Toward Making Opaque Web Content More Accessible: Accessibility From Adobe Flash to Canvas-Rendered Apps

Thomas Steiner
tomac@google.com
Google Germany GmbH
Hamburg, Germany

ABSTRACT

Adobe Flash, once a ubiquitous multimedia platform, played a pivotal role in shaping the digital landscape for nearly two decades. Its capabilities ranged from animated banners and immersive websites to complex online applications and games. Flash content was embedded on websites with the `embed` or the `object` element. To the browser, the embedded content is opaque by default, which means Flash content can't be used for the accessibility tree that the browser creates based on the DOM tree, which is used by platform-specific accessibility APIs to provide a representation that can be understood by assistive technologies, such as screen readers. With Flash losing out on popularity, HTML 5 introduced the `canvas` element, which for the first time allowed developers to draw graphics and animations with either the canvas scripting API or the WebGL API directly natively in the browser. Similar to Flash, such canvas-rendered content is opaque by default and unusable for the accessibility tree. Lastly, the implementation of the WebAssembly Garbage Collection (WasmGC) standard in browsers allowed developers to port applications, written in non-Web programming languages like Kotlin for non-Web platforms like Android, to the Web by compiling them to WasmGC and rendering the entire app into a `canvas`. In the most extreme of cases, this means that the entire HTML code of an application can consist of a sole `<canvas>` tag, which evidently is opaque to the browser and impossible to leverage for the accessibility tree. Without judging their quality, this paper focuses on documenting approaches then and now for making such opaque Web content more accessible to users of assistive technologies.

CCS CONCEPTS

;

KEYWORDS

Adobe Flash, Canvas, WebAssembly, Wasm, WasmGC, Accessibility

WWW '24 Companion, May 13–May 17, 2024, Singapore

ACM ISBN 0123456789.
https://doi.org/123456789

Thomas Steiner. 2024. Toward Making Opaque Web Content More Accessible: Accessibility From Adobe Flash to Canvas-Rendered Apps. In *Companion Proceedings of the ACM Web Conference 2024 (WWW '24 Companion), May 13–May 17, 2024, Singapore.* ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION AND BACKGROUND

This section provides some background and history on the covered technologies, starting with Adobe Flash and then over to the `canvas` element and WebAssembly Garbage Collection.

1.1 Adobe Flash

Adobe Flash, once a ubiquitous multimedia platform, played a pivotal role in shaping the digital landscape for nearly two decades. Developed by Macromedia and later acquired by Adobe in 2005, Flash revolutionized the way interactive content was delivered on the Internet. Its capabilities ranged from animated banners and immersive websites to complex online applications and games. Flash's popularity soared during the late 1990s and early 2000s, establishing itself as the go-to technology for Web-based multimedia content. Steve Jobs' open letter Thoughts on Flash from 2010 in which he criticizes Adobe Systems' Flash platform and outlines reasons why the technology would not be allowed on Apple's iOS hardware products is widely regarded as the nail in the coffin of Flash. Browsers don't support Flash anymore, albeit a lot of Flash content can still be run in browsers through WebAssembly players like Ruffle.

1.2 The `canvas` element

The `canvas` element is part of HTML 5 and allows for dynamic, scriptable rendering of 2D shapes and bitmap images. It is a low level, procedural model that updates a bitmap. While the HTML 5 canvas offers its own 2D drawing API, it also supports the WebGL API to allow 3D rendering. The spec said: *"When authors use the canvas element, they should also provide content that, when presented to the user, conveys essentially the same function or purpose as the bitmap canvas. This content may be placed as content of the canvas element. The contents*

[1] Adobe Flash: http://adobe.com/flash
[2] Steve Jobs, Thoughts on Flash: https://web.archive.org/web/20100501010616/http://www.apple.com/hotnews/thoughts-on-flash/
[3] Ruffle Player: https://ruffle.rs/demo/
[4] COHG 5 `canvas`: https://www.w3.org/TR/2008/WD-html5-20080122/#the-canvas

of the canvas element, if any, are the element's fallback content". The `canvas` element is supported in browsers since 2008 and highly popular. It appears on ¿40% of pageloads in Chrome and is used for games, charts, graphics, text editing, and more 2D and 3D use cases, typically (but not necessarily) surrounded by regular HTML content. Content like games can be entirely canvas-rendered.

