within some specified coordinates. To achieve this, we adapted code from the jquery++[60] helper library for JQuery. It, too, is distributed under the MIT license[57].

For detection of clickables, we used code from StackOverflow.com discussions[61]. All StackOverflow discussions are distributed under a CC BY-SA 3.0 license[62].

5.3.2 Libraries Used for Performance Evaluation

For the quantitative study of our system that would follow, we needed to be able to measure keypresses, and save this data out for analysis. Tracking keypresses was trivial, as we already listened for input from the user, to control the system. Saving out this data was the technical challenge that remained.

We decided, upon effecting a 'click' in ChibiPoint, to dump all recorded data into a string, convert that string into a 'Blob' data type, and force a download of that 'Blob'. For saving the Blob, we used Eli Grey's implementation of the W3C File API's saveAs() function[63]. Our legacy code includes also Eli Grey's Blob implementation[64], but this particular code is no longer needed by the system. Both are available under the X11/MIT license[65].

5.4 Novel Problems

5.4.1 Detecting 'Clickables'

A crucial requirement of our system was the ability to detect 'clickables' (as described in Section 4.2.5) — we needed to do this in order to suggest shortcuts to users.

Taxonomy of Clickables

Detection of clickables is problematic, because there are many ways in HTML5 webpages to make page elements respond to clicks. Koch provides a taxonomy[66]:

- DOM Level 0 - inline model[67]
- DOM Level 0 - traditional model[68]
- DOM Level 2 - event listeners, W3C model[69, 70]
- DOM Level 2 - event listeners, Microsoft model

There exists also a DOM Level 3 specification for events[71], which is, at the time of writing, a work in progress.

Detecting clickables is further complicated by the concept of event delegation[72]: it is incorrect to conclude that an element with no events must be 'not clickable', because an element's ancesters can listen for these events. For example, a list that contains graphics can be 'delegated' responsibility for listening to click events on all of those graphics. In this situation, a whole set of distinct 'clickables' can exist, described by just one event listener.

We chose not to look into the 'delegates' problem — we can demonstrate that our pointing system works on a conceptual level, so long as at least some types of clickables are detected. We felt we could capture a large portion of clickables by targeting at least those bound as 'DOM Level 2 - event listeners, W3C model' events — this is the event binding strategy used by JQuery, a library adopted by upwards of 58.2% of websites[73]. Other popular alternatives, such as PrototypeJS[74], use this strategy also.

We draw attention also to the fact that clickable suggestions are not the only pointing mechanism in ChibiPoint — in cases where detection fails, the MouseGrid-style crosshairs mechanism we describe in Section 4.2.4, can be used as a fallback to send click events to any element that the user can point at.

Detection

It was challenging to detect which elements were listening for click events. There exists no standardised way to retrieve the list of event listeners bound to an element. Rather, an attempt was made to introduce such a standard in the W3C DOM Level 3 Events specification[75], but this addition was later redacted[71]. User agents (that is, web browsers) can expose access to their own internal representations of event listener lists. In fact, our browser extension could perhaps use Chrome's Command Line API[76] to look up such event listener lists — but this requires its Developer Tools Console to be open, and removes our browser-independence. A such, a different solution was needed.

A discussion on StackOverflow.com provided the answer we needed[61]: before the DOM is constructed, overwrite the definition of 'addEventListener' used in the prototype for HTML elements. In

this way, we can ensure that, any time an event listener is added to an element, it is tracked in a list on that element (which we can later query for). After this it was trivial to, using JQuery, select all elements in the page which have this list attributed to them.

For additional coverage, we infer 'clickability' of elements simply by reading their tag name. For example, all 'anchor' tags are assumed (admittedly generously) to be purposed as hyperlinks. Admittedly it is possible to confirm the role of the anchor tag as clickable (for example by checking whether it has a URL associated with it), but it is not too harmful to present false positives — users are given multiple other suggestions, so assuming that user realizes they do not need to click the element, it does not cause them great problems. Form elements are assumed also to be of interest always to the user, so we register these as clickables.

A visibility check is performed, since this is a visual pointing system and users have no need to click on elements they are not aware of. For this we simply check whether the value of an element's 'hidden' and 'visible' attributes.

Candidacy for Recommendation to User

On all clickables that are detected, we perform coordinate tests to detect which grid quadrant they fall into, if any. Coordinate comparisons are performed relative to the browser's viewport of the page, so scrolling is respected. We were generous with coordinate comparison — a partial overlap was sufficient to qualify as 'within bounds'. Incidentally, coordinate lookup was performance-intensive, so we made sure to filter the list of page elements down to just the clickables before we persued coordinate comparisons. Additionally, any coordinate lookups that would be used in further comparisons were cached, to avoid repeated lookup.

We tried to observe these priorities for building a useful shortlist of clickables were as follows:

1. To be eligible, a clickable needs to be within coordinate bounds of most specific grid
2. Clickables from differing directions are preferred, to improve the variety of content detected
3. If there are no clickables available in varied directions, then it is permissible for clickables from a similar direction to be offered

In response to those priorities, this is the algorithm we used to generate a shortlist of clickable suggestions for the user:

1. Create list of all clickables on the page (using JQuery selector)
2. Narrow down that list to those clickables within our most specific grid
3. Categorise each clickable — which grid segment(s) do they overlap?
4. Each grid segment attempts to elect as their suggestion, one clickable found within them (and this is removed as a candidate for proceeding suggestions)
    - Any grid segment that finds no clickables within them, is permitted to elect instead clickables found in other grid segments — we end up with multiple suggestions from the same area, but this is preferable to making no suggestion at all. This clickable is removed as a candidate for proceeding suggestions.
5. Present 'flyouts' for each clickable suggestion elected on the shortlist

5.4.2 Citizenship

Keybindings

We had to be careful to avoid blocking the user's interaction with websites. In this vein, the ChibiPoint interface is not even rendered until the time it is invoked. When capturing keyboard input from the user, we ensured first that keyboard focus was not in use to interact with forms.

The hotkey to invoke ChibiPoint was given first-level citizenship in the web browser, so that it did not clash with other keybindings. This was a feature afforded to us by the Chrome Extension packaging, so it violates our browser-independence rule. As such, we assigned also for invocation, an obscure key (keycode 167 — on Mac, this is the 'section sign' character, §). We felt that use of this rare key was a fair compromise on citizenship (it is unlikely to be listened for in standard web browsing), whilst still enabling use of ChibiPoint's featureset on all platforms.

One advantage of giving ChibiPoint invocation a keybinding with first-level browser citizenship, is that such a keybinding is not used in form interaction — thus, the system could be invoked with this shortcut, even from within a form context. Other keys used by ChibiPoint do not enjoy the same privilege, and could be intercepted in contexts like forms. However that is not a problem following invocation, since by then we have seized focus and escaped any form context we might have been in.

Whenever a keypress event is captured by ChibiPoint, we ascertain whether the key is mapped to any functionality within ChibiPoint. If so, that functionality is invoked, and we end propagation of the event, so that the website does not receive the same keypress. However, if said keypress does not match any of our mappings (for example: the tab key), then we take no action, and allow the event to propagate to other event listeners. In this way, we allow co-citizenship for all keypresses that are not relevant to our system, and crucially we maintain compatibility with tabbing.

Performance

Performance was important, as ChibiPoint resides in all browsing contexts that are opened. Querying the page for clickables and their coordinates was very intensive, so we had to ensure that this was done sparingly, and onl on user request. As such, production of suggestions was only effected in response to user input — invocation of ChibiPoint, or drilling into a grid segment were the ways to effect re-appraisal of suggestions.

Respecting Viewport Changes

We had to respect that the page could resize or scroll at any time. To cope with this, the entire interface is specified in relative dimensions; changes to the viewport size stretched ChibiPoint appropriately, and scrolling did not move the ChibiPoint interface out of view, as its coordinates are 'fixed' relative to the viewport itself.

Upon scroll or size change, the 'flyouts' suggestions have to move. They stay 'clipped' to the clickable they describe (and scroll with it). Once the clickable itself is off-screen, we remove the flyout — the user does not need keybindings applied to elements they can't see, as this would offer unpredictable behaviour. The 'crosshairs' pointing mechanism behaves slightly differently — instead of following the target at which it points, it stays fixed in space, and allows new elements to scroll into its reticule. This allows it to aim like a mouse. For visual indication, it continues to 'paint' its latest target (and 'unpaint' targets it loses).

Markup

Our system shares a DOM with the webpage in which it is hosted. Thus there is the possibility the markup we introduce could get caught in the crossfire of existing scripts or styles on the webpage — selectors that apply a universal text style could also modify ChibiPoint, for example.

We cannot make our interface resilient to all types of interface change — the hosting webpage has just as much power to change our interface as we do, as it uses the same technologies in the same context. The only difference is that they are not trying to change our interface, and do not know it exists! Conflicts can occur by accident, though, if generous selectors are used to edit webpages.

To protect our markup, we prefaced all our CSS class names with 'chibiPoint_', to reduce the likelihood of accidentally being selected by queries or styles on a webpage. This turned out to be a necessary decision — early versions of ChibiPoint were being accidentally re-styled by some websites, such as Kotaku.com.

Some websites, such as Facebook, had interface that occluded ChibiPoint's. We fixed this by explicitly reserving the highest z-index in the draw order.

Use of the browser's Developer Mode became troublesome, since the ChibiPoint interface occludes all elements behind it; it became difficult to inspect elements, as they could not be clicked on when ChibiPoint was loaded. We provided a partial fix to this — the ChibiPoint interface is now rendered only when it is actually in use. Thus elements behind it can be inspected, provided it is closed.

Aside from Developer Mode selections, we needed to make sure also that regular clicking was still possible whilst ChibiPoint was loaded. Thus we use a '

```
pointer-events: none;
```

' style to ensure that ChibiPoint does not intercept clicks, even when it is present.

Regarding the markup of our grids within grids, we considered using a Composite pattern[77], so that all grid levels could be treated the same way in code. However we realised that is was more useful to be able to give special treatment to certain grid levels — in particular, the existence of a top-level grid is an important factor in the state of the system: it determines whether ChibiPoint is in use. We can also bound any page changes we make to be constrained to within this container, keeping good citizenship.

Thus we specify a particular container, 'chibiPoint_gridContainer', in which to build grids. This container was always present, whether there were grids within it or not, so it provided a consistent place to check the state of the system. We were able also to apply transformations to that high-level container when we wanted to modify the entire grid, such as removing the interface from render.

Necessary conflicts existed also — in cases where we 'painted' page elements (for example to show which clickables are designated by each flyout, and also which element is targeted by crosshairs), we had to be careful not to paint them in a manner that would break other code that interacted with it. We add a 'chibiPoint_painted' class to such elements, or otherwise look for and remove that class to 'unpaint' them afterward. Admittedly this produces conflicts for code which is not expecting multiple classes to exist on some element. This is a regrettable problem of our browser extension's sharing a DOM with the page it resides in. ChibiPoint does not modify elements unless it is in use, so even if problems are caused by this mechanism, the user can still fall back on their standard web browsing experience if they close ChibiPoint. A more insulated painting mechanism could be achieved by 'painting' an element owned by ChibiPoint (as opposed to painting the element directly), in a similar manner to how flyouts provide their own arrows to point to targets.

5.4.3 Interface Malleability

Philosophy

Since we were creating a novel navigation system, it was useful to have a lot of expression over how the interface works. In fact we wanted to extend this flexibility to the user — users can provide custom stylesheets[78] to change the way web content is presented to them. Additionally, different form factor devices such as mobile phones with small screens could benefit from an alternative interface style (for example, text size differences.

We place a lot of power in the styling of the interface, by using the presentation layer to inform the state of the system. That is: the pointing grid's size is specified in terms relative to the viewport of the window, and this affects where the system will point (as crosshairs are aligned to the center of the most specific grid). If a user had different requirements, they could specify the dimension rules of the grid differently.

Example Usage: 'Grid Growing'

To give a specific example of the interaction power that emerges through styling, we present 'grid growing': a rule used to specify sizes of nested grids. Here is the problem that 'grid growing' solves:

Consider the case where a user wishes to click a button that exists along an edge of the pointing grid. Crosshairs only target at the center of the most specific grid. Thus edge-aligned elements are difficult to point at, because grids nested within a grid will also be built flush with the edges of their parent grid — drilling down with the objective of converging towards some grid edge takes many lookups, with necessarily less and less distance travelled each time the user drills.

With 'grid growing', nested grids are not strictly bounded inside their parent; they float over it, and are allowed to be slightly larger than the segment from which they were constructed. Crucially, the edges of nested grids are not flush with their parent grid segment; their boundaries are expanded by a small percentage compared to that parent (though still guaranteeing that the grids are always more specific than their parent). Thus, convergence of the crosshair position towards a grid edge is enabled in possibly fewer lookups, as each step allows the user to travel more distance.

Since clickable dimensions are not highly-specific (ie, buttons need to be designed large to be easy to click), it is better to use 'grid growing', as this biases the system towards travelling less specific (farther) distances, rather than more specific distances. This allows less lookups for the expected general case: clicking on relatively large buttons — in exchange for slightly more lookups being required for tiny buttons. We perform no analysis of which is the more effective default, but the fact that 'grid growing' is achieved and parameterized entirely in CSS stylesheets, allows the user to tweak this behaviour if they have different needs.

