Accessibility Works: Enhancing Web Accessibility in Firefox

John T. Richards, Vicki L. Hanson, Jonathan P. Brezin, Calvin B. Swart, Susan Crayne, and Mark R. Laff

IBM T. J. Watson Research Center, 19 Skyline Dr, Hawthorne, NY, USA
{ajtr,vlh,brezin,cals,crayne,mrl}@us.ibm.com

Abstract. This paper reviews several techniques we have discovered while trying to extend the Firefox browser to support people with visual, motor, reading, and cognitive disabilities. Our goal throughout has been to find ways to make on-the-fly transformations of Web content including adjustments of text and image size, text style, line and letter spacing, text foreground color, text background color, page background removal, content linearization, and reading text aloud. In this paper, we focus primarily on the changes we make to the browser's Document Object Model (DOM) to transform Web content. We review the kinds of approaches we have used to make DOM modifications sufficiently fast and error free. We highlight the problems posed by Web pages with a mix of static and dynamic content generated by client-side scripts and by Web pages that use both fixed and relative placement of page elements, pages of the sort we expect to see in increasingly in the future.

Keywords: Web accessibility, transcoding, Document Object Model.

1 Introduction

The W3C has developed guidelines for accessible Web pages [1] [2]. Websites that conform to these guidelines provide a number of critical features allowing blind or deaf users to get Web information in another modality. These guidelines, however, often fall short of enumerating the kinds of modifications needed by users with what might be viewed as less severe deficits (e.g., enabling low-vision users to use the vision they do possess to greater advantage, or simplifying page structure to help users unable to navigate complex layouts). Even if the guidelines were broader, they would not solve the problem, because many websites do not conform to them, and the cost of coming into conformance is prohibitive. Even conforming websites do not typically allow individual users to control their browsing experience as outlined in the User Agent work of the W3C [3] [4]. Allowing individuals to adapt Web content for their unique needs has inspired our work on Web page transformations.

For the past several years, we have been developing extensions to the Internet Explorer[tm] (IE) browser to support on-the-fly transformation of Web content to support Web access for people with limited vision and dexterity, as well as for people with language or cognitive issues that impact their ability to use the Web. The key

C. Stephanidis (Ed.): Universal Access in HCI, Part III, HCII 2007, LNCS 4556, pp. 133–141, 2007.
© Springer-Verlag Berlin Heidelberg 2007

idea was to provide a single software application that could serve a wide number of users with a variety of needs.

Our early work explored using a proxy server to transform a Web page's source before it got to the browser, but a number of problems with this approach motivated us to move instead to all client-side transformations [5]. The resulting transformations were based on a number of techniques including: modification of the Windows[tm] system registry, dynamic generation of user style sheets, manipulation of the browser's internal model of the loaded Web page, and the launching of separate executables for filtering user input and magnifying images. This Internet Explorer implementation, previously reported [6] [7], is available in seven languages and used by thousands of people worldwide.

Two years ago, we began a fundamental redesign of this software for use in the Firefox browser [8]. Part of the appeal of Firefox is that it runs on multiple platforms including Windows, Mac OS, and Linux. While we use Mozilla's XPCOM mechanism [9] to access platform specific speech functions, and while our mouse and keyboard adaptations require platform specific interaction with the operating system [10] [11], we concentrated most of our Firefox work on changing the browser's internal model of the loaded Web page – its Document Object Model (DOM). In this paper, we discuss our approach and some of the techniques we used to achieve enhanced Web accessibility for users of Firefox.

2 Changing Page Elements

We have created several Firefox accessibility transformations for changing the appearance of Web pages. One useful transformation allows foreground and background colors to be changed to heighten contrast, or produce a custom color combination for people with certain types of low vision or dyslexia. Another lessens static clutter by removing background images. A third eliminates dynamic distractions by stopping the movement of animated gifs. What these transformations have in common is that they do not change the placement of elements on the page since visual elements retain their sizes and locations.

A more complex class of accessibility transformations involves changing the size of page elements. Transformations found to be useful from our earlier IE implementation include magnifying the entire page, changing the spacing between lines of text, changing the spacing between letters of text, and changing the font size of text.