1.3 Canvas-rendered apps with Wasmgc

WebAssembly (Wasm) has emerged as a widely adopted technology for executing high-performance code on the Web. One of the critical challenges in its runtime environment is memory management, particularly garbage collection. Wasm's memory model is designed to be efficient and low-level, providing explicit control over memory resources. However, with the advent of Garbage Collection (GC) in WebAssembly, developers can now leverage automatic memory management to alleviate manual memory handling complexities. This means that it has now become feasible to compile managed languages like Kotlin to Wasm, without having to also compile the language's own garbage collector. First browsers started to support WasmGC in 2023. Early commercial products like JetBrains Compose Multiplatform, a declarative framework for sharing UIs across multiple platforms that is based on Kotlin and Jetpack Compose, have started to use WasmGC as a means to let developers compile their Android applications to the Web and other platforms by rendering the entire application onto a `canvas` element.

2 TECHNIQUES FOR MAKING OPAQUE CONTENT MORE ACCESSIBLE

This section provides an overview of various techniques that have been implemented or proposed over the years for making opaque Web content created with Adobe Flash or drawn onto a `canvas` element more accessible. Note that this is not a qualitative study of the success of these attempts, but barely a documentation thereof.

2.1 Flash Techniques for wcag 2.0

Adobe Flash player was a cross-platform browser plug-in. Authors creating content for display by the Flash Player may have chosen to do so for a variety of factors, including video support, authoring preference, vector-based graphics capabilities, or to take advantage of available components. The motivation of the author notwithstanding, it was deemed equally important to ensure that content playing in the Flash player met the accessibility criteria in WCAG 2.0 as it was and is for other Web content. The Flash Techniques for WCAG 2.0 lists 37 techniques labeled FLASH followed by a number that Flash authors should apply. See Listing 1 for an example of FLASH2. Most of these techniques have corresponding best practices in today's accessibility recommendations for regular HTML content. Some representative examples include:

> *"Setting the description property for a non-text object in Flash"* (FLASH2)
> *"Marking objects in Flash so that they can be ignored by AT"* (FLASH3)
> *"Using the tabIndex property to specify a logical reading order in Flash"* (FLASH15)
> *"Reskinning Flash components to provide highly visible focus indication"* (FLASH20)
> *"Labeling a form control by setting its accessible name"* (FLASH25)

An interesting artifact is FLASH34: *"Using screen reader detection to turn off sounds that play automatically"*. The intent of this technique was to prevent sounds from playing when the Flash movie loads. This was seen as useful for those who utilize assistive technologies (such as screen readers, screen magnifiers, switch mechanisms, etc.) and those who may not (such as those with cognitive, learning, and language disabilities). By default, the sound would be played automatically. When a screen reader such as JAWS was detected however, the sound would have to be started manually. To perform screen reader detection, Flash provided the `'ash.accessibility.Accessibility.active` property. If this property was `true`, it meant that the Flash player had detected running assistive technology and the Flash developer could choose to run different functionality. Not every screen reader was detected using this mechanism. In general, the property would be set to `true` when any Microsoft Active Accessibility (MSAA) client was running. MSAA is an API for user interface accessibility that was introduced as a platform add-on to Microsoft Windows 95 in 1997.

```
// Setting a textual description for a chart.
chart.accessibilityProperties = new AccessibilityProperties();
chart.accessibilityProperties.name = "October Sales Chart";
chart.accessibilityProperties.description =
 "Bar Chart showing sales for October.\
 There are 6 salespersons.Maria is highest with 349 units.Fran
 with 301.Then comes Juan with 256, Sue with 250, Li with 200
 with 195.The primary use of the chart is to show leaders, so
 description is in sales order.";
```