Architecture

Incidentally, we found that we had a need for macro language and inheritance in our CSS stylesheets — often colours needed to be repeated, or otherwise we had portions of our styles that needed often to be re-used in other styles. CSS does not classically support concepts like variables, or true inheritance. Admittedly something like inheritance can be achieved on the markup level — for example by defining some object in terms of multiple styles — but no semantics exist to enforce this relationship, so maintenance becomes a problem.

We chose an architecture that allowed us to control the relations between styles, and allow our system

to be defined by re-usable parameters — such as line thicknesses or color schemes. Even the 'grid growing' percentage is able to be defined like this, so it can be trivially turned off. To achieve this, we wrote styles in the LESS (Leaner CSS)[79] pre-processor, which compiles down to CSS. This gave us the feature-set that we needed from a styling technology, without breaking our commitment to use widely-available web technologies in our implementation.

5.5 Summary

5.5.1 Conformance to requirements

Here we summarise how well we delivered on our requirements:

Delivery on Non-functional requirements

We responded in the following ways to these high-priority requirements:

R1 High availability
Our software is free in price, and theoretically independent of browser, as well as using as input a key-based vocubulary that many devices can map to.

R2 Good citizenship (input)
Invocation is possible with a shortcut blessed by the browser (though this is browser-specific). ChibiPoint gives priority to form input, and does not intercept keypresses unless it is in use and sees a key it can act upon.

R3 Good citizenship (output)
Interface can be closed, and is semi-opaque to allow content to be read. Flyouts attempt to fan out to reveal the clickables they target.

R4 Predictability
User can choose direction to navigate, with increasing specificity. All targets the user can click on are painted beforehand, disclosing in advance what will be clicked.

R5 Intuitivity (output)
Indicators are provided for which element will be clicked, and a flashing animation is used to show that a click has been sent (regardless of whether it achieves the expected result).

R6 Intuitivity (input)
Primary keyboard mappings are presented in the interface, adorning the controls they invoke. Invocation shortcut can be specified by the user.

R7 No time constraints
There are no time-sensitive aspects to the interface interaction.

R8 Independent of markup
The system detects clickables and navigates through the page spatially, so is largely independent of the markup used (although there exist types of clickable we cannot detect).

R9 Resilient to change
The system survives scrolling and resizing. Elements that exist within collapsed forms are not suggested as clickables until they are visible.

Overall we meet well all our high-priority non-functional requirements.

We responded in the following ways to these low-priority requirements:

R10 Co-citizenship with tabbing
Tabbing presses are not intercepted, so tabbing can be used alongside ChibiPoint. Moreover, tab focus does not preclude the usage of ChibiPoint, as its browser-blessed shortcut can be observed even when typing in a form context.

Even our low-priority non-functional requirements were met.

5.5.2 Delivery on Functional requirements

We responded in the following ways to these high-priority requirements:

R11 Efficient navigation (semantically unrelated elements)
The system uses a hierarchical navigation system, so efficient navigation is expected. Certainly we are not vulnerable to markup distance between each focusable element, as we traverse spatially. A more rigorous evaluation of efficiency will be pursued in the quantitive evaluation that follows.

R12 Efficient navigation (semantically related, but distant elements)
This will be ascertained in the quantitative evaluation. However the hierarchical navigation should mean that travelling distance is efficient.

R13 Efficient navigation (arbitrary position)
This, too, will be ascertained in the quantitative evaluation. Certainly hierarchical traversal is suited for efficient access to arbitrary locations though.

These will be evaluated more fully in Chapter [6], but theoretically we can expect an efficient solution.

5.5.3 Conformance to design

We succeeded in meeting our design criteria — we produced a system that:

1. Directs navigation spatially
2. Simulates at least the pointing behaviour of a mouse cursor (but not limited only to this)
3. Constrains search to a bounded, predictable space
4. 'Drills down' through search space hierarchically
5. Makes a variety of suggestions at each stage of traversal

That is: a MouseGrid-like system provided predictable spatial navigation, and provided pointing via crosshairs. We produced a shortlist of clickables for every stage of traversal.

Chapter 6

Evaluation

6.1 Overview

The primary system comparison we sought to pursue was 'ChibiPoint' versus 'tabbing navigation'. We aimed also to evaluate the contribution made by the 'flyouts' feature of ChibiPoint: whether it helps or hinders. As such, we evaluated three systems:

1. Tabbing navigation
2. ChibiPoint (with just the crosshairs feature enabled)
3. ChibiPoint (with both the crosshairs feature AND the flyouts feature enabled)

We aimed to quantitatively evaluate the following things:

1. Which system is the most efficient for each class of navigation tested
2. Which system is the fastest for each class of navigation tested
3. Which system is preferred by the participants
4. Which system is considered by the participants to be easiest to use
5. How reasonable is the amount of keypressing demanded by each system, according to the participants

Qualitative data was useful also, as it offered feedback on specific advantages or disadvantages of each system, as well as providing insight to usability and the reasons for user preferences.

We performed first a usability study, as a precursor to the more detailed quantitative testing. The usability study was expected to reveal how a new user approaches the system — what level of instruction is necessary, and what problems are encountered with usage. Addressing issues here would reduce problems in the larger, quantitative study that followed.

Regarding the quantitative study: a pilot study was conducted first, to ensure that the study approach was effective, before widening the participation. Omissions from that pilot study were addressed. A larger study then recruited 12 users to evaluate the systems in a counterbalanced 3×8 ANOVA (three systems, 8 navigation types).

6.2 Study 1: Usability Study

We recruited a lecturer of Computer Science for this study. The participant was chosen due to her being a known power user of keyboard navigation. Additionally she had large experience with using

accessibility hardware and software. We hoped that this participant could bring out the full potential of our system, and reveal what an expert of keyboard navigation expected from a system, as well as what strategies they tried when learning said system.

A full transcript of the session can be found in Listing [B.1].

The participant controlled the system using a complex accessible keyboard, with which she was already well-trained. This was an opportunity to confirm the concept that ChibiPoint's key input does not need to come from a conventional keyboard input device. No problems piloting ChibiPoint arose from this input choice, although the mapping of 'numpad layout' to drilling direction was not possible, since there was no numpad available (hotkeys had to be mapped elsewhere). It was found that, with the chosen drilling mapping (left hand of Qwerty, QWE|ASD|ZXC), the participant did not notice the spatial connection until it was pointed out explicitly. Perhaps if the user configured the mapping themselves, they would learn the relation (although this is out of scope for the proof-of-concept). For the evaluation study, instructions will be given to the participant, pointing out the mapping.

6.2.1 Intuitivity

Generally the system was found to be quite intuitive. Exploratory use was encouraged, with the researcher disclosing only the ChibiPoint initial invocation shortcut (Cmd K).

The participant understood that the labels upon each quadrant were keyboard shortcuts. She understood immediately that these shortcuts directed ChibiPoint's search into that region. Additionally, unlabelled shortcuts were able to be guessed; she worked out within two attempts that 'Enter' was the shortcut to click the element under the crosshairs.

The participant soon noticed the flyouts, and understood them to be keyboard-activated suggestions. Even on her first use of these, her efficiency was just one keystroke short of expert performance.

The participant understood intuitively that page scrolling keys co-operate with the ChibiPoint interface; she was able to click off-screen elements by scrolling to them and then invoking ChibiPoint, without instruction.

6.2.2 Interface problems

The BBC website had complicated nesting of clickables within its navigation bar. It became difficult to distinguish which flyout referred to which button. Judgement of which element the flyout was bound to, appeared to be based on the position of the label. This was an incorrect mental model; the label's position is only a heuristic for which interface element is bound. In reality, the arrow coming off the label is the indicator of which element will be clicked. Observation of the video suggests that the arrows were barely visible in this situation; their dark arrows were flush with the (also dark) frame of the webpage. Ultimately the participant was able to recover from the situation by relying on ChibiPoint's other pointing mechanism, crosshairs. This showed that the participant learned that both could be used situationally.

More contrast would help highlight the existence of the flyout arrows. Another solution is to uniquely color-match the flyout with the element it was bound to, so that the pairing is visualized. This led us in the next version of ChibiPoint to 'paint' flyout-bound elements, in the same way ChibiPoint's crosshairs paint their target. Flyouts were also given discretely different colors to each other, for further identity.

The participant was seen to crane toward the screen to read flyouts. We decided to respond to this by

increasing the size and legibility.

The participant noted that the crosshairs did not wholly emulate a mouse, since they did not trigger mouseover or hover events, nor display a 'mouseover' cursor. A cursory effort was made to address this, but no fix could be found at the time. Further work could pursue a solution for this, to improve the feedback of the system and complete the cursor metaphor.

The participant assumed incorrectly that drilling could not continue beyond the point where the quadrant labels were hidden. The grid gets smaller each time, and past a threshold the text and gridlines are removed so as not to occlude the content in the quadrants. However, drilling remains possible provided the user recalls the shortcuts. To address any misconceptions, we decided to explictly cover this behaviour in future training.

6.2.3 Implementation problems

ChibiPoint could not click buttons within Flash Player interfaces (such as 'play' buttons on web-based video). For a proof-of-concept this is unimportant. More work could investigate communicating with Flash elements.

Additionally key listener conflicts were seen when ChibiPoint was used on the Google Search page; input to ChibiPoint was intercepted by the webpage, rendering the system unusable. Again this is not important at proof-of-concept stage. Possibly the reason lies in the order in which key listeners are bound; ChibiPoint waits for the page to be loaded in its entirety before setting up its event listeners. Future work could investigate whether establishing event listeners earlier makes a difference.

A problem was encountered with detecting clickables on the *The Bath Chronicle* (`http://bath.thisisads.co.uk/register.php`) registration page. Flyouts were not able to be provided for several buttons on this page. A later investigation revealed that the clickables were created using inline DOM level 0 events, which ChibiPoint does not look for. Future work could extend clickable detection to include these.

6.2.4 Feedback

Overall, the participant was enthusiastic about ChibiPoint. She felt that she was able to learn the system "pretty fast", especially compared to "quite a lot of usability stuff". She liked the flyouts feature, expressing that "they were pretty good guesses". She understood how to point with it, and felt that the grid drilling was a "great" way to point. She said that she would prefer this system to tabbing "of course", and that it was "obviously faster because it's hierarchical. My first-years should be able to answer that question". She described the system as "pretty predictable [excepting bugs]", and compared to tabbing "absolutely [more predictable]".

As for extending the system, the participant expressed that it would be useful to be able to hide the interface when it gets cluttered. However she also conceded that the existing feature for closing the interface sufficed. She advised that a cheatsheet or documentation would be useful for discovering hidden features. We resolved to print all controls on the evaluation study instructions.

The participant also assented with the notion that she would desire sometimes a method to undo 'drilling down' ("there was, like, one time"). However this feature already existed, so we resolved simply to improve its visibility with future instructions.

6.3 Study 2: Quantitative Comparison of Pointing Systems

6.3.1 Outline

A pilot study set the sequence for the full study to follow. The main output of the study was the evaluation of pointing systems; participants were instructed to complete several tasks of the following ilk:

1. Go to a specified website.
2. Using the specified pointing system, click a specified button/page element.

After completing this set of tasks with one pointing system, the participant was asked to repeat it with another of the pointing systems under test, until all three systems had been tested. We counterbalanced the order in which pointing systems were allocated to participants, to reduce biases in learning the tasks or systems.

In addition to this evaluation of system performance, participants were asked to fill out two questionnaires. The first questionnaire was posed before the pointing tasks: participants declared their proficiencies with the computing concepts involved, as well as describing their demographic. The second questionnaire was posed after the pointing tasks, and asked the participants to give subjective feedback on the systems they used.

6.3.2 Demographic

All participants were known personally by the researcher. Participants were asked to volunteer, and offered a chocolate biscuit as recompense. A majority (9) explicitly identified as being current Computer Science students. One further was known to be a graduate of Computer Science, another pursuing an 'EngD in Digital Media' and the final participant a Physics student.

The mean average age for participants was 22.58, with a standard deviation of 2.151. The range was 21–28. These data are tabulated in Table [C.2]. A quarter of participants were female, the rest were male (Table [C.3]).

Proficiencies

Most participants were frequent users of computing devices; 5 using computers or smartphones for 13+ hours per day, 5 using the same for 7-9 hours, and just 2 using devices for 4-6 hours. Other categories (0-1, 2-3, 10-12) were offered, but no participants identified with these. These frequencies can be seen in Table [C.1].

Many participants (5) preferred to navigate using a mouse, with a majority (6) recruiting hotkeys in addition to the mouse. Only one participant actually preferred keyboard controls, and no participant identified with completely requiring full mouse support or full keyboard support (Appendix [C]).

Touch-typing proficiency was rated on a five-point scale ranging from the sentiment 'I look at every key before pressing' to 'I always type without even a reference glance at the keyboard' (Appendix [C]. Participants on (mean) average considered themselves a 3.42. A standard deviation of 1.084 was observed. Two primary users of the Dvorak keyboard layout were asked to use the (familiar, but not preferred) Qwerty layout during this study, and rated their touch-typing with respect to the constraint that they would be typing in Qwerty during this time.