We have found that magnifying the entire page, easily done in IE by dynamically generating a user style sheet with the "zoom" attribute, is impractical in Firefox. Since at least earlier versions of Firefox lack a native zoom feature we would be forced to re-layout the entire page, element by element.

Unlike zooming, the line spacing and letter spacing, transformations can be accomplished straightforwardly in Firefox. The line space transformation is made by modifying the page body's line height style along with attributes of certain TABLE, DIV, PARAGRAPH, and LIST items in the DOM. The complete set of elements requiring attribute modification has been determined somewhat empirically so may not cover all current and future Web page designs. But it seems to work well for most.

Letter spacing is somewhat simpler than line spacing in that we can simply modify the page body style for letter spacing. We have not yet encountered pages that modify this style at the element level so we can bypass the element modifications needed for line spacing.

Changing text size is trickier than might be supposed. There is support in HTML itself for asserting *relative* text size, and for those situations, changing the default size scales the text nicely. But the range of enlargement is too small for many of our users since only four larger sizes are available. To go beyond this range we have to use explicit font size specifications, and to enforce them, we have to process all style sheet rules and FONT elements in the DOM. But font size changes can easily interact with explicit placement of nearby elements, causing text that was adjacent to another element to overlap. If pages include fixed width elements the problem can quickly get out of hand as shown in Figure 1.

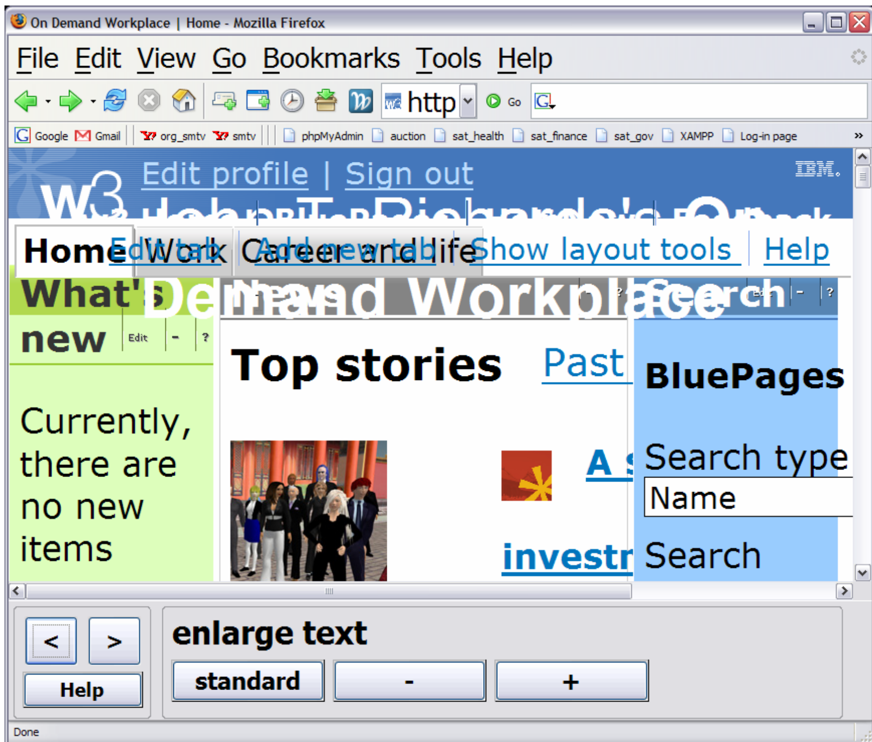

Fig. 1. Potential problems with naïve text enlargement

3 Changing Page Geometry

Our software is intended to serve people with a variety of disabilities. In an individual person these disabilities often interact. For example, a person may need a page to be

enlarged so as to be easily read, but may also need to minimize use of the mouse due to a motor disability.

Consider the case of a multicolumn page. As it is enlarged, it can extend beyond the right hand edge of the browser window, causing a horizontal scroll bar to appear. Horizontal scrolling, which is already undesirable for those without any motor disability, can become a major barrier to effective browsing for those who find mouse and keyboard operation difficult. To accommodate this need, we created a feature that causes multiple column content to be ‘linearized’ into a single column as shown in Figure 2.

