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 declining in 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 and natively in the browser. As with Flash, such canvas-rendered content is opaque by default and unusable for the accessibility tree. Ultimately, 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 is opaque to the browser and impossible to read 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

• **Information systems → Browsers**.

KEYWORDS

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

ACM Reference Format:
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 13–17, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3589335.3651999

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

ACM ISBN 979-8-4007-0172-6/24/05
https://doi.org/10.1145/3589335.3651999

1 INTRODUCTION AND BACKGROUND

This section provides some background and history of the covered technologies, starting with Adobe Flash and then the `canvas` element and WebAssembly Garbage Collection. (Other opaque Web content exists, namely images, and audio and video, but those have alternative text, and transcripts or captions for accessibility.)

1.1 Adobe Flash

Adobe Flash,[1] 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*[2] from 2010 criticizes the technology and outlines why it would not be allowed on Apple's products. The letter is widely regarded as the nail in the coffin of Flash as it meant Flash would never make it onto the popular iPhone. Browsers don't support Flash anymore, albeit a lot of Flash content can still be run in browsers through WebAssembly players like Ruffle.[3]

1.2 The `canvas` element

The `canvas` element[4] 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 says: *"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 of the canvas element, if any, are the element's fallback content"*. The `canvas` element has been supported in browsers since 2008 and highly popular. It appears on >40% of pageloads in Chrome[5] 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] 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] HTML 5 canvas: https://www.w3.org/TR/2008/WD-html5-20080122/#the-canvas
[5] Chrome Platform Status page load statistics for canvas: https://chromestatus.com/metrics/feature/timeline/popularity/1503

1.3 Canvas-rendered apps with WasmGC

WebAssembly (Wasm) has emerged as a widely adopted technology for executing high-performance code on the Web. It is optimized for runtime 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 its integration of Garbage Collection (GC),[6] developers can now leverage automatic memory management to alleviate manual memory handling complexities. This means it is now feasible to compile managed languages like Kotlin to Wasm, without having to also compile the language's own garbage collector. Browsers started to support WasmGC in 2023. Early commercial products like JetBrains Compose Multiplatform,[7] a declarative framework for sharing UIs across multiple platforms that is based on Kotlin and Jetpack Compose, include WasmGC 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 merely 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 Flash Player may have chosen to do so for a variety of reasons, 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 Flash Player met the accessibility criteria in WGAC 2.0.[8] The Flash Techniques for WCAG 2.0[9] lists 37 techniques (labeled FLASH1, FLASH2, *etc.*) that authors should use. 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 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)

FLASH34 is particularly interesting: *"Using screen reader detection to turn off sounds that play automatically"*. The intent of this technique is to prevent sounds from playing when the Flash movie loads. This was considered useful both for those who utilize assistive technologies (such as screen readers, screen magnifiers, switch mechanisms, etc.) and some who may not (such as those with cognitive, learning, and language disabilities). By default, the sound would play automatically. However, when a screen reader such as JAWS was detected however, the sound would have to be started manually. To perform screen reader detection (considered a privacy no-go by today's standards), Flash provided the `flash.accessibility.Accessibility.active` property. If this property was `true`, it meant that Flash Player had detected an assistive technology and the Flash developer could choose to run different process. 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 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.Frances is next\
 with 301.Then comes Juan with 256, Sue with 250, Li with 200 and Max\
 with 195.The primary use of the chart is to show leaders, so the\
 description is in sales order.";
```

Listing 1: Setting the description property for a non-text object in Flash (FLASH2)

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 supporting technologies such as XBL".*[10] Early on (last edit 2011), in a World Wide Web Consortium (W3C) wiki document named Canvas Accessibility Use Cases,[11] challenges like hit testing, magnification, explorability, and dynamic focus were identified. Another wiki entry entitled AddedElementCanvas[12] tracked canvas element accessibility issues and questions. It listed reasons why this element should have accessibility provisions. For example, a *"native accessible* `<canvas>` *would provide a person with disabilities equal opportunity"*. It also listed counterarguments 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.

[6] GC Proposal for WebAssembly: https://github.com/WebAssembly/gc
[7] JetBrains Compose Multiplatform: https://www.jetbrains.com/lp/compose-multiplatform/
[8] Web Content Accessibility Guidelines (WCAG) 2.0: https://www.w3.org/TR/WCAG20/
[9] Flash Techniques for WCAG 2.0: https://www.w3.org/WAI/GL/2010/WD-WCAG20-TECHS-20100617/flash
[10] XBL (XML Binding Language) is an XML-based markup language for altering the behavior of XUL widgets devised at Netscape in the late 1990s as an extension of XUL.
[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

- Fallbacks based on short or long text, elements linked to the canvas, and others.
- Adding a warning to the spec for reasons not to use canvas, and other suitable alternatives.
- Using metadata like `aria-role="application"` or RDFa.

The HTML Living Standard now contains a section on canvas best practices.[13] For example, it actively discourages authors from implementing text editing controls using the `canvas` element and instead recommends using 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 semi-opaque, so the full circle can be seen
    underneath.
  </p>
</canvas>
```

Listing 2: Interactive canvas with text alternative. (Source: https://www.html5accessibility.com/tests/canvas.html)

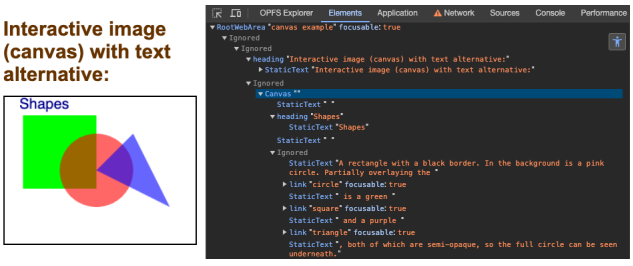

Figure 1: Inspecting the accessibility tree of an interactive canvas with text alternative.