A majority (11) used mainly Windows as their desktop Operating System. Browser choice was more divided, with 7 participants using mainly Google Chrome, 3 Firefox and 1 'Other' (at the time disclosed to be Safari, which was not an option offered on the questionnaire).

Disability

Regarding participant's exposure to accessibility needs or solutions, three had used Dragon NaturallySpeaking speech-to-text. The two who elaborated on the extent of their use of speech-to-text described only limited use. No participant had — at the time, or ever — had computer-relevant vision difficulties. Two had had in the past minor experiences with RSI, with one describing "RSI from mouse use, but nothing too severe to require a large change in typing/clicking", the other elaborating "Have changed input device due to pain in very specific circumstances.". No participants had RSI at the time of the study.

6.3.3 Methodology

The following describes the methodology for the full study. Deviations exclusive to the pilot study are disclosed in its separate section.

The researcher followed a script to ensure that all instances of the study were performed the same way.

Participants were briefed on the purpose of the dissertation and of the study, as well as what participation in the study would entail. They were assured that no identifying information would be kept about them, that it was the system performance being measured rather than their personal performance, and that they were free to withdraw participation at any point. They were told that there was no desire for them to generate any particular outcome, and that we simply wanted to compare the performance of the systems. With these explained, participants were asked to sign a permission slip, and were offered also a copy in case they wished to contact the researchers after the fact.

Participants were assigned a unique number. This enabled both the questionnaires they would fill in to be related as being from the same participant. Additionally participants were assigned a sequence in which to evaluate the three systems. Biases pertaining to the order in which systems are used, were counterbalanced by testing using all sequence permutations of the three systems equally. There were 6 permutations, so the study recruited a multiple of this: 12 participants.

The study began with the first questionnaire's being allocated. This asked participants to disclose their demographic (occupation, gender and age).

The participants began the testing activity by reading instructions on the system they had been allocated. The researcher then guided them through a training scenario to confirm that they had understood the instructions. During this training period, the researcher would answer any questions and correct any mistakes that occurred. In the case of learning the system 'ChibiPoint with crosshairs AND flyouts' before learning the system 'ChibiPoint with crosshairs ONLY', participants were trained in the use of both features in turn.

After training, all test websites were opened, and the participant was directed to the attend to the first browsing context, which was navigated to the website for task 1. The researcher would point out to them the page element that needed to be clicked in this task. The pointing objectives were also printed on a piece of paper that they could refer to.

The participant completes a task by clicking the specified element using the pointing system they have been allocated. At this point, telemetry (of keypress count and timing) is automatically downloaded

by the browser extension (that is, ChibiPoint — which is repurposed as a keylogger — whether its pointing features are in use or not). The researcher then confirms whether the correct button was clicked.

If the correct button was clicked (or some equivalent, such as a caption with the same hyperlink), the participant moved to the next task. If some other button was clicked by mistake, then the telemetry was discarded by the researcher, and the participant was directed to attempt that task again.

Once the participant completed all navigation tasks, they were allocated the next system to use for pointing, and given instructions and training in it (in the same way as before). They would repeat the array of tasks from before with this new system. This process was completed until all three systems were tested.

With all systems tested, the participants were finally asked to fill in a post-experiment questionnaire, describing their preferences and thoughts on the systems tested.

Upon completion, the participants were thanked, and were offered a chocolate biscuit.

6.3.4 Limitations

Only successful task completion is logged; mistakes are discarded. Thus any measurements will not necessarily be a model of 'first-time' usage; we capture instead 'beginner' usage of ChibiPoint.

In the same way, users of tabbing could perhaps improve their performance if they retried after an attempt where mistakes are made. So again this does not model 'first-time' usage of tabbing. Here we cannot assert that users are 'beginners' of tabbing, either: general keyboard navigation proficiency was assessed in the questionnaire, but from this no confident assertion can be made about the tabbing navigation experience. However, tabbing navigation is very deterministic in journey, so provided no overshooting occurs, the number of keypresses from a beginner is no different to an expert's.

We perform no analysis to assess the tabbing skill level of our participants (that is, how close to expert performance they achieve), but necessarily the tabbing performance measured is 'at least that of a beginner', and ChibiPoint performance is 'at most that of a beginner', so our comparison should be interpreted in the context of those boundaries.

Skiplinks are one exception to the non-determinism of tabbing journeys. We chose to disallow the use of these, as they are website-dependent, and would therefore not represent the performance of a general browsing mechanism. Thus variance for tabbing is arguably not as large as it could be. However the tests chosen were not ones whose performance could have been improved upon with the available skip links (with the possible exception of Test 8 on Wikipedia, but this Skiplink is broken on Google Chrome, the browser in use, so provides no benefit).

6.3.5 Metrics

The metric that was most important to study was efficiency — how many keypresses (and by extension, how much exertion) are required for a given navigation task. Secondary to this metric was time taken by each system to perform a navigation task — it is preferred, but not essential, for the system to be fast; the main priority is reducing exertion, which is connected to efficiency.

During the navigation tasks, user keypresses were recorded, as well as the times that they occurred. We recorded only keypresses relevant to pointing with the current system. For example, attempts to use the 'tab' key during use of ChibiPoint were intercepted, and answered with an alert to the user that tabbing was disallowed. We disregarded keyboard input that did not invoke functionality within tabbing or the

currently activated version of ChibiPoint. Page scrolling via the keyboard was considered unrelated to pointing, and so was not recorded. Attempts to invoke flyouts were only recorded when said flyout was present; if no flyouts were created (for example because no clickables are in the specified area, or if flyouts are disabled altogether), then the keypress was disregarded.

Time was measured from the first to last keypress, so no time was measured during page load or task explanation.

The post-task questionnaire asked participants to give subjective feedback on the systems they used. Some of the feedback was quantitative — which was the preferred pointing system, how easy was each system to use, and how reasonable was the amount of key pressing required — and some of the feedback was qualitative, asking for general thoughts on the task and systems.

6.3.6 Pilot Study

The pilot study existed as a 'dry run' to catch problems in the method of the larger study, before scaling up.

We recruited a Software Research Engineer with experience developing and assessing accessibility software. His preliminary questionnaire response (Figure [A.1]) describe a proficient computer user (with 13+ hours/day of device usage, 5/5 self-rating for touch typing, and a user of hotkeys).

A list of pointing tasks was created, but these proved to be uninformative — many of them duplicated classes of navigation already tested, and overall they did not test a wide enough spread of navigation types — to the extent that tabbing's strengths in form traversal were not represented. Additionally it was not clear what the delineation was between task setup and the actual recorded portion. The larger study uses a clearer separation of actions and setup.

Though these inspired the eventual classifications we would use for tasks in the full study, they do not in themselves paint a full picture of system performance.

These tasks were as follows:

1. Search 'piano' on YouTube
    (a) Then, select first result
    (b) Then, select 'Favorite videos'
2. Search 'sport' on BBC
    (a) Then, select first result
3. Search 'pillow' on Amazon
    (a) Then, select first result
4. View first article on Engadget.com
5. Google 'slam'
    (a) Then, select first result
6. Select 'Archives' on Megatokyo.com
7. Search 'computer' on Wikipedia
    (a) Then, select 'Section 2.3: The modern computer'

Some tests (for example the Google search) were found to be invalid, as ChibiPoint could not activate on this website due to key listener conflicts. The later study withdrew such tests.

Bugs in the instrumentation were found — certain actions, such as invoking flyouts — were not being recorded, so some keypress counts were artificially low. Luckily these were predictable and could be detected before reporting results. Additionally, it highlighted the need for waiting for the first keypress before starting a timer; explanations given at the start of each task were being accounted in the measurements, but less explanation was needed for each repeat of a task.

The study highlighted the need for automation. A lot of intervention was needed to navigate to the webpages used for tasks, and start the focus in consistent places. As well, extra unmeasured pointing tasks were added just to set up the task that followed. This was labour-intensive and distracting. The larger study designed tasks to use webpages where focus began in the desired starting point (for example, inside a form). Additionally, a bookmark folder was created so that all tasks could be opened in batch. The later versions of the instrumentation also added support for detecting form traversal (which effects a 'focus' event rather than a 'click' event), so more types of navigation could be studied.

The need was seen for a record of what the next task was.

Questionnaire errors were found, with some copy-paste mistakes being identified in the questions.

The pilot study also caught the omission of instructions for the use of tabbing navigation, which was needed for a fair comparison. Additionally it was decided that an explicit training period would be necessary to ensure that all participants started with the same minimum amount of knowledge.

For shorthand, the ChibiPoint systems will be named: CX (ChibiPoint with just crosshairs feature enabled) and CX+F (ChibiPoint with crosshairs and flyouts enabled).

The participant's post-questionnaire response can be seen in Figure [A.2]. His favourite system was CX+F. Overall he found easier the use of both ChibiPoint modes (CX = 4/5, CX+F = 5/5) to tabbing (1/5). The amount of keypressing required by ChibiPoint was considered reasonable ('unreasonability' rating CX = 2/5, CX+F = 1/5), and tabbing considered completely unreasonable (5/5). He refers in feedback to its being problematic to select the intended element with crosshairs ("Crosshairs without flyouts can be difficult to accurately pinpoint the desired text."). This was a known problem with how crosshairs paints even elements which are not 'clickables'. We decided to address this by mentioning it in future training.

6.3.7 Navigation tasks

For the larger study, a concrete set of tasks was chosen, to try to represent many discrete classes of pointing.

1. Go to (bookmarked) Halifax.co.uk login page
   (a) Then, click the 'password' field
2. Go to (bookmarked) Kickstarter.com signup page
   (a) Then, click the 'Re-enter password' field
3. Go to BBC.co.uk
   (a) Then, click the 'Sport' tab
4. Go to (bookmarked) University of Bath Library's 'Computer Science' resources page
   (a) Then, click 'Useful websites'
5. Go to Youtube.com
   (a) Search for 'piano' *This is just setup; not measured.*

   (b) *The page will change, which is the true start point of the test*
   (c) Then, click the first search result
6. Go to (bookmarked) Amazon.co.uk search page for 'pillow'
   (a) Then, click the first search result
7. Go to Megatokyo.com
   (a) *Using the arrow keys, scroll the page until you see the 'Archives' button.*
   (b) Then, click 'Archives'
8. Go to (bookmarked) Wikipedia.org page for 'Computer'
   (a) Then, click 'Section 2.3: The modern computer'

Each task's recorded portion is the 'Then' step; everything before is just setup to get the focus in the desired starting point. A list of URLs used are provided in Table [C.6].

An explanation of the task classifications follows:

1. **[Within form] Immediate related traversal**: the Halifax login form places focus automatically inside the 'Username' field. Move focus to the next item in the form, 'password'.
2. **[Within form] distant related traversal**: the Kickstarter signup page places focus automatically inside 'Full Name' field. Move focus to the final field in the medium-sized form, 'Re-enter password'. *Note: Computer must not be logged into Kickstarter at the time, as an existing login will cancel the registration process*
3. **Visually early element**: the BBC front-page has a navigation bar, which is the first visual element. Click one of the buttons within this, 'Sport'.
4. **Visually late element**: the University of Bath Library's 'Computer Science' resources page has many. many hyperlinks on it, in content and also in sidebars. Click 'Useful websites', a hyperlink that is relatively far down the list.
5. **[Within form] Spatially close, markup distant traversal**: navigating to a YouTube search page from the front-page search guarantees that your focus starts in the 'search' field. From here, the first search result is right next to the search form, but it is a far longer journey in the page markup. Click that first search result.
6. **Low visual indicator of focus**: Amazon is one of many websites where, in Google Chrome, focusing a page element will provide no visual feedback. With this being the case, click the first search result.
7. **Scrolled element**: Megatokyo hosts a tall webcomic, which fills some screens. There are buttons beneath this, but it is necessary to first scroll to see these. Click one of these off-screen elements, 'Archive'.
8. **Link surrounded by links**: the Table of Contents for a Wikipedia article contains many hyperlinks cramped together. Click one of these, '2.3 the mdern computer'.

An explanation of the pointing ramifications of these tasks follows. We describe why these classes of navigation create different situations for tabbing's relative, linear, markup traversal, compared to ChibiPoint's absolute, hierachical navigation.

1. **[Within form] Immediate related traversal**: focusing an adjacent element in an already-focused form is trivial for tabbing, which traverses the markup adjacent to the focused element. ChibiPoint does not use relative navigation, and has no such advantage.