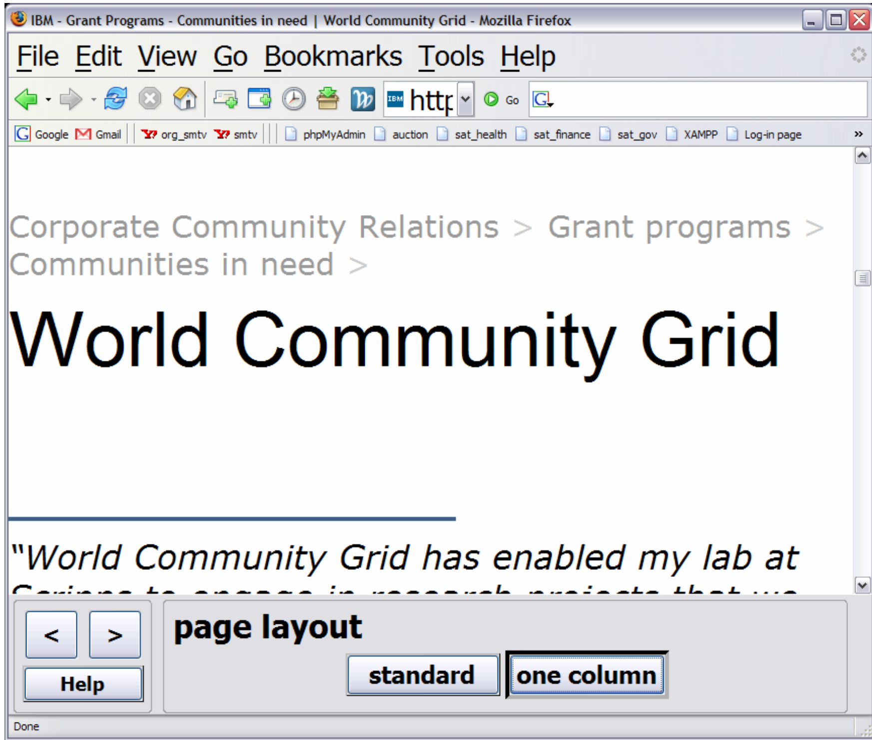

Fig. 2. A page after substantial text enlargement and linearization

Like the previous size manipulations, page linearization is changing the geometry of the page. But the change here is so extensive that we cannot affect it by merely manipulating attributes of individual elements. We must actually modify the overall structure of the DOM. And we must do this in a way that maximizes the likelihood that scripts and style rules attached to the document will continue to operate correctly. For example, making each TD or TH element its own row by inserting TR elements is a very bad idea: just imagine that the original TR had an id like “PopulationTableHeadings” that a style rule depended on. Either we recognize the dependence and replace it with a dependence on a class (which we have to create), or we have to maintain the containment hierarchy implied by the original DOM. A better

solution is to replace every table element with a DIV that has all the relevant attributes of the element copied into it. That way the parent-child relations in the DOM are unchanged. Of course, we still have to take into account the border implied by the TABLE element (and possibly with the table as well).

4 Banner Text and Spoken Text

For some users, the most effective transformation of Web content is to greatly enlarge small blocks of text and/or to read portions of text aloud using text-to-speech software. Reading text aloud has proven to be especially useful for people with very low vision and those unable to read for reasons of either language competency or general reading ability [6] due to dyslexia, low literacy, or cognitive disability. Using IBM's ViaVoice^tm product, our software will read aloud selected portions of a page or unselected text regions up to the whole page. Assuming that multiple ViaVoice languages are installed on a user's computer, the software can even read a page with the correct language pronunciation. It does this by using various techniques for determining the language of the page and then using the appropriate ViaVoice language. In the simplest case the language is specified by the <HTML lang="…"> attribute [12]. Relatively few pages, however, follow this W3C recommendation, so other techniques, based on an analysis of language specific characteristics of the page content, must be used.

Blind users, of course, rely exclusively on spoken text provided by screen readers. Such screen readers require the user to memorize a large number of specialized keyboard commands to control navigation and speech output. While the payoff for dedicated users is obvious, casual users will not memorize complex keyboard commands. Moreover, some cognitively disabled users would not be able to memorize these commands. Fortunately, our target users can all be assumed to have at least some usable vision. The technique we have evolved – something you could call "hover to hear" – takes advantage of this fact. By bring the mouse pointer to rest above the text area of interest our users can see greatly enlarged text and/or hear text read aloud from the beginning of the corresponding text element.