2.2 Canvas alternative content

The HTML 5 specification mandates: *"Authors should not use the canvas element in a document when a more suitable element is available. For example, it is inappropriate to use a canvas element to render a page heading: if the desired presentation of the heading is graphically intense, it should be marked up using appropriate elements (typically h1) and then styled using CSS and*

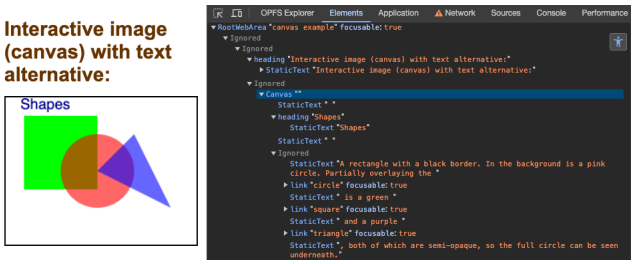
[5] Chrome Platform Status page load statistics for `canvas`: https://chromestatus.com/metrics/feature/timeline/popularity/1503
[6] B Proposal for WebAssembly: https://github.com/WebAssembly/gc
[7] JetBrains Compose Multiplatform: https://www.jetbrains.com/lp/compose-multiplatform/
[8] Web Content Accessibility Guidelines (R ' B) 2.0: https://www.w3.org/TR/WCAG20/
[9] Flash Techniques for R ' B 2.0: https://www.w3.org/WAI/GL/2010/WD-WCAG20-TECHS-20100617/flash

supporting technologies such as XBL*"*. Early on (last edit 2011), in a wiki document named Canvas Accessibility Use Cases, challenges like hit testing, magnification, explorability, and dynamic focus were identified. Further, a wiki entry titled AddedElementCanvas tracked canvas element accessibility issues and questions. It collected arguments for why this element should have accessibility provisions such as a *"native accessible <canvas> would provide a person with disabilities equal opportunity"* as well as counterarguments for why this element should not have accessibility provisions such as an *"accessibility solution for canvas is not needed, it [is] a usage problem solvable through education and evangelism"*. Several solutions were proposed:

> Specifying an accessibility API for canvas itself expanding on the DOM concept with `aria` and support for platform accessibility APIs.
> Fallbacks based on terse or long text fallbacks, elements linked to the canvas, and others.
> Adding a warning and advice to the spec for reasons not to use canvas and other suitable alternatives.
> Using meta data like `aria-role="application"` or RDFa.

The HTML Living Standard now contains a section on canvas best practices. For example, it actively discourages authors from implementing text editing controls using the `canvas` element and to instead use the `input` element, the `textarea` element, or the `contenteditable` attribute. As another solution, Listing 2 shows an interactive `canvas` with alternative textual content as fallback. Figure 1 depicts the generated accessibility tree based on the code.

```
<canvas width="260" height="200">
  <h2>Shapes</h2>
  <p>
    A rectangle with a black border. In the background is a pink circle.
    Partially overlaying the
    <a
      href="http://en.wikipedia.org/wiki/Circle"
      onfocus="drawCircle();"
      onblur="drawPicture();"
      >circle</a
    > Partially overlaying the circle is a green
    <a
      href="http://en.wikipedia.org/wiki/Square"
      onfocus="drawSquare();"
      onblur="drawPicture();"
      >square</a
    >
    and a purple
    <a
      href="http://en.wikipedia.org/wiki/Triangle"
      onfocus="drawTriangle();"
      onblur="drawPicture();"
      >triangle</a
    >, both of which are semiopaque, so the full circle can be seen
    underneath.
  </p>
</canvas>
```



2.3 The Accessibility Object Model

Assistive technology such as screen readers use the browser's accessibility API to interact with web content. The underlying model of this API is the accessibility tree: a tree of accessibility objects that assistive technology can query for attributes and properties and perform actions on. Native HTML elements are implicitly mapped to accessibility APIs. For example, an `img` element will automatically be mapped to an accessibility node with a role of `"image"` and a label based on the `alt` attribute (if present). Alternatively, ARIA allows developers to annotate elements with attributes to override the default role and semantic properties of an element, for example, to make a `div` act like a `button`. Web developers shape and manipulate the accessibility tree primarily through DOM properties such as ARIA attributes for HTML.