2. **[Within form] distant related traversal**: focusing distant elements in forms is also trivial for tabbing, which traverses contiguous markup in a linear fashion. But hierarchical navigation can theoretically catch up quickly with a linear traversal — ChibiPoint may be able to perform comparably over this distance.
3. **Visually early element**: in lieu of any other starting focus, tabbing traversal begins at the start of a webpage's markup. Assuming that early visual elements will be earlier in markup (admittedly not guaranteed, since styling is powerful), navigation bars will be very early in the tab order of a webpage. Thus tabbing can have an advantage navigating to elements in navigation bars. Again, absolute hierarchical navigation does not have this advantage.
4. **Visually late element**: this is the polar opposite of the previous test. The relative nature of tabbing forces it into a longer journey when the element is visually far away. Absolute navigation is not harmed by this.
5. **[Within form] Spatially close, markup distant traversal**: this test shows that visual proximity is, at best, only a heuristic for how far away an element is in markup. Tabbing may not do as well here as it appears it should. Absolute navigation is not harmed by this.
6. **Low visual indicator of focus**: this test is likely to harm tabbing, where seeing the currently focused element is crucial to estimating how far the user is from the target, and also whether they are already on the target. Mistakes and backtracking may be seen if the situation is confusing enough. ChibiPoint produces its own focus indicators, so should not be so affected by the page markup.
7. **Scrolled element**: this is mainly a confirmation of effectiveness; early iterations of ChibiPoint did not follow the page as it scrolled, so it was seen that this was a class of navigation that a system can have varying success at. The only reason this differs from 'Visually late element' is because the position ChibiPoint is launched from affects the pointing journey; an expert user could scroll an element into the middle of the screen so that the crosshairs would begin in the right place, saving keypresses.
8. **Link surrounded by links**: this class of pointing could cause problems for ChibiPoint flyouts; many suggestions need to be squeezed into a small space. Though the keypress count might not be affected, reading time may be harmed. Additionally, in the case of tabbing, there will be more elements to tab past.

Admittedly these are not completely isolated classes; for example, visual indicator of focus happens also to use an element that is late in markup, as well as testing the general lack of visual feedback. So some keypress count will be attributed to unrelated factors. However, we feel this correctly represents the real usage context of a navigation system — the web is not so kind as to be designed as a set of isolated pointing cases.

Ultimately what we test is 'at least' the described pointing case. Unrelated factors such as the starting point in the page, or how many clickables there are along the way, will always add or subtract from the difficulty. We cannot know whether a keypress difference is caused primarily by unrelated factors or by the actual task intention.

Were a large discrepancy to be caused by such unrelated factors, then the tested system would be seen to be so vulnerable to outside influence that it could not complete the task without being overwhelmed by those factors. Perhaps capturing this is similar enough to finding that the system is bad at that class of navigation — since we found we cannot rely on this system to always cope with that class of navigation, due to its existing vulnerabilities.

Chapter 7

Results of larger study

7.1 Overview

We compared, for each of the eight navigation tasks, the number of keystrokes each system required to complete that navigation. Time was compared also, but as this was of secondary interest, less analysis was performed.

7.1.1 Keystrokes

Each system was found to have different strengths. Tabbing was found to excel significantly at just one task: traversing short distances in forms (for example, from username field to password field). For longer form traversals, it was matched or beaten by the various ChibiPoint systems, and it even failed to surpass ChibiPoint at accessing visually early elements. In the remaining five cases, both versions of ChibiPoint greatly outperformed tabbing. In all but one case, the 'flyouts' feature of ChibiPoint reduced the number of keypresses. For the remaining case, 'Visually Early Element', there was no significant difference between the versions of ChibiPoint.

7.1.2 Time

[PLACEHOLDER]

7.2 Analysis used

Keystroke results were analysed, for each task, using a repeated measures ANOVA. Greenhouse-Geisser correction was used to determine how significantly the mean keystrokes differed between systems. This correction was used because sphericity could not be assumed of our data; there is no homogeneity of variance, as tabbing variance emerges very differently to ChibiPoint variance (tabbing is deterministic excepting mistakes, whereas ChibiPoint navigation can take multiple valid routes).

After ascertaining whether there existed within that task a significant difference in performance between the systems, we performed — using Bonferroni correction — post-hoc tests pairwise between each combination of systems used. This told us whether, between any two systems, there existed a significant difference in keystrokes.

7.3 Analysis of Efficiency

7.3.1 Analysis Preamble

For shorthand, the ChibiPoint systems will be named: CX (ChibiPoint with just crosshairs feature enabled) and CX+F (ChibiPoint with crosshairs and flyouts enabled).

We consider statistical significance for $p < 0.01$.

We remind the reader of our hypotheses (whose detail can be read in full at Section 1.3.2):

Hypothesis 1: ChibiPoint is generally more efficient at web navigation than tabbing
Hypothesis 2: ChibiPoint's 'flyouts' feature, reduces further the required keypresses for web navigation

7.3.2 Overview of Efficiency

We provide in Figure [7.1] a clustered bar chart to show the performance difference between each system, on a per-task basis:

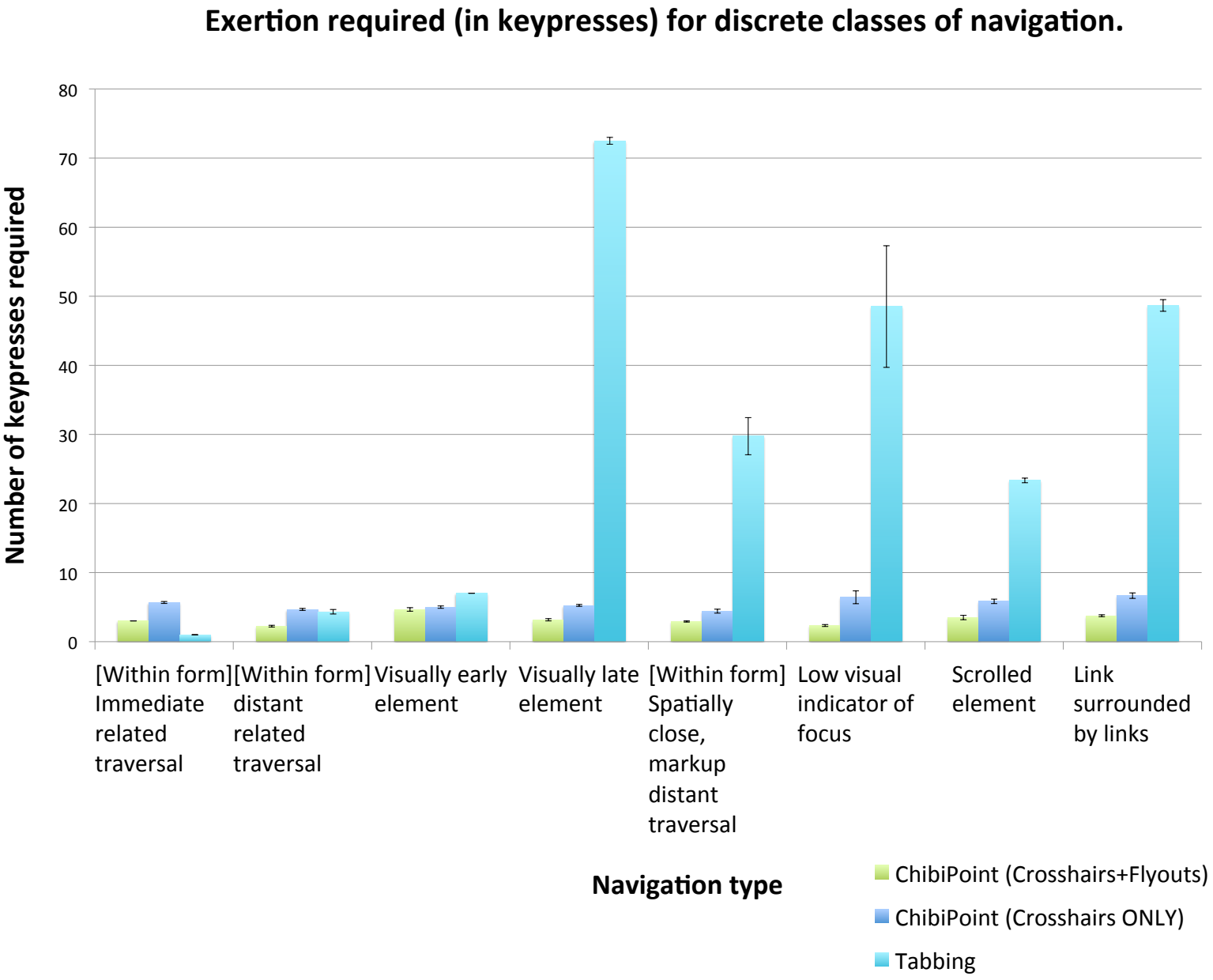

Figure 7.1: Exertion (in keypresses) per navigation task.

Visually it is evident that in the latter five tasks, the number of keypresses required for pointing by tabbing navigation is very large (>20). However, tabbing performance is lower (<10) for the first three tasks. Thus the performance across task types is inconsistent. Both ChibiPoint systems appear to have more stable behaviour — never exceeding 10 keypresses. This is likely because hierarchical navigation can account for exponentially more targets with every lookup used, so it quickly becomes likely that a target can be pointed at.

More detailed statistical analysis ensues in the following sections. Section 7.3.4 will appraise also the task differences in a qualitative sense, and explain the differences in performance.

7.3.3 Efficiency Analysis by Task

Task 1: [Within form] Immediate related traversal

System	Keypresses Mean (s.d.)
CX	5.67 (.492)
CX+F	3.00 (.000)
Tabbing	1.00 (.000)

Mean keystrokes differed statistically significantly between systems.
($F(1, 11) = 814.000$, $p < 0.001$).

CX keystrokes significantly greater than Tabbing ($p < 0.001$).
Analysis failed to produce significance value for CX+F vs Tabbing.

CX+F keystrokes significantly fewer than CX ($p < 0.001$).

Hypothesis 1 refuted in this case.
Hypothesis 2 supported.

Task 2: [Within form] distant related traversal

System	Keypresses Mean (s.d.)
CX	4.67 (.492)
CX+F	2.25 (.452)
Tabbing	4.33 (1.155)

Mean keystrokes differed statistically significantly between systems.
($F(1.574, 17.311) = 33.543$, $p < 0.001$).

CX keystrokes non-significantly different to Tabbing ($p = 1.000$).
CX+F keystrokes significantly fewer than Tabbing ($p = .001$).

CX+F keystrokes significantly fewer than CX ($p < .001$).

Hypothesis 1 supported (for CX+F only).
Hypothesis 2 supported.

Task 3: Visually early element

System	Keypresses Mean (s.d.)
CX	5.00 (.603)
CX+F	4.67 (.888)
Tabbing	7.00 (.000)

Mean keystrokes differed statistically significantly between systems.
($F(1.590, 17.489) = 54.057$, $p < 0.001$).

CX keystrokes significantly fewer than Tabbing ($p < .001$).
CX+F keystrokes significantly fewer than Tabbing ($p < .001$).

CX+F keystrokes non-significantly different to CX ($p = .797$).

Hypothesis 1 supported.

Hypothesis 2 not supported.

Task 4: Visually late element

System	Keypresses Mean (s.d.)
CX	5.25 (.452)
CX+F	3.17 (.577)
Tabbing	72.50 (1.732)

Mean keystrokes differed statistically significantly between systems.
($F(1.246, 13.701) = 15056.088$, $p < 0.001$).

CX keystrokes significantly fewer than Tabbing ($p < .001$).
CX+F keystrokes significantly fewer than Tabbing ($p < .001$).

CX+F keystrokes significantly fewer than CX ($p < .001$).

Hypothesis 1 supported.
Hypothesis 2 supported.

Task 5: [Within form] Spatially close, markup distant traversal

System	Keypresses Mean (s.d.)
CX	4.42 (.996)
CX+F	2.92 (.289)
Tabbing	29.75 (9.324)

Mean keystrokes differed statistically significantly between systems.
($F(1.017, 11.188) = 91.990$, $p < 0.001$).

CX keystrokes significantly fewer than Tabbing ($p < .001$).
CX+F keystrokes significantly fewer than Tabbing ($p < .001$).

CX+F keystrokes significantly fewer than CX ($p = .002$).

Hypothesis 1 supported.
Hypothesis 2 supported.

Task 6: Low visual indicator of focus

System	Keypresses Mean (s.d.)
CX	6.42 (3.204)
CX+F	2.33 (.492)
Tabbing	48.50 (30.485)

Mean keystrokes differed statistically significantly between systems.
($F(1.012, 11.128) = 24.176$, $p < 0.001$).

CX keystrokes significantly fewer than Tabbing ($p = .002$).
CX+F keystrokes significantly fewer than Tabbing ($p = .001$).

CX+F keystrokes significantly fewer than CX ($p = .001$).

Hypothesis 1 supported.
Hypothesis 2 supported.

Task 7: Scrolled element

System	Keypresses Mean (s.d.)
CX	5.83 (1.115)
CX+F	3.50 (1.087)
Tabbing	23.33 (11.155)

Mean keystrokes differed statistically significantly between systems.
($F(1.349, 14.837) = 1304.682$, $p < 0.001$).

CX keystrokes significantly fewer than Tabbing ($p < .001$).
CX+F keystrokes significantly fewer than Tabbing ($p < .001$).

CX+F keystrokes significantly fewer than CX ($p = .001$).

Hypothesis 1 supported.
Hypothesis 2 supported.