To support this "hover to hear" technique we must first determine what the user is pointing at. In the simple case, the mouse is over a relatively small text node and we can start reading from the beginning of that node. For larger blocks of text it is difficult to determine where in the text node the mouse pointer lies. As such, we would be forced to read from the beginning of the block even though the user is pointing somewhere within it. As you might imagine, this is quite frustrating for our users. To make spoken text more usable, we insert SPANs into the DOM to break up large text blocks into smaller units allowing finer localization. While in principle this should affect neither scripts nor style rules, in practice it can result in the text and explicitly positioned elements being drawn incorrectly vis-à-vis one another. The general problem of how to deal with text within explicitly positioned elements remains open.

Assuming we can find the starting point in the text corresponding to the user's expectations (and we usually can), we try to keep reading until the user asks us to stop by moving the mouse outside the text area, or clicking, or hitting any key. It has also

proven desirable to dynamically highlight what is being read, even over page boundaries (forcing us to scroll if necessary). This is fairly straightforward as long as one can continue to scroll down. But, if the next text in document order is above the current text in the display, we have found it is best to stop, because scrolling the display back may leave the user bewildered as to where the current text is on the screen.

Of course, in order to do any of this work, our software must see suitably localized mouse as well as keyboard events. In the case of IE, we ran a thread parallel to the browser thread that attached event handlers to every DOM node that we cared about (which, alas, were most nodes). In this way, we could get control and react to the mouse moving over a piece of text and hovering there. We were forced to do this at the element level in IE because one cannot tell from the Event object where in the DOM the event occurred, only that it occurred inside whatever element handled the event. Firefox's Event API provides a bit more information about where the event occurred relative to the DOM so we only need event handlers for the root (body) element and each FRAME or IFRAME. The reason for the latter is that when we are trying to read the "next" text, we sometimes have to go to the document containing the document we are now in (the parent frame of our current frame). Hence we need to track what frames have the text we are reading.

Handling the mouse events involves a little finesse. By trapping mouse-up events one can track selection, and (assuming the user has told us to) we can read the selection. Click events, on the other hand, always end the current reading session and may be part of a double click that selects a word. A simple but not trivial state machine handles this bookkeeping. We also use a timer to know when to start reading the next text element. Once speaking starts, we start a timer, and when it fires, we ask whether we are still speaking (unfortunately there is no end-of-spoken-text event available to us). If we are done, we proceed to find the next block of text to be read and do so. If not, we restart the timer.

5 Dynamically Constructed DOMs

A browser's DOM is a fairly straightforward data structure. As specified by the W3C [13], the DOM represents hierarchical HTML documents basically as trees and provides element access and manipulation through an API intended to be conveniently used by script writers. As a practical matter, all DOM renderers allow an HTML DOM to be "live". That is, it is automatically re-rendered whenever any change is made to it that might affect the rendered appearance. One common way to make such a change is to create a piece of ordinary HTML as text, and then to use the scripting API to insert this text as the content of an element already present in the DOM. Since scripts using this technique can run quite some time after a document is loaded it is by no means clear when a DOM is *finally* ready to be modified for purposes of enhancing the readability, or general appearance, or overall layout of the document.

Consider the following simple example. Figure 3 shows a help page that is provided with our Firefox software. In it we describe how text foreground and background colors can be changed to heighten overall contrast or provide custom color combinations suitable for people with some types of dyslexia. This help page