The Accessibility Object Model (AOM) project aims to improve certain aspects of the user and developer experiences concerning the interaction between Web pages and assistive technology. In particular, this project is concerned with improving the developer experience around, among others, expressing semantics for visual user interfaces which are not composed of elements, such as canvas-based user interfaces. At the time of this writing, the AOM is still in an early draft stage. The original idea was to use virtual nodes as a solution and to create nodes specifically for accessibility. See Listing 3 for an example.

```
// Conveying the semantics of a canvasdrawn table
canvas.attachAccessibleRoot();
const table = canvas.accessibleRoot.append(new AccessibleNo
```

[10] SVG (SHG Binding Language) is an SHG-based markup language for altering the behavior of SPG widgets devised at Netscape in the late 1990s as an extension of SPG.
[11] Canvas Accessibility Use Cases: https://www.w3.org/WAI/PF/HTML/wiki/Canvas_Accessibility_Use_Cases
[12] AddedElementCanvas: https://www.w3.org/html/wg/wiki/AddedElementCanvas
[13] Canvas best practices: https://html.spec.whatwg.org/multipage/canvas.html#best-practices
[14] Doing so is highly discouraged and developers should use a `button` directly.
[15] Accessible Rich Internet Applications (WAI-ARIA) 1.3: https://w3c.github.io/aria/
[16] Accessibility Object Model: https://wicg.github.io/aom/spec/

```
table.role = table;
table.colCount = 10;
table.rowcount = 100;
const headerRow = table.append(new AccessibleNode());
headerRow.role = row;
// etc.
```

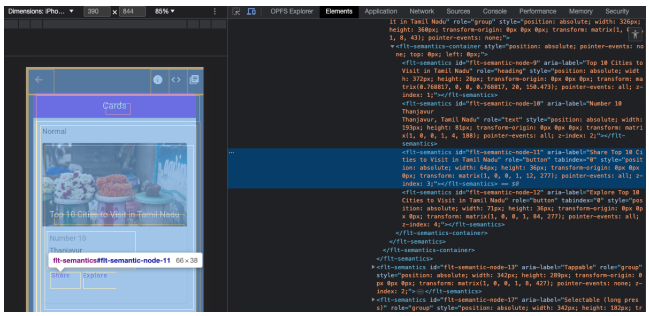
AccessibleNode

One of the biggest concerns around this proposal was that any events fired on virtual nodes would be an immediate indication that the user must be using assistive technology, which is private information that the user may not want to reveal. Adding a permission dialog might help if virtual nodes were only truly needed on a small number of specialized websites, but that would preclude their use in any widget library. To avoid privacy concerns, the currently feasibly path forward for custom-drawn UI is to promote the use of true DOM elements with ARIA, described in the following.

2.4 Hidden dom tree for canvas-rendered apps

The DOM tree of fully canvas-rendered apps can, in the extreme case, consist of nothing more than a `canvas` element. This means the accessibility tree that browsers like Chrome try to create based on the DOM tree, can't be populated. An approach for dealing with this situation is to recreate what's currently drawn onto the canvas in a hidden DOM structure that purely serves for the browser to create the accessibility tree. An industry framework that fully renders to a `canvas` to guarantee pixel perfection and platform consistency is Flutter. The framework has a large number of default widgets that generate an accessibility tree automatically. For custom widgets, Flutter's `Semantics` class lets developers describe the meaning of their widgets, helping assistive technologies make sense of the widget content. For performance reasons, at the time of this writing, Flutter's accessibility is opt-in by default.