Task 8: Link surrounded by links

System	Keypresses Mean (s.d.)
CX	6.67 (1.371)
CX+F	3.75 (.452)
Tabbing	48.67 (2.934)

Mean keystrokes differed statistically significantly between systems.
($F(1.387, 15.257) = 2296.627$, $p < 0.001$).

CX keystrokes significantly fewer than Tabbing ($p < .001$).
CX+F keystrokes significantly fewer than Tabbing ($p < .001$).

CX+F keystrokes significantly fewer than CX ($p < .001$).

Hypothesis 1 supported.
Hypothesis 2 supported.

7.3.4 Observational Analysis by Task

Task 1: [Within form] Immediate related traversal

This task demanded traversal in a login form from a 'Username' field to its markup-adjacent 'Password field'. Naturally this is a perfect task for tabbing, which achieves the navigation in just 1 keypress.

Both versions of ChibiPoint take at least that many keypresses just to invoke the interface.

In this particular case, pointing at the field's location took CX a relatively high number of lookups (>5). In some cases elements happen to line up in the center of one of the first constructed grids, allowing traversal in few lookups. Likewise there can also be positions on the page that take more lookups to navigate to, like this one.

CX+F, once invoked, took only 2 keypresses (no variance), which is admirable efficiency considering ChibiPoint is not recruiting any data from the markup or current focus — this is, for all intents and purposes, an arbitrary clickable in ChibiPoint's eyes.

Task 2: [Within form] distant related traversal

This task, too, utilises tabbing's strength in form traversal — however, it has to navigate this time to the end of a medium-sized form. This journey will be quite predictable (provided the form is marked up correctly), so the user knows in advance they have to press tab as many times as there are form elements to traverse.

Incidentally, some variance is seen in the tabbing results, despite the journey's being entirely deterministic. This is due to one participant's backpedaling through the form, possibly due to misunderstanding which target was the destination.

Unavoidably, tabbing demands more keypresses to traverse further in forms. Thus CX catches up in performance to tabbing in this task, with a non-significant difference in their means.

Meanwhile CX+F manages to overtake both systems significantly — suggestions allow many form elements to be accessed, and CX+F can quickly drill through space enough to specify interest in the form.

Task 3: Visually early element

Similarly to distant form traversal, tabbing is strong at traversing the first elements in a page's markup. There is some lack of predictability here, as it is non-guaranteed that the first element visually will be first also in the markup. Additionally there are invisible elements in the tab order: a 'skip link' (see Section 2.1.3) to the page content, and a link to view an 'accessibility help' document.

As early as the intended button is in the tab order, there are nonetheless enough elements in the way that the ChibiPoint systems' hierarchical trawls catch up with tabbing's linear trawl.

CX+F had the potential to out-perform CX here, but due to confusion in determining which flyout referred to which element, participants grew to either over-specify their target (until no other flyouts were offered), or otherwise fall back on the crosshairs mechanism for pointing. Thus in this case, the flyouts system did not significantly aid navigation.

It is worth disclosing also, that in this scenario, many mis-clicks were made due to CX+F's hard-to-distinguish suggestions. We discarded such attempts and had the participants attempt the task again until they succeeded. Thus this 'first-time' performance is not strictly captured here.

Nevertheless, both ChibiPoint systems outperformed tabbing.

Task 4: Visually late element

This test was particularly bad for tabbing, with >70 keypresses. If visual distance from the start of the page is to be taken as a heuristic for how much markup needs to be traversed to reach that element, then we should expect tabbing to require many lookups to reach the visually late element.

In particular, this was a page with many hyperlinks. There was a site-wide navigation bar at the top, as well as a navigation sidebar, each filled with hyperlinks to traverse. Even the page itself had many hyperlinks on it. Without a skipping mechanism, or starting the traversal from a position relatively close to the element in question, tabbing was at a disadvantage in this sort of task.

The performance of the ChibiPoint systems remained similar to the figures they had been producing until now; all elements are arbitrary to ChibiPoint, so it is immaterial how late the element is situated in the visual flow of the page.

CX+F out-performed CX here, since it was quickly able to convey interest in a specific area containing the hyperlink.

Task 5: [Within form] Spatially close, markup distant traversal

This was a highly typical navigation case — choosing the top search result after performing a search. This YouTube interface was task-oriented in its visual design, with the search results immediately adjacent to the search form. Here we test how well the heuristic of 'visually close implies markup close' applies.

The markup of the page specified a longer journey than did its visual design. As such, tabbing exited the search form, and proceeded to traverse elements such as the login controls, then the site navigator, playlists and channel subscriptions. Only after this detour did it begin to navigate the search results.

Some variance was seen in tabbing, because with so many elements to be traversed, and an unpredictable journey, participants had little heuristic for how far they were from their target, and often overshot their destination in an attempt to hurry the navigation.

By comparison, both ChibiPoint systems performed significantly better, as they were not affected by this detour. CX+F's suggestions worked particularly well, as the search results were visually large, and thus featured in even broad spatial selections.

Task 6: Low visual indicator of focus

This task showed well the potential ineffectiveness of tabbing. Amazon.co.uk was styled (intentionally or otherwise) to have no visual indicator of keyboard focus for a vast majority of its interacting elements. Though highlights existed, it was found that its controls listen for 'mouseover' events rather than 'focus', and thus give feedback only to the mouse.

Participants had to infer from the browser's 'status bar' (a control which visualizes, amongst other things, the URL of any currently focused hyperlink), which element was focused. This information could come in forms such as: "

```
http://www.amazon.co.uk/Polycotton-Hollowfibre-Non-Allergenic-Pillows-Pack/d
```

", which was complex enough to be missed by many participants, who were observed to overshoot their target by pages, and even miss it on the way back also after backpedaling.

Far more effective were the ChibiPoint systems, which provided their own visual indicators over targeted elements. They were also more efficient, performing in far fewer lookups than tabbing.

Task 7: Scrolled element

This task existed mainly to confirm that all systems were compatible with off-screen elements; once scrolled, this is simply equivalent to the 'visually late element' test. It remains nevertheless an essential class of navigation to be compatible with, but the efficiency comparison is less important here.

Of particular note was that some participants realized in this task that they could recruit page scrolling as another vector to move elements under the ChibiPoint crosshairs, which re-assess targets on every viewport change.

Task 8: Link surrounded by links

This task was designed to create problems for a suggestion-based system like 'flyouts' — it would explore whether users can cope with many suggestions packed densely into one place. This is more of a concern for time performance than it is for efficiency, as the number of keypresses is not affected (unless the user chooses to over-specify their target to reduce the number of suggestions to trawl through).

For tabbing this was not an interesting problem, although in a page populated densely with links, tabbing naturally has to traverse more controls.

We found that the number of keypresses required for flyout lookup was still a significant improvement on CX and on tabbing, so this situation apparently did not compromise CX+F's efficiency.

7.4 Discussion

7.4.1 Support for Hypotheses

The primary purpose of this evaluation was to ascertain whether our hypotheses, which pertain to system efficiency, were supported. We remind the reader again of those hypotheses (whose detail can be read in full at Section 1.3.2):

Hypothesis 1: ChibiPoint is generally more efficient at web navigation than tabbing
Hypothesis 2: ChibiPoint's 'flyouts' feature, reduces further the required keypresses for web navigation

7.4.2 Support for Hypothesis 1

Task	Supports / Refutes Hyp.1
1. [Within form] Immediate related traversal	Refuted
2. [Within form] distant related traversal	Supported (for CX+F only)
3. Visually early element	Supported
4. Visually late element	Supported
5. [Within form] Spatially close, markup distant traversal	Supported
6. Low visual indicator of focus	Supported
7. Scrolled element	Supported
8. Link surrounded by links	Supported

Hypothesis 1 is supported for both versions of ChibiPoint in tasks 3–8 inclusive.
Hypothesis 1 is supported for CX+F, too, in task 2.
Hypothesis 1 is refuted in task 1 only.
Thus CX+F reduced keypresses compared to tabbing in 7 of 8 tasks, which we consider to be a large majority (>80%).

CX reduced keypresses compared to tabbing in 6 of 8 tasks (75%), which is certainly overall an improvement, but not the large majority sought by the hypothesis.
Since one mode of ChibiPoint (CX+F) meets the requirements, Hypothesis 1 holds.

7.4.3 Support for Hypothesis 2

Task	Supports / Refutes Hyp.2
1. [Within form] Immediate related traversal	Supported
2. [Within form] distant related traversal	Supported
3. Visually early element	Not Supported
4. Visually late element	Supported
5. [Within form] Spatially close, markup distant traversal	Supported
6. Low visual indicator of focus	Supported
7. Scrolled element	Supported
8. Link surrounded by links	Supported

Hypothesis 2 is supported for tasks 1–2 inclusive, and 4–8 inclusive.
Hypothesis 2 is not refuted in task 3 — it is merely not supported. Thus CX+F reduced keypresses compared to CX in 7 of 8 tasks, which we consider to be a large majority (>80%), as required.
It should also be observed that in the non-supported task, no refute was seen to the claim; in all cases, CX+F either outperforms or matches CX.

7.4.4 Discussion of Task Performance

7.5 Summary

Chapter 8

Conclusions & Future work

[Revisit this once Results section is finalized]

Chapter 9

Glossary

9.1 Acronyms

access keys a means for users to travel between distant or unrelated interface components using keyboard shortcuts. 11

browsing context a window or tab in a web browser that hosts a webpage. 34, 43

Cth DDA Commonwealth Disability Discrimination Act 1992. 16

DOM Document Object Model. 30, 32, 41

GOMS Goals, Operators, Methods, and Selection rules. 20

key binding mapping of keys, or combinations thereof, to actions. 11, 12

RSI Repetitive Strain Injury. 5, 9

SOCOG The Sydney Organizing Committee for the Olympic Games. 16, 17

W3C World Wide Web Consortium. 10, 17

WCAG Web Content Accessibility Guidelines. 10, 16, 17

web browser extension (in the context of the Google Chrome web browser): extensions are small software programs that can modify and enhance the functionality of the Chrome browser. You write them using web technologies such as HTML, JavaScript, and CSS.. 12, 13

Bibliography

[1] Shari Trewin and Helen Pain. “Keyboard and mouse errors due to motor disabilities”. In: *International Journal of Human-Computer Studies* 50.2 (1999), pp. 109–144.

[2] Eliana M Lacerda et al. “Prevalence and associations of symptoms of upper extremities, repetitive strain injuries (RSI) and’RSI-like condition’. A cross sectional study of bank workers in Northeast Brazil”. In: *BMC Public Health* 5.1 (2005), p. 107.

[3] Ann E Barr and Mary F Barbe. “Pathophysiological tissue changes associated with repetitive movement: a review of the evidence”. In: *Physical therapy* 82.2 (2002), pp. 173–187.

[4] Athula Ginige and San Murugesan. “Web engineering: An introduction”. In: *Multimedia, IEEE* 8.1 (2001), pp. 14–18.

[5] Microsoft. *Outlook Web Client*. URL: http://www.outlook.com (visited on 11/17/2013).

[6] Wendy Chisholm, Gregg Vanderheiden, and Ian Jacobs. “Web content accessibility guidelines 1.0”. In: *Interactions* 8.4 (2001), pp. 35–54.

[7] Loretta Guarino Reid and Andi Snow-Weaver. “WCAG 2.0: a web accessibility standard for the evolving web”. In: *Proceedings of the 2008 international cross-disciplinary conference on Web accessibility (W4A)*. ACM. 2008, pp. 109–115.

[8] WHATWG. *HTML5 Specification - User Interaction*. URL: http://www.whatwg.org/specs/web-apps/current-work/multipage/editing.html#the-accesskey-attribute (visited on 11/17/2013).

[9] Web Accessibility in Mind. *Keyboard Accessibility*. URL: http://webaim.org/techniques/keyboard/accesskey (visited on 11/17/2013).

[10] CabinetOffice. *Web page navigation*. 2002. URL: http://webarchive.nationalarchives.gov.uk/20100807034701/http://archive.cabinetoffice.gov.uk/e-government/resources/handbook/html/6-6.asp (visited on 11/17/2013).

[11] Willian Massami Watanabe, Renata PM Fortes, and Ana Luiza Dias. “Using acceptance tests to validate accessibility requirements in RIA”. In: *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*. ACM. 2012, p. 15.

[12] Craig Campbell. *Mousetrap: A simple library for handling keyboard shortcuts in Javascript*. 2012. URL: http://craig.is/killing/mice (visited on 11/17/2013).

[13] tokland. *Type-ahead-find browser extension*. URL: https://chrome.google.com/webstore/detail/type-ahead-find/cpecbmjeidppdiampimghndkikcmoadk?hl=en (visited on 11/17/2013).

[14] userstyles.org. *Stylish browser extension*. URL: https://chrome.google.com/webstore/detail/stylish/fjnbnpbmkenffdnngjfgmeleoegfcffe?hl=en (visited on 11/17/2013).

[15] Y Deng. *Accommodating mobility impaired users on the Web*. 2001. URL: http://www.otal.umd.edu/uupractice/mobility/ (visited on 09/25/2005).

[16] Aspasia Dellaporta. "Web Accessibility and the Needs of Users with Disabilities". In: *KURNIAWAN, Sri; ZAPHIRIS, Panayotis. Advances in universal web design and evaluation: research, trends and opportunities. EUA, Idea Group* (2007).