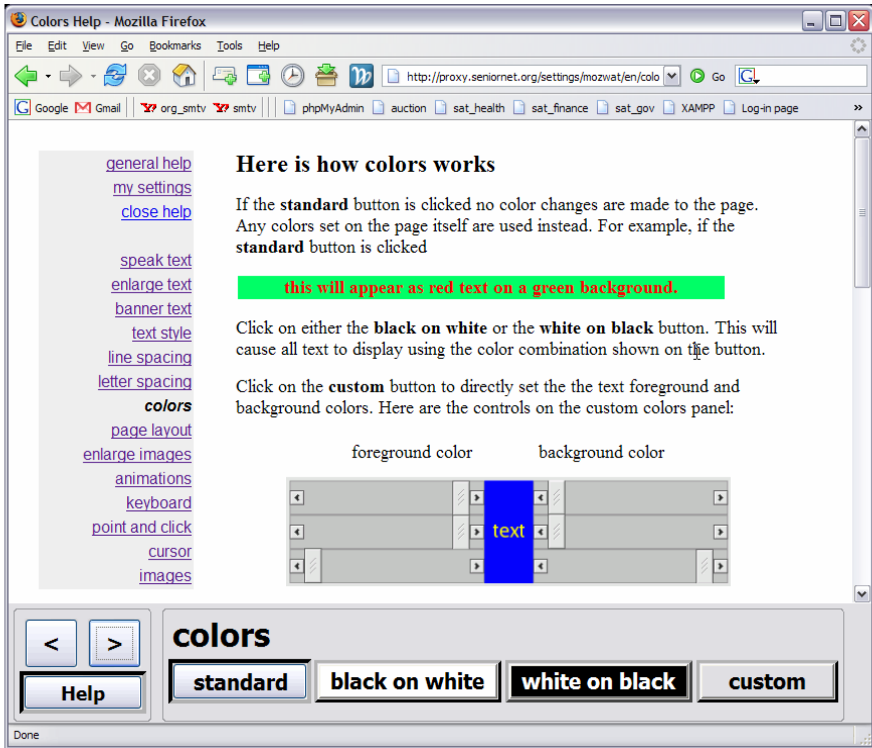

Fig. 3. Help page for changing colors with dynamically constructed navigation bar on the left

also has a navigation bar on the left hand side to allow users to conveniently move to help for the other transformations we provide.

Now consider how this fairly normal looking navigation bar is actually created. In the early days of our work, it was simply coded as part of the static HTML of each individual help page. This worked fine as long as the number of pages was small and their order didn't change too frequently (the order corresponding to the order of transformation panels as one clicks the left/right arrow buttons on the bottom panel). But as the number of transformation panels grew, as we started responding to feedback from our users as to their preferred ordering of panels, and as we started providing translations for multiple national languages, the time required to make even a simple help change became prohibitive. To remedy this, the help pages were modified so the navigation bar became a placeholder, an empty DIV, whose "inner HTML" was set by a script in the onLoad handler of each help page. This script iterates over an ordered array of national-language independent help page identifiers, gets the translated string for the current user's language's name for that page, and adds it as a table row either as a clickable anchor or as an emphasized, italicized string for the current help page.

While this script runs without perceptible delay, the insertion of the inner HTML into the DIV (and hence into the underlying DOM) came too late for our extension to see it in its DOM traversal, this traversal being similarly triggered by the onLoad

event. As such, the page would display with the navigation bar having the original gray background rather than the background dictated by any color transformation the user may have set. Since the entire help system was designed to dynamically illustrate how each transformation would be applied to pages on the Web, this was an unacceptable outcome.

To fix this anomaly, we did two things. First, we optimized the script building the navigation bar's inner HTML so it ran in the shortest possible time. Next we delayed our DOM walk by a fraction of second to allow the full DOM to be built. It does not take much imagination to see that this is not a very robust solution. But lacking a dependable event that tells us when a page's onLoad scripts have finished executing, it may be the best we can do.

In general, we are finding that more and more Web content is being constructed using dynamic techniques of the sort illustrated by this simple example. The increasing prevalence of the software pattern known as Asynchronous JavaScript and XML (AJAX) makes the situation even more challenging since a DOM can change at any time in response to user events such as clicking on a button or selecting an item from a pull down list. If we are to continue to make Web content accessible we need to develop a range of software techniques that can cope with this sort of dynamicity [14] [15].

6 Conclusions

Enhancing Web accessibility through on-the-fly content transformation remains challenging. In part this is due to the increasing richness of techniques employed by content designers. In addition, the use of AJAX and client-side, script-based content generation, while having the desirable effect of heightened interactivity and ease of use, make DOM manipulation a complex and often multi-step process. The use of style sheet rules to control element placement further complicates even simple transformations such as font size adjustments.

Between the occasional browser bug, the arguably misguided use of fixed or relative element placement, and the sheer complexity of some commercial web sites, we are faced with enough unpredictability that our work becomes essentially empirical. We can be guided by how things should work, but are often forced to create workarounds for how things actually work. Of course, empirically testing against the enormity of the Web is not feasible. But even with these difficulties our transformations have attained a level of speed and accuracy that allows them to be used on a daily basis by many disabled people around the world.