The core idea in Flutter is to create an accessible DOM structure that reflects what's right now displayed on the canvas. See Figure 2 for a concrete example. This structure consists of an `'t-semantics-host` parent custom element that has `'t-semantics` and `'t-semantics-container` child elements that in turn can be nested. What in the accessibility tree is represented as, for example, a button, is not actually a `button` element that would be created in this DOM structure. It's an `'t-semantics` element with an `aria-label` attribute with the button's text as its value and a `role` attribute with the value of `button` that is made focusable by giving it a `tabindex` of `0`. This `'t-semantics` element is absolutely positioned to appear exactly at the position where the corresponding button is painted on the canvas. What this means is that whenever the user scrolls, the positions need to be adjusted, which is CPU-intense. For text editing, Flutter has an `'t-text-editing-host` element that has either an `input` or a `textarea` as its child that it places pixel-perfectly onto the corresponding canvas area. This means browser conveniences like autofill work as expected. This feature is always enabled, independent of whether accessibility is enabled or not.



2.5 Accessibility with AccessKit

How could this approach be generalized for other frameworks that render to a canvas? Abstracting the hidden DOM structure solution is something the developers of an open-source project called AccessKit are working on: *"The heart of AccessKit is a data schema that defines all the data required to render an accessible UI for screen readers and other assistive technologies. The schema represents a tree structure, in which each node or branch is either a single UI element, or an element cluster such as a window, pain, or document. Each node has an integer ID, a role (e.g., button, edit, etc.) and a variety of optional attributes. The schema also defines actions that can be requested by assistive technologies, such as moving the keyboard focus, invoking a button, or selecting text"*. Platform adapters then implement the accessibility APIs across various platforms. This cuts down significantly on manual legwork when designing apps, as individual accessibility implementations will not need to be manually created for each platform. For now, there are adapters for Windows, macOS, and Unix; with plans for adapters for Android, iOS/tvOS, and web (by creating a hidden HTML DOM).

2.6 Will ai save us?

Rather than going from the original framework code bottom up to the to-be-generated accessibility tree, one could also imagine an AI-based approach, where the AI *ad hoc* in the browser would try to top down understand what is currently painted onto the canvas and translate that into an accessibility tree. While definitely challenging, it doesn't look completely impossible based on enough training data. The AI model would always be constrained to analyzing what it can currently "see" on the canvas, though, so anything outside of the viewport of the canvas or anything outside of the viewport of a widget, like hidden items in a long scrollable listbox, would be completely hidden from the approach, so

[17] Accessibility tree in the Chrome browser: https://developer.chrome.com/blog/full-accessibility-tree
[18] Flutter: https://flutter.dev/
[19] AccessKit: https://accesskit.dev/how-it-works/

this method seems at least partly infeasible and impractical for some potential use cases. Additional challenges could come from the processing power required for such a solution to work in a timely enough manner. Low-powered devices might already struggle with the tasks of running the canvas-rendered application in the first place, and not have enough spare CPU cycles to run the AI on top.

3 CONCLUSIONS

Opaque content on the Web existed from the early days of Flash, to the present day with partial content like charts that get rendered opaquely to a canvas, all the way to entire apps that consist of nothing but a sole `canvas` element (and loads of JavaScript and WebAssembly). Interesting questions like on the desirability of screen reader detection keep being raised, starting from Flash's `flash.accessibility.Accessibility.active` property to Flutter's opt-in button to heuristics that some apps use based on keyboard navigation patterns. Accessibility can't be an afterthought, and protecting the privacy of those dependent on assistive technologies is deeply rooted in the Web Platform Design Principles. On the one hand, browser API designers need to make sure that their API doesn't provide a way for authors to detect that a user is using assistive technology without the user's consent. Early versions of the well-intentioned AOMdid not respect this principle and were hence removed. On the other hand, the web platform must be accessible to people with disabilities, which, in the case of canvas-rendered apps or canvas-rendered content can be challenging to impossible without knowing assistive technologies are in use. Concluding, ensuring the accessibility of opaque Web content was and remains a strong challenge worth solving. Often, not using opaque Web content in the first place may be the simplest solution.

[20] Charts.js library: https://www.chartjs.org/

[21] Web Platform Design Principles: https://www.w3.org/TR/design-principles/#do-not-expose-use-of-assistive-tech