[17] David Hoffman and Lisa Battle. "Emerging issues, solutions & challenges from the top 20 issues affecting web application accessibility". In: *Proceedings of the 7th international ACM SIGACCESS conference on Computers and accessibility*. ACM. 2005, pp. 208–209.

[18] Eric Bergman and Earl Johnson. "Towards accessible human-computer interaction". In: *Advances in human-computer interaction* 5 (1995), pp. 87–113.

[19] Apple. *OS X User Experience Guidelines*. URL: https://developer.apple.com/library/mac/documentation/userexperience/conceptual/applehiguidelines/UEGuidelines/UEGuidelines.html (visited on 11/17/2013).

[20] Microsoft. *Designing Accessible Applications*. URL: http://msdn.microsoft.com/en-us/library/aa291864(v=vs.71).aspx (visited on 11/17/2013).

[21] The GNOME Project. *GNOME Human Interface Guidelines*. URL: https://developer.gnome.org/hig-book/3.10/principles-broad-userbase.html.en (visited on 11/17/2013).

[22] Paul Hendrix and Mike Birkmire. "Adapting Web Browsers for Accessibility". In: *Proceedings of Technology and Persons with Disabilities Conference*. 1998.

[23] Apple. *Safari Features*. URL: http://www.apple.com/uk/safari/features.html (visited on 11/17/2013).

[24] Apple. *OS X keyboard shortcuts*. URL: http://support.apple.com/kb/ht1343 (visited on 11/17/2013).

[25] Martin Sloan. "Web Accessibility and the DDA". In: *The journal of information, law and technology (JILT)* 2 (2001), pp. 01–2.

[26] Catherine Russell. "Access to technology for the disabled: the forgotten legacy of innovation?" In: *Information & Communication Technology Law* 12.3 (2003), pp. 237–246.

[27] Martin Schrepp. "On the efficiency of keyboard navigation in Web sites". In: *Universal Access in the Information Society* 5.2 (2006), pp. 180–188.

[28] Disability Rights Commission. *The Web: Access and Inclusion for Disabled People; a Formal Investigation*. TSO Shop, 2004.

[29] Kara Pernice and Jacob Nielsen. "Beyond ALT text: Making the web easy to use for users with disabilities". In: *California, USA: Nielsen Norman Group* (2001).

[30] Carlos A Velasco and Tony Verelst. "Raising awareness among designers accessibility issues". In: *ACM SIGCAPH Computers and the Physically Handicapped* 69 (2001), pp. 8–13.

[31] Ian Jacobs et al., eds. *User Agent Accessibility Guidelines 1.0*. Available at http://www.w3.org/TR/UAAG10/. W3C Recommendation, 2002.

[32] J Allan et al., eds. *User Agent Accessibility Guidelines (UAAG) 2.0*. Available at http://www.w3.org/TR/UAAG20/. W3C Last Call Working Draft, 2013.

[33] Todd Kloots. *Enabling Full Keyboard Access on the Mac*. URL: http://yaccessibilityblog.com/library/full-keyboard-access-mac.html (visited on 11/19/2013).

[34] W3C. *Comparison of WCAG 1.0 Checkpoints to WCAG 2.0*. URL: http://www.w3.org/WAI/WCAG20/from10/comparison/ (visited on 11/21/2013).

[35] Andre Pimenta Freire, Rudinei Goularte, and Renata Pontin de Mattos Fortes. "Techniques for developing more accessible web applications: a survey towards a process classification". In: *Proceedings of the 25th annual ACM international conference on Design of communication*. ACM. 2007, pp. 162–169.

[36] Christos Kouroupetroglou, Michail Salampasis, and Athanasios Manitsaris. "A Semantic-web based framework for developing applications to improve accessibility in the WWW". In: *Proceedings of the 2006 international cross-disciplinary workshop on Web accessibility (W4A): Building the mobile web: rediscovering accessibility?* ACM. 2006, pp. 98–108.

[37] Tim Berners-Lee, James Hendler, Ora Lassila, et al. "The semantic web". In: *Scientific american* 284.5 (2001), pp. 28–37.

[38] Christos Kouroupetroglou, Michail Salampasis, and Athanasios Manitsaris. "Browsing shortcuts as a means to improve information seeking of blind people in the WWW". In: *Universal Access in the Information Society* 6.3 (2007), pp. 273–283.

[39] Vicki L Hanson and Susan Crayne. "Personalization of Web browsing: adaptations to meet the needs of older adults". In: *Universal Access in the Information Society* 4.1 (2005), pp. 46–58.

[40] Peter G Fairweather, John T Richards, and Vicki L Hanson. "Distributed accessibility control points help deliver a directly accessible Web". In: *Universal Access in the Information Society* 2.1 (2002), pp. 70–75.

[41] Sara J Czaja and Chin Chin Lee. "Designing computer systems for older adults". In: *The human-computer interaction handbook*. L. Erlbaum Associates Inc. 2002, pp. 413–427.

[42] Vicki L Hanson and John T Richards. "Achieving a more usable World Wide Web". In: *Behaviour & Information Technology* 24.3 (2005), pp. 231–246.

[43] Vicki L Hanson et al. "Improving Web accessibility through an enhanced open-source browser". In: *IBM Systems Journal* 44.3 (2005), pp. 573–588.

[44] Jean et al. Mouyade. "Computer navigation method". Pat. EP2385452. Nov. 2011. URL: http://www.freepatentsonline.com/EP2385452A1.html.

[45] Mozilla. *Accessibility features of Firefox*. URL: http://kb.mozillazine.org/Accessibility_features_of_Firefox (visited on 11/20/2013).

[46] James J Powlik and Arthur I Karshmer. "When accessibility meets usability". In: *Universal Access in the Information Society* 1.3 (2002), pp. 217–222.

[47] Stuart K Card, Thomas P Moran, and Allen Newell. *The psychology of human computer interaction*. Routledge, 1983.

[48] Bonnie E John and David E Kieras. "The GOMS family of user interface analysis techniques: Comparison and contrast". In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 3.4 (1996), pp. 320–351.

[49] Bonnie John. "Why GOMS?" In: *interactions* 2.4 (1995), pp. 80–89.

[50] Jef Raskin. *The humane interface: new directions for designing interactive systems*. Addison-Wesley Professional, 2000.

[51] S Keates, PJ Clarkson, and P Robinson. "Developing a methodology for the design of accessible interfaces". In: *Proceedings of the 4th ERCIM Workshop*. 1998, pp. 1–15.

[52] Google. *Browser Extensions: Content Scripts*. URL: https://developer.chrome.com/extensions/content_scripts (visited on 04/02/2014).

[53] tokland. *Chromium extension - Find as you type (a.k.a. type-ahead-find)*. 2009. URL: https://code.google.com/p/chrome-type-ahead/ (visited on 04/02/2014).

[54] GNU. *GNU General Public License*. 2007. URL: http://www.gnu.org/licenses/gpl.html (visited on 04/02/2014).

[55] The Dojo Foundation. *RequireJS: A JavaScript Module Loader*. URL: http://requirejs.org/ (visited on 04/02/2014).

[56] The Dojo Foundation. *RequireJS License*. URL: https://github.com/jrburke/requirejs/blob/master/LICENSE (visited on 04/02/2014).

[57] Free Software Foundation. *Expat License, (aka MIT License)*. URL: http://directory.fsf.org/wiki/License:Expat (visited on 04/02/2014).

[58] nonowarn. *Demonstration for how to use require.js in content scripts*. URL: https://github.com/nonowarn/content-script-with-requirejs (visited on 04/03/2014).

[59] The JQuery Foundation. *JQuery JavaScript Library: Write Less, Do More*. URL: http://jquery.com/ (visited on 04/02/2014).

[60] Bitovi. *JQuery++ Helper Library for JQuery*. URL: http://jquerypp.com/ (visited on 04/02/2014).

[61] kdzwinel Michael Spector. *How to find out what event listener types attached to specific HTML element in Chrome extension?* URL: http://stackoverflow.com/revisions/8915461/3 (visited on 04/03/2014).

[62] Creative Commons. *License: Attribution-ShareAlike 3.0 Unported*. URL: http://creativecommons.org/licenses/by-sa/3.0/ (visited on 04/03/2014).

[63] Eli Grey. *An implementation of W3C File API's saveAs() FileSaver*. URL: https://github.com/eligrey/FileSaver.js/blob/master/FileSaver.js (visited on 04/02/2014).

[64] Eli Grey. *An implementation of Blob*. URL: http://purl.eligrey.com/github/Blob.js/blob/master/Blob.js (visited on 04/02/2014).

[65] Free Software Foundation. *X11 License, (aka MIT License)*. URL: http://directory.fsf.org/wiki/License:X11 (visited on 04/02/2014).

[66] Peter-Paul Koch. *Introduction to Events*. URL: http://www.quirksmode.org/js/introevents.html (visited on 04/02/2014).

[67] Peter-Paul Koch. *Early Event Handlers*. URL: http://www.quirksmode.org/js/events_early.html (visited on 04/02/2014).

[68] Peter-Paul Koch. *Traditional event registration model*. URL: http://www.quirksmode.org/js/events_tradmod.html (visited on 04/02/2014).

[69] Tom Pixley. *Document Object Model (DOM) Level 2 Events Specification*. URL: http://www.w3.org/TR/DOM-Level-2-Events/events.html (visited on 04/02/2014).

[70] Peter-Paul Koch. *Advanced event registration models*. URL: http://www.quirksmode.org/js/events_advanced.html (visited on 04/02/2014).

[71] Jacob Rossi Doug Schepers Björn Höhrmann Philippe Le Hégaret Tom Pixley Gary Kacmarcik Travis Leithead. *Document Object Model (DOM) Level 3 Events Specification*. URL: http://www.w3.org/TR/DOM-Level-3-Events/ (visited on 04/02/2014).

[72] Jack Franklin. "More Events". In: *Beginning jQuery*. Springer, 2013, pp. 71–82.

[73] W3Techs. *Usage statistics and market share of JQuery for websites*. URL: http://w3techs.com/technologies/details/js-jquery/all/all (visited on 04/03/2014).

[74] Prototype Core Team. *PrototypeJS: A foundation for ambitious web user interfaces*. URL: http://prototypejs.org/ (visited on 04/03/2014).

[75] Jacob Rossi Doug Schepers Björn Höhrmann Philippe Le Hégaret Tom Pixley Gary Kacmarcik Travis Leithead. *[Historical revision] Events Interface changes in the Document Object Model (DOM) Level 3 Events Specification*. URL: http://www.w3.org/TR/2002/WD-DOM-Level-3-Events-20020208/changes.html (visited on 04/03/2014).

[76] Google. *Command Line API Reference*. URL: https://developers.google.com/chrome-developer-tools/docs/commandline-api (visited on 04/03/2014).

[77] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[78] Ankit Ahuja. *StyleBot Chrome Extension: Change the appearance of websites instantly*. URL: https://chrome.google.com/webstore/detail/stylebot/oiaejidbmkiecgbjeifoh=en (visited on 04/04/2014).

[79] Alexis Sellier. *Getting started with LESS, the Leaner CSS pre-processor*. URL: http://lesscss.org/ (visited on 04/04/2014).

Appendix A

Pilot Study

Question	Response
Gender	Male
Age	23
How many hours a day do you use a computer or smartphone?	13+
How proficient are you at controlling computers with just the keyboard?	I use hotkeys in addition to a mouse/pointing device
Do you /currently/ have any motor impairments/disabilities that hinder computer usage? If so, please explain.	None
Do you presently have any difficulty reading the screen? If so, please explain.	None
Have you ever had any motor impairments/disabilities that hinder computer usage? If so, please explain.	None
What is the main desktop OS you use?	Windows
What is the main web browser you use?	Internet Explorer
Have you ever used any accessibility software or hardware to control computers? If so, please explain which, and the extent to which you used them.	Developed and used adaptive interface software for World of Warcraft.
Is there any additional information you wish to declare?	None
How proficient are you at touch-typing?	5
What is your occupation?	Software research engineer

Figure A.1: Questionnaire response of Pilot Study participant

Question	Response
Which system did you prefer using?	ChibiPoint (with crosshairs AND flyouts)
How easy was it to point using 'tabbing navigation'?	1
How was the amount of keypressing required in 'tabbing navigation'?	5
How easy was it to point using 'ChibiPoint (with just crosshairs)'?	4
How was the amount of keypressing required in 'ChibiPoint (with just crosshairs)'?	2
How easy was it to point using 'ChibiPoint (with crosshairs AND flyouts)'?	5
How was the amount of keypressing required in 'ChibiPoint (with crosshairs AND flyouts)'?	1
Have you any other feedback?	Crosshairs without flyouts can be difficult to accurately pinpoint the desired text.

Figure A.2: Post-experiment questionnaire response of Pilot Study participant
Ratings were all on a five-point scale, 1–5.

