An Overview of DHTML Accessibility

The Problem

Dynamic client-side user interfaces open up a whole new can of worms for Web accessibility. The ugly reality is that the vast majority of DHTML is inaccessible to users who can't use the mouse or who require an assistive technology such as a screen reader. Worse yet, the standards and techniques for DHTML accessibility are still being worked out, and browser support is significantly in flux. Fluid's approach has been to get involved early and help influence the standards, but life on the bleeding edge can be complex and buggy.

Most JavaScript toolkits offer a library of client-side widgets that mimic the behaviour of familiar desktop interfaces. Sliders, menu bars, file list views, and more can be built with a combination of JavaScript, CSS, and HTML. But the current HTML spec is limited to simple tags such as forms, buttons, lists, and the like. For more sophisticated interfaces, generic tags such as `<span>` and `<div>` rarely provide sufficient semantics to make the widget comprehensible to assistive technologies. Most hand-rolled DHTML widgets don't work consistently like their desktop counterparts, and keyboard navigation is often ignored completely or limited to tedious Tab-key based modes.

Screen readers and other ATs were built with the assumption that the Web consisted largely of static documents. With the advent of AJAX-updated Web applications, new techniques need to be adopted to ensure that assistive technology users are informed of the dynamic changes that can occur within a single page. It's complicated and often out of the reach of regular Web developers. That's where Fluid and the ATRC's associated projects come in.

Keyboard Navigation

Rich client-side user interfaces have the potential to significantly improve user experience, particularly in regards to keyboard navigation. In traditional Web applications, the Tab key can be a pretty tedious way to navigate around the page. One of the most interesting problems we encountered in Sakai, uPortal, and other applications while doing UX Walkthroughs was how time-consuming it is to navigate deeply within a page. The Tab and Shift-Tab keys provide only a one-dimensional way to move forward and backward through every link and form element on the page. On fairly dense pages, such as portals and dashboards, you often have to press the Tab key dozens of times to get to the section you need.

With JavaScript-driven pages, the interface can more readily be grouped into discrete chunks of functionality. Significant sections of the page – often referred to as portlets or gadgets – can be accessed using the arrow keys instead of the Tab key. Since the arrows provide two-dimensional control, this technique provides a speedy and effective way to get to your desired section of the page. The user can then tab into the section and navigate conventionally from there. This is the kind of navigation scheme we've built into the Layout Customizer, and early test results suggest it's a significant improvement.

Many other types of controls will require support for navigation with the arrow keys. File list views, progress bars, sliders, trees, and other controls should all be navigable in this way, particularly if they have ARIA roles attached (and they should). We've been participating in a W3C effort to create a keyboard accessibility style guide intended for widget developers. While I don't agree with all of these recommendations, it's a really good starting point for learning how to support familiar keyboard interactions.

The specific conventions for how the Tab, Arrow, Enter, and Spacebar keys should behave is described in Fluid's DHTML Developer's Checklist.

ARIA

For any custom controls such as the ones mentioned above, the only way to make these accessible is by providing ARIA roles and states. ARIA, which stands for Accessible Rich Internet Applications, is a draft specification from the W3C that defines special attributes to convey additional information about user interface controls. Designed to fill the gap between standard HTML tags and more varied types of controls found in rich Internet applications, ARIA provides roles and states that describe the behaviour of most familiar UI widgets.

ARIA roles and states are designed to be interpreted automatically by the browser and translated to the operating system's native accessibility APIs. Assistive technologies are thus able to recognize and interact with custom DHTML controls in exactly the same way that they do with their desktop counterparts. Assuming widget developers take the time to make their widgets follow convention, we can provide a more consistent user experience than was possible in the previous generation of Web applications.

At the moment, Firefox is the only browser that supports ARIA, but Microsoft and Opera are implementing ARIA in their next versions. There has been a bit of flux in the syntax of ARIA attributes, so I'd strongly recommend using a library such as jQuery to hide away the inconsistencies. I expect future versions of jQuery to ship with ARIA support built right in.

Fluid, Toolkit Accessibility, and the Open Web

One way to avoid some of the complexity of adding keyboard handlers and ARIA semantics is to use a DHTML toolkit. Two choices immediately come to mind: Dijit, the widget library for the Dojo toolkit, and jQuery.

We've been involved in the Dojo community for over a year now, assisting with their efforts to make many of the Dijit widgets more accessible. This work includes keyboard navigation, ARIA semantics, and high-contrast styling. The whole team has done a first-rate job on Dijit accessibility, and continue to make excellent progress. When we started the Fluid Project, we genuinely hoped to leverage this work ourselves, but ultimately realized that Dojo's approach just wasn't aligned with our needs.

With this in mind, we think that a healthy, open Web should include more than one accessible DHTML toolkit. The Mozilla Foundation agrees with us, and have kindly offered us some funding to help mentor the jQuery community on accessibility issues. We'll help ensure that the jQuery widget set is accessible, and that there are simple tools available to make third-party jQuery plugins accessible.
Fluid is contributing a lot of the code required by DHTML developers to support accessibility in their own widgets and plugins. If you're building DHTML interfaces, our plugins will make your job significantly easier by hiding away many of the complexities of accessibility support. Other members of the jQuery community are getting involved, and we expect a rich set of accessibility and progressive enhancement tools to emerge over the coming months. Many of these accessibility plugins are ready to use now, and we'd love your feedback and suggestions for how we can improve them in the future.

Dynamic Page Updates and Live Regions

The first thing you should know is that dynamic page updates, at the current time, are often met with complete silence from screen readers. Most assistive technologies assume that the Web consists of static, document-based pages. They simply don't expect chunks of content to be updated on the fly without triggering a full browser refresh. It is only recently, with the emergence of ARIA, that screen readers are becoming aware of dynamic updates.

Even assuming the assistive technology knows about the potential for live page updates, there are still some interesting questions to be answered, many of which are context-dependent. Which controls on the page can trigger a dynamic update? What types of DOM manipulations are relevant to the user? Is it appropriate to interrupt a user mid-stream to provide them with information about the update? The ARIA Live Regions attributes are designed to provide developers with the markup required to communicate their intentions to the assistive technology. ARIA includes features such as politeness levels, which specify when it is okay to interrupt a user with information about a live update, and for which types of data. The Mozilla Developer Center has a great overview of live regions if you're interested in learning more.

Unfortunately support for live regions is still in its infancy. There are, however, some workarounds to consider while we wait for comprehensive live regions support. One of the techniques you can use is to strategically move keyboard focus to the information that has changed. So for example, if you ask the user to fill out a form that will be automatically submitted and validated by AJAX, you can programmatically throw focus onto the validation status message. This will require dynamic modifications to an element's tabindex, and isn't appropriate for all situations, but it can be helpful in easing the confusion of live updates. Another technique is build your user interfaces using simple, semantic HTML delivered from the server and enhance it incrementally with rich interactivity. That way, you can still support the more traditional page-refresh model in suboptimal conditions, such as with browsers or ATs that don't fully support DHTML.

Should live updates be batched or added to the DOM individually? This is a classic "it depends" question. Under many circumstances, batch updates to a page can improve usability and accessibility because it imposes less interruptions for the user. In other cases, smaller, more frequent page updates may be more effective for time-sensitive or high-priority information. The aria-atomic property lets you specify if updates should be read as a whole or in more frequent chunks.

More Information

- DHTML Developer Checklist: A checklist of techniques for modern, accessible JavaScript programming