That being said, there are a few things that would make our task easier. IE, for example, would benefit from the sort of continuous font scaling that Firefox supports. And Firefox would be easier to augment if we could "batch" changes to a style sheet so that multiple changes would cause only single re-rendering. Both browsers could also provide new APIs to allow us to obtain the sorts of fine grained location-to-structure mappings we now have to construct in a laborious and sometimes error prone way. For example, all browsers know what element lies beneath the current pointer. It would be quite useful to have an API that could directly return the text offset corresponding to the pointer location to better support tasks like reading text aloud. More generally, APIs that caused the browser's renderer to make the sorts of

transformations we have found useful for people with visual disabilities would be highly desirable.

Finally, a richer set of events could be signaled (or at least the current set of events could be more reliably signaled) to allow browser extensions to know when it is safe to begin manipulating the DOM. While our software works reasonably well with the current set of events and the current mix of Web content, the increased interactivity associated with "Web 2.0" will likely require additions to the event model if accessibility transformations such as ours are to continue to work well.

References

1. Web Accessibility Initiative (Retrieved November 19, 2006) from http://www.w3.org/WAI/
2. Web Content Accessibility Guidelines Working Group (WCAG WG), (Retrieved February 15, 2006) from http://www.w3.org/WAI/GL/
3. Gunderson, J.: W3C user agent accessibility guidelines 1.0 for graphical Web browsers. Universal Access in the Information Society 3, 38–47 (2004)
4. Jacobs, I., Gunderson, J., Hansen, E. (eds.): W3C user agent accessibility guidelines 1.0. (Retrieved November 19, 2006) (2002) from http://www.w3.org/TR/UAAG10
5. Hanson, V.L., Richards, J.T.: A Web Accessibility Service: An Update and Findings. In: Proceedings of the Sixth International ACM Conference on Assistive Technologies, ASSETS 2004, pp. 169–176. ACM, New York (2004)
6. Hanson, V.L., Richards, J.T.: Achieving a usable World Wide Web. Behaviour and Information Technology 24(3), 231–246 (2005)
7. Hanson, V.L., Snow-Weaver, A., Trewin, S.: Software personalization to meet the needs of older adults. Gerontechnology 5(3), 160–169 (2006)
8. Hanson, V.L., Brezin, J., Crayne, S., Keates, S., Kjeldsen, R., Richards, J.T., Swart, C., Trewin, S.: Improving Web accessibility through an enhanced open-source browser. IBM Systems Journal 44(3), 573–588 (2005)
9. XPCOM. (Retrieved November 19, 2006) from http://www.mozilla.org/projects/xpcom/
10. Trewin, S.: Automating accessibility: the dynamic keyboard. In: Proceedings of the 6th international ACM SIGACCESS Conference on Computers and Accessibility (Atlanta, GA, USA, October 18 - 20, 2004), pp. 71–78. ACM Press, New York (2004)
11. Trewin, S., Keates, S., Moffatt, K.: Developing steady clicks: a method of cursor assistance for people with motor impairments. In: Proceedings of the 8th international ACM SIGACCESS Conference on Computers and Accessibility (Portland, Oregon, USA, October 23 - 25, 2006), pp. 26–33. ACM Press, New York (2006)
12. W3C HTML 4.01 Specification (Retrieved November 19, 2006) from http://www.w3.org/TR/html4/struct/dirlang.html
13. W3C Document Object Model (DOM) Level 1 Specification (2nd edn.) (Retrieved November 19, 2006) from http://www.w3.org/TR/2000/WD-DOM-Level-1 - 20000929/ Overview.html
14. Gibson, G.: AJAX accessibility overview (Retrieved November 19, 2006)(2006) from http://www-306.ibm.com/able/resources/ajaxaccessibility.html
15. Gibson, B., Schwerdtfeger, R.: DHTML accessibility: solving the JavaScript accessibility problem. In: Proceedings of the 7th international ACM SIGACCESS Conference on Computers and Accessibility (Baltimore, MD, USA, October 09 - 12, 2005). Assets '05, pp. 202–203. ACM Press, New York (2005)