Appendix B

Usability Study

Listing B.1: Transcript of Usability Study

```
R: (Researcher)
P: (Participant)

====Flyouts On====

    First system, Flyouts ON.

R:  Start the system by pressing Apple K.
P:      Okay, is there an Apple on here?
R:  Yeah, I've mapped Ctrl to Apple.
P:      Okay, great.
P:      I'll do what I have to do to get started, and...
        *Presses Apple K*
        *Chibipoint interface appears*
P:      Okay.

R:  Your mission, should you choose to accept it, is to search for the Olympics
  .
P:      Oh, search, okay, so...
        *Presses indicated key to drill toward top-right quadrant, where search
   is*
        // No instruction required on how to point, or what keys to use!
P:      And now, the question is... oh, I see.
        *Continues drilling toward search box*

P:      Uh, it keeps coming...
        *Continues drilling toward search box.*
    [Field is now in crosshairs.]
P:      Ah!
P:      Am I there yet?
        *Guesses key*
P:      Nope. Do I have to hit return or something?
        *Hits return*
        *Search field is selected*
        // No instructions required!
        *Types Olympics, submits search*

P:      Who would search for the olympics... haha! Okay, now what?
P:      I might have to do the same again. Ctrl K?
        *Presses Apple K*
        *Chibipoint interface appears*
```

```
P:      Okay, what have I got to do?

R:  Let me think, what would be a good idea.. try and hit that article, Sochi
  2014 [top result].
P:      Hm, oh the first Sochi 2012? Sochi 2014, dude! Haha.
P:      The one with that guy, the one with the picture.
        *Drills toward article*
        *Drills toward article*
P:      And it's missed the link..
P:      Oh, I see.
        *Activates an appropriate flyout*
        // Unprompted! Only one lookup later than expert perf.
P:      It does that. Very clever of you, that extra thing.

P:      Hm, okay. This is great! I would totally take this.
        *Presses Apple K*
        *Chibipoint interface appears*
P:      Especially as it's keyboard, and it saves you from moving to it.
P:      And it's not slower than, uh, mouse; you don't have to move it to. So
  that's great.
P:      I used to keep the mouse on this; it's obnoxiously slow. I used to use
  that in 1998, when I was working for Lego, I was so screwed up I did use that
   sometimes.
P:      Anyway, yes.

R:  Now click on Sport.
R:  Err, that sport.
P:      Oh, that sport up there.
        *Drills toward*
P:      Mm, (counting noises)
        *Presses wrong flyout*
P:      Oops, missed. So what's the back button?
R:  Backspace.
        *Presses back*

P:      Let's try again.
        *Presses Apple K*
        *Chibipoint interface appears*
P:      Okay.
        *Drills toward*
P:      Mm.
        *Again activates inappropriate flyout*
P:      Ah, it's, it's.. there's something wrong.
P:      I hate to tell you this, but I'm definitely clicking on the right thing
  . Haha.

R:  Okay, uh..
P:      I guess I could therefore try to compensate for that one, cause..
        *Drills toward*
R:  I dunno how much I should say..
        *Activates correct flyout*
P:      Yeah, okay. I compensated, but there's definitely something wrong. Haha
  .

R:  Okay, how about a different website; go to Amazon.
        *Changes website*
        *Presses Apple K*
        *Chibipoint interface appears*
R:  Let's do a... I dunno; let's do a search.
P:      Okay.
```

```
94          *Drills toward*
95          *Many flyouts appear close together with tags tying them to their
      buttons in various directions*
96    P:    Hah! That's really all over.
97          *Presses appropriate flyout for search field*
98    R:    Hm, what's your favourite book?
99          [search term redacted]
100         [discussion of how to select suggested products using arrow keys on
      accessible input device]
101         *search submitted*
102
103   R:    Try and click on [author for first result redacted].
104         *Some unproductive inputs [accessible input device typing in wrong mode
      ]*
105   P:    Hello?
106   P:    Oh, I've got to...
107         *Presses Apple K*
108         *Chibipoint interface appears*
109         *Drills toward*
110         *Presses appropriate flyout*
111         //  Expert performance already!
112
113   R:    I wonder if that's enough for Amazon. Let's go to Wikipedia.
114   R:    Looks like it's already selected the input field. Shall we pretend it didn'
      t?
115   P:    Why? Why don't you just have me go to the German [Wikipedia] or
      something?
116         *Presses Apple K*
117         *Chibipoint interface appears*
118   P:    Let's go to the German one.
119   R:    Okay.
120         *Presses appropriate flyout*
121         // No drilling required!
122
123   P:    Okay, now what?
124         *Presses Apple K*
125         *Chibipoint interface appears*
126         *Drills toward search*
127         *Drills toward search*
128         *Drills toward search*
129   R:    German foods?
130         *Presses appropriate flyout*
131         *Searches currywurst*
132
133   R:    Let's go to the bottom; there should be some interesting markup there.
134         *Scrolls to end*
135   R:    Alright, let's go to Über Wikipedia.
136   P:    Alright.
137         *Drills toward*
138         *Presses appropriate flyout*
139
140       [Discussion to find an inaccessible website to navigate next]
141
142         *Google searches Bath Chronicle.*
143         *Presses Apple K*
144         *Chibipoint interface appears*
145         *Presses key for appropriate flyout*
146         *Bug in Chibipoint means the key press is caught by Google instead*
147   P:    That's interesting. So the 'J' got captured.
148         *More typing*
```

```
P:      Yeah, every time you type something, Google's capturing it.
P:      Yeah, so, can't... it's just totally failing now.

R:  Yeah, there's still other ways to use Google with the keyboard.
R:  I found this earlier. I don't know how to make my key listeners the highest
  priority.
R:  It's JavaScript injected into the page, so it's fighting the rest of the
 page.
P:      So we can cheat briefly.
        *Clicks news article of Bath Chronicle*

P:      So now..
        *Presses Apple K*
        *Chibipoint interface appears*
P:      So what do you want me to click?
        *Drills toward top-left*
R:  Click 'place an advert'.
        *Presses appropriate flyout*

R:  Register. Let's see how it does with forms.
P:      Okay.
        *Presses Apple K*
        *Chibipoint interface appears*
        *Drills toward Register*
        *Closes Chibipoint accidentally*
P:      Oops.
        *Presses Apple K*
        *Chibipoint interface appears*
        *(Immediately) drills to original position*
        // Appears to be able to reproduce practiced spatial traversals quickly
  without pausing to think.
P:      ...That's a 'Z'; okay. [Text was obscured, so appeared as numeral 7]
        *Drills toward*
        *Drills toward*
    [No flyouts have appeared for this button; clickable detection failed, so
 crosshairs are the only option. It's now in crosshairs.]
        *Presses enter*

    [A form appears]
        *Tabs a few times to enter form*
        // Candidate seems to notice that both systems can co-exist. This tab
 journey was a short one.
P:      Oops.
        *Presses Apple K*
        *Chibipoint interface appears*
    [First 6 fields of form have flyouts suggested, without drilling.]
        *Presses flyout for first form element*
        *Focuses drop-down menu*
P:      Heh. Yeah, thanks. [Presumably wanted menu to open, as with a real
 click].
        *Starts filling out form (unprompted) without Chibipoint*
        *Accidentally closes window.*

        *Restores window.*
        *Presses Apple K*
        *Chibipoint interface appears*
        *Drills toward 'Address lookup' button*
        *Drills toward [non-trivial because it's on an edge after each drill,
 and also because no flyouts are offered; clickables detection failed]*
        *Drills toward [now in crosshairs]*
```

```
202         *Presses Return*
203         *Address lookup is clicked; panel appears overlaying existing content*
204
205         *A focusing keypress is made, possibly tab or Apple L (select address
    bar). It is not clear if this is intentional.*
206         *Keyboard focus leaves browser*
207         *Presses Apple K*
208         *Chibipoint interface appears*
209         *Attempts to drill, but keyboard focus is in the address bar; types
    into address instead.*
210       [Much attempting to regain focus and resume drilling, but even tabbing
    fails]
211       [Resort to mouse click to put focus back in browser content pane]
212
213         *Drills toward address*
214         *Drills toward*
215         *Drills toward address [now in crosshairs, though never detected by
    flyouts]*
216         *Presses return*
217         *Click sent, element flashes, but no change in form*
218         // Without mouseover probing, we cannot get affordances for which
    element needs to be clicked, and confused results can occur.
219       [Attempt to verify using mouse which element is meant to be clicked]
220       [Panel closes]
221
222       [Recover panel by refreshing page and re-opening Address lookup]
223       [Clicks same element again with Chibipoint; form option remains unclicked]
224  P:       It must want [me to click] the checkbox next to it.
225         *Presses Apple K*
226         *Chibipoint interface appears*
227         *Drills toward*
228         *Drills toward*
229         *Drills toward*
230       [Button is not yet in crosshairs. Drilling labels now hidden since grid is
    small.]
231  P:       Hmh.
232         *Presses return*
233  P:       Yeah, I have no idea. It doesn't seem to be..
234  P:       So, does that..
235         *Hovers mouse over to confirm feedback*
236  P:       [The button] doesn't seem to notice [with Chibipoint] that something's
    there.
237  P:       Obviously there's no signal that [an emulated mouse cursor is above].
238  P:       'Cause the [cursor] changes into a hand when you hover over it.
239       [Repeats previous navigation:]
240         *Presses Apple K*
241         *Chibipoint interface appears*
242         *Drills toward*
243         *Drills toward*
244         *Drills toward*
245       [Button is not yet in crosshairs. Drilling labels now hidden since grid is
    small.]
246         *Presses return*
247         // Seems to think that when the drilling labels are gone, it is not
    possible to continue drilling.
248
249         *Reopens Chibipoint, drills lightly into an alternative journey.*
250         *Closes Chibipoint without clicking*
251
252         *Reopens Chibipoint, redoing a previous journey.*
```

P: Yeah, nothing I do seems to get the tick.
R: Try pressing [bottom-left quadrant (now unlabelled)]
Drills
R: And again.
Drills
R: You can inch closer.

R: Boxes aren't perfect sub-boxes; they grow a little bit.
P: Okay. I didn't see Z as an action, so...
R: Ah. Yes, it... I can't make text any smaller.
P: Well, is that enough information?

====Flyouts Off====

R: Now try a version with the Flyouts turned off.
P: Okay.
[On BBC homepage]
Presses Apple K
Chibipoint interface appears
R: Try and watch Top Gear. [This is at the very bottom edge of the grid]
P: Okay, down there.
Drills toward
Drills toward
Drills toward
Presses return
[Page navigates]
// Even buttons in tricky positions were able to be clicked easily.

Presses Apple K
Chibipoint interface appears
[Top Gear Flash pane is painted wholly by Chibipoint; inner structure not detected]
Presses space bar to attempt to play video
[Page scrolls upon space bar since video does not have focus]
Drills toward Flash pane
Drills toward centre of Flash pane
Presses return
[Flash pane is clicked, but no meaningful change occurs]
Presses key to scroll up
[Page does not scroll, despite input]
P: How do you scroll up?
R: Your focus is trapped by Flash.
[Mouse recruited to recover focus]
[Page scrolled back to beginning]

Presses Apple K
Chibipoint interface appears
P: So, is the big yellow thing..? I'm sure if it could just see which button I'm trying to click..
P: No, it's not doing anything.
Presses Apple K
Chibipoint interface appears
Closes Chibipoint

R: So I'll spoil that; Flash players can't take clicks [from ChibiPoint].
P: Oh, okay.

```
R:  So you've discovered that.

    [We return to BBC homepage]
R:  Try to click something on the bottom of the page - that's not on-screen.
R:  Music.
        *Presses Apple K*
        *Chibipoint interface appears*
        *Scrolls using Page Down*
        // Unprompted, understands that page scrolling keys co-exist with
  ChibiPoint.
        *Drills toward an element in the same list as 'Music'*
P:      Too late; I'm going to go into 'Learning'.
P:      Although it's actually not what you said.
    [Actual 'Learning' text is not highlighted, but clicking the box that
  contains it worked in this case; did not have to drill too specifically]
        *Presses return*
    [Navigates to Learning]

R:  Let's see if there's anything hard to click on; keep scrolling.
        *Presses Page Down*
R:  Click the 'Accessibility Help'.
P:      Okay.
        *Presses Apple K*
        *Chibipoint interface appears*
        *Drills toward*
P:      Hmm..
        *Drills toward*
    [Painting of link container obscures the white hyperlinks within]
R:  Oh god.
        *Drills toward*
    [Presently highlighted element is the link below Accessibility Help]
P:      And then, how do you move it around? Did you say the Z key..
R:  The keybindings continue to be exactly the same.
P:      Oh, but I didn't memorise them.
R:  Uhh.. they're this grid.
    [Demonstrates grid of 9 keys on accessibility hardware that map to on-
  screen controls]
P:      Oh, okay, so..
        *Drills toward*
        *Presses enter*
    [Accessibility Help is clicked]

R:  So, did you notice that they were...
P:      No, I totally didn't notice that they were,
P:      and I shouldn't, since there was a cue.
        // Mapping was not noticed nor memorised; singular lack of incidental
  information.
P:      Okay.
P:
P:      Okay, so..
    [Decides to click one of the webpage accessibility articles]
        *Presses Apple K*
        *Chibipoint interface appears*
        *Drills toward*
        *Drills toward*
        *Drills toward*
        *Presses return*

P:      Well anyway, this doesn't seem to be a problem, so..
R:  Right, I'll end.
```

====Feedback====

R: What do you think of the two systems you tried (flyouts on, off)?
P: I couldn't tell the difference, I'm afraid.
P: What was the... oh, I dunno... oh, the flyouts with commas and periods and things?
R: Yeah.
P: Yeah, I liked the commas and periods and things. They were pretty good guesses. Although it was a bit cluttered. I'd like to be able to switch back and forth depending on what I was trying to do.
R: Uhh, by switch back and forth, do you mean 'hide them'?
P: Yeah. Exactly, so like mostly it's great. Although anyway you can hide it with Ctrl-K, so I guess that's enough.

R: Was the problem that [the flyouts] obscured [what you were trying to point at]?
P: It just looks a bit cluttered sometimes. But it was really useful, so I think I'd probably mostly just have them on. I could just imagine, I dunno some time I might want to turn them off...

R: Would it be useful to have, like, holding down SHIFT dispels the interface?
P: What?
R: Uhm, there's a lot of heads-up display on the page; you could hold down a modifying key to hide stuff temporarily, but not lose where your grid is.
P: Uhh, I don't know what... one thing is that Ctrl-K doesn't work very well for me, 'cause I use emacs; that's why it was after a little while when I was doing the wrong things because I doing the wrong commands, because I was confusing all the commands.
P: So for me it would be better if it was something different, like Alt+ something, just because I hardly ever use Alt. Then I'd remember it; I could remember that. And also, when I'm on emacs, I'd be able to control it.

R: There's a limit to what keyboard shortcuts I'm allowed to choose. Actually, it's a limit to what I can 'suggest' as a keyboard shortcut, but the user can override the system ones if they manually choose them.
P: Okay. Yeah.
R: So that could be done.

P: Yeah, but it was really nice. It was a nice way to do it.
P: I could imagine using that a lot.
P: Although now I'm pretty much just using the pen tablet.
P: But it saves you find the tablet - the pen, tablet; it's just right there.

R: Were there any things that you didn't understand?
P: Oh yeah, like I said, I didn't notice that the grid was continuing; or rather I didn't pick up that - they just looked like random letters to me. I didn't know why you were doing it.
P: But had I noticed... 'cause, the thing is that in the old days, my first guess would've been, was the right hand.
P: In the old days, the arrow keys used to be right around there.
P: I should've picked up that you were using the left hand instead, but I didn't.
R: I'm less worried about that, because it usually would be the numpad, but it 's [accessible input device] mode.

406 P: Yeah, okay. The problem would be the - because they were slowing down the typist, when you're in Qwerty, all the characters that you use are mostly on the left, so you wouldn't normally expect them to use the right hand for the arrow keys.
407 **R:** Okay, so they can be swapped.
408
409 **R:** Did you ever desire to be able to 'undo' a drill-down, and retreat?
410 P: Yeah, once, and what I did was just Ctrl+K to start over, but there was , like, one time.
411 **R:** Okay. There's a feature I should've disclosed to you; there is a key that can 'back up'.
412 P: Okay. Yeah, no, I would've wanted that if I was, I got it.
413 P: But that was pretty fast; considering how long it took me to get... quite a lot of usability stuff downloads, the first time you try using a pen tablet(!), yeah, so that was, I was able to get into that; that was fine.
414 P: Again, I am old, and I did used to navigate with the arrow keys on the other hand(!)
415
416 **R:** Would you say it's intuitive?
417 P: It was largely intuitive; I mean, there was a couple things you didn't know. It might be nice to have, like, a little helping thing that could pop up and tell you the cheat sheet, to tell you if there's any other stuff.
418 P: 'Cause usually what I do is, I hack around a bit with the command, and then I go and read the manual, and see if there's anything I've missed.
419 P: So this was a very successful first hack-around, but then you would want somewhere to be able to read and find out what you'd missed.
420 **R:** Maybe I'll print a cheat-sheet rather than coding one.
421 P: Yeah, sure.
422
423 **R:** Were the 'clickable' suggestions - made by the flyouts - were they good enough guesses?
424 P: Yeah! Yeah, they were highly accurate; I was very impressed by that.
425 P: I especially liked that they were sometimes, you picked the wrong region, and they still guessed that it was just slightly off the region; it was like, way! Thinking outside the box.
426
427 **R:** Did you find that there were any things that it didn't seem to be able to see?
428 P: I didn't notice any in particular - except for something, you know, that one thing that we tried over and over again, and couldn't get anything to see.
429 **R:** That button didn't seem to be marked up properly.
430 P: Yeah.
431 **R:** It has a lot of intelligence to work out if something's clickable, but...
432 P: But somehow, like I said with the mouse you could hover over it and it would change into a pointy thing, so there must be something there, but it wasn't much.
433 **R:** I put in a feature to force it to 'mouseover' anything that the crosshairs are on. But it doesn't seem to... I told it to trigger the mouseovers, but it never happens.
434 P: I never really 'got' the crosshairs, so I hadn't realised - right until the end you showed it to me - but I didn't earlier pick up on that.
435
436 **R:** Yeah, it's problematic that as well as the flyouts, there's also the crosshairs.
437 P: Okay. I wasn't really noticing [the crosshairs], because the other system was working so well for me, so I was sort of reflecting, and concentrating on what I saw.
438 **R:** And yet, once you were switched to the system without flyouts, you were happy to use crosshairs.

439 P: Well, once you showed me how to use them, that you could still navigate even if there's no letters, and then I was like, okay, well that was like what I was used to before with the [accessible input device], so I learned my lesson.

440

441 **R:** Would you say you could understand how to get to a place?
442 P: Yeah, obviously; you just saw me do it(!)
443 **R:** Was the grid-splitting an intuitive way?
444 P: Yeah, no, I thought that was fine; that was great. Uh, yeah, that made sense. I liked that it was very bright, and there was this little {unclear}.
445 P: I'm probably not your best - it depends how you think about it. I'm probably not the most stereotypical - it's not like taking home a heavier[?] - somebody who doesn't use a computer - to try it.
446 **R:** Well, it's designed for [people with accessibility requirements]. You'd probably be a pretty good sample for accessibility needs.
447 P: Well, if it's designed for disabled people with a lot of experience with different tech, so that's what I'm saying - you'd need to find some other people who haven't had the same breadth of experience of available tech .

448

449 **R:** If we want to model performance of an expert, that's fine though.
450 P: Yeah, no, that's fine. I'm just saying that you should realise I'm not a typical user. In fact, that's one of the things that Google will tell you, probably(!) is that their problem is that all the people they hire are not at all - can't even conceive of the problems that their users have. Figuring out, for example, that keyboards {unclear}(!)
451 P: Uhm, yeah, we're not the normal. I'm particularly mechanically inclined , so I tend to figure these things out pretty quickly.
452 P: But still, it was good. I'm just saying that the hacking through it might not... well of course, to be fair, who does hack through your stuff? People who are good at hacking. Look at the manual.

453

454 **R:** How do you feel this system compares to navigating using tabbing around a page?
455 P: Oh, it's obviously faster because it's hierarchical. My first-years should be able to answer that question.
456 **R:** Would you prefer this system?
457 P: To tabbing? Of course, yeah. As long as it didn't interfere with anything else.

458

459 **R:** Bonus question: how does it compare to the mouse?
460 P: Uh, like I said, the main benefit is... well, there's actually two benefits. The main benefit is that for me, because of whatever - I don't know if everybody has this - is that especially by the time you get your hand over, and you find the mouse, that sort of thing, it was just much faster to have it right on the keyboard.
461 P: So, that's... and the other thing is that, honestly, sometimes it's really hard to click on, so being able to choose a letter is just better than , than just, life, you know, even with a pointing device.
462 P: But, yeah. So, basically, dunno, I'd have to try it. Uh, it might be... the other, the downside is that one of the problems with typing injuries is that you are too much in the same posture doing the same thing all the time, so it is good to use other devices if you have them, just to move around a bit.
463 P: In fact especially when you have to go searching for the pen, that's always good.

464

465 **R:** Would you use it alongside tabbing?
466 P: Tabbing? No, probably I would just use this, once I was used to it. I probably wouldn't use tabbing unless [Chibipoint] was breaking.

467 P: But if you see that [what you want to focus] is the next thing, then you tab because you have no reason to go through a hierarchy. 'Cause it was one, if it is in a form, then it's one keystroke to get to the next thing. If you [use Chibipoint, it's] always 2 keystrokes because you first have to call it up, and then you have to go again.

468 P: So I would use the tab - I would always go for the fewest keystrokes. So you can answer your own question by saying how many times would you have to press tab to get to the next point.

469

470 **R**: Okay. So you would see yourself using the two systems together?

471 P: Yeah, I think so because - but, well, I - yeah, depending on what you're trying to do. But for the average page, you would never ever tab through it, but it's gonna be just how many - like I said - I'm always gonna try and use the fewest keystrokes in the shortest amount of time. So, but, that's, you know, for a form that's gonna be tab, because you have to fill everything in.

472 P: But for any normal page, like on the BBC page, where you're skipping around, then of course you don't wanna use the tab.

473

474 **R**: And, how predictable was it?

475 P: I thought it was pretty predictable, I found. Except from, you know, there were bugs. The capture problems.

476 **R**: Was it more predictable than tabbing?

477 P: Yeah. Absolutely. Who knows what's gonna - yeah, when people {unclear}, I never figured it out. Use the tab to, like, {unclear}, you're just like, why did they need that? You know. And sometimes they skip over things, that were just not so {unclear}. I do a lot of {unclear}, so you know {unclear}.

Appendix C

Quantitative Study

How many hours a day do you use a computer or smartphone?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	4-6	2	16.7	16.7	58.3
	7-9	5	41.7	41.7	100.0
	13+	5	41.7	41.7	41.7
	Total	12	100.0	100.0	

Table C.1: Device usage by participant

Descriptive Statistics

	N	Minimum	Maximum	Mean	Std. Deviation
Age	12	21	28	22.58	2.151
Valid N (listwise)	12				

Table C.2: Age of participant

Gender of Participant

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Female	3	25.0	25.0	25.0
	Male	9	75.0	75.0	100.0
	Total	12	100.0	100.0	

Table C.3: Gender of participant

Participant Keyboard Navigation Proficiency

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	I prefer a mouse/pointing device	5	41.7	41.7	41.7
	I use hotkeys in addition to a mouse/pointing device	6	50.0	50.0	100.0
	I prefer keyboard controls where possible	1	8.3	8.3	50.0
	Total	12	100.0	100.0	

Participants were asked to pick which of these sentiments they identified best with. The statements attempted to describe a proficiency scale for keyboard usage:

1. I need a mouse/pointing device
2. I prefer a mouse/pointing device
3. I use hotkeys in addition to a mouse/pointing device
4. I prefer keyboard controls where possible
5. I strongly prefer full keyboard support

Votes were cast only for the middle descriptors, 2–4.

Table C.4: Keyboard Navigation Proficiency of Participants

Descriptive Statistics

	N	Minimum	Maximum	Mean	Std. Deviation
Touch-Typing Proficiency	12	1	5	3.42	1.084
Valid N (listwise)	12				

Touch-typing proficiency was rated on a five-point scale ranging from the sentiment ‘I look at every key before pressing’ to ‘I always type without even a reference glance at the keyboard’.

Table C.5: Touch-Typing Proficiency of Participants

Task №	URL
1	https://www.halifax-online.co.uk/personal/logon/login.jsp
2	https://www.kickstarter.com/signup?ref=nav
3	http://www.bbc.co.uk/
4	http://www.bath.ac.uk/library/subjects/comp-sci/
5	http://www.youtube.com/
6	http://www.amazon.co.uk/s/ref=nb_sb_noss_2?url=search-alias%3Daps&field-keywords=pillow&rh=i%3Aaps%2Ck%3Apillow
7	http://www.megatokyo.com/
8	http://en.wikipedia.org/wiki/Computer

Table C.6: URLs used in tasks (full study)

Participant №	Which system did you prefer using?	How easy was it to point using Tabbing?	How was the amount of keypressing required in Tabbing? (1= Reasonable, 5=Unreasonable)	How easy was it to point using CX?	How was the amount of keypressing required in CX? (1= Reasonable, 5=Unreasonable)	How easy was it to point using CX+F?	How was the amount of keypressing required in CX+F? (1= Reasonable, 5=Unreasonable)
1	CX+F	2	5	4	2	5	1
2	CX	2	5	4	1	4	1
3	CX	1	5	5	2	5	1
4	CX+F	1	5	3	3	4	2
5	CX+F	1	5	5	2	5	1
7	CX+F	4	5	4	2	5	2
6	CX+F	2	4	4	3	5	2
8	CX+F	2	5	4	2	5	1
9	CX+F	1	5	4	3	5	2
10	CX+F	4	4	4	3	5	2
11	CX+F	2	4	4	1	5	1
12	CX+F	3	5	5	1	4	1

Table C.7: Post-experiment questionnaire responses of Quantitative Study participants Ratings were all on a five-point scale, 1–5.