[13]Canvas best practices: https://html.spec.whatwg.org/multipage/canvas.html#best-practices

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 to override the default role and semantic properties of an element, for example, to make a `div` act like a `button`.[14] Web developers shape and manipulate the accessibility tree primarily through DOM properties such as ARIA attributes for HTML.[15]

The Accessibility Object Model[16] (AOM) project aims to improve certain aspects of the user and developer experience concerning the interaction between Web pages and assistive technology. In particular, the project is concerned with improving the developer ability to express 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 create *virtual* nodes specifically for accessibility. See Listing 3 for an example.

```
// Conveying the semantics of a canvas-drawn table.
canvas.attachAccessibleRoot();
const table = canvas.accessibleRoot.append(new AccessibleNode());
table.role = 'table';
table.colCount = 10;
table.rowcount = 100;
const headerRow = table.append(new AccessibleNode());
headerRow.role = 'row';
// etc.
```

Listing 3: The now retracted `AccessibleNode` API of the Accessibility Object Model.

One of the biggest concerns of this proposal was that any events fired on virtual nodes would immediately indication that the user is 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 needed on a small number of specialized websites, but that would preclude their use in any widget library. To avoid privacy concerns, the currently feasible solution for custom-drawn UIs is to promote the use of true DOM elements with ARIA, as described below.

2.4 Hidden DOM tree for canvas-rendered apps

The DOM tree of fully canvas-rendered apps can, in the reductive case, consist of nothing more than a `canvas` element. This means that DOM-based accessibility trees in browsers like Chrome[17] can't be populated. One approach for dealing with this is to recreate what's currently drawn onto the canvas in a hidden DOM structure

[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/
[17]Accessibility tree in the Chrome browser: https://developer.chrome.com/blog/full-accessibility-tree

used exclusively by the browser to create the accessibility tree. Flutter[18] is an industry framework that fully renders to a canvas to guarantee pixel perfection and platform consistency. 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 currently displayed on the canvas (Figure 2). This structure consists of an `flt-semantics-host` parent custom element that has any combination of `flt-semantics` and `flt-semantics-container` child elements that can themselves be nested. For example, what is represented in the accessibility tree as a button would not actually be a `button` in the DOM structure. It's an `flt-semantics` element with an `aria-label` attribute, with the button's text as its value, and a `role` attribute with the value of `button`. It's made focusable by giving it a `tabindex` of `0`. This `flt-semantics` element is absolutely positioned to appear exactly where the corresponding button is painted on the canvas. As such, whenever the user scrolls, the positions need to be adjusted, which is CPU-intense. For text editing, Flutter has an `flt-text-editing-host` element that has either an `input` or a `textarea` as its child that it places with pixel precision onto the corresponding canvas area. Browser features such as autofill then work as expected. This feature is always enabled, independent of whether accessibility is enabled or not.

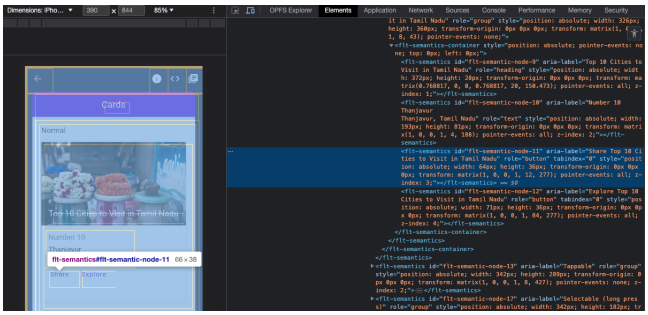

Figure 2: Flutter widget with hidden DOM structure. (Source: https://flutter-gallery-archive.web.app/#/demo/card)

2.5 Accessibility with AccessKit

How could this approach be generalized for other frameworks that render to a canvas? Abstracting the solution of hidden DOM structures is something the developers of an open-source project called AccessKit[19] 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"*. So-called platform adapters then implement the accessibility APIs across various platforms. This cuts down significantly on development time when coding apps, as individual accessibility implementations will not need to be manually created for each platform. Currently, there are adapters for Windows, macOS, and Unix, with plans for Android, iOS/tvOS, and Web (by creating a hidden HTML DOM).

[18] Flutter: https://flutter.dev/
[19] AccessKit: https://accesskit.dev/how-it-works/

3 FUTURE DIRECTIONS—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. Here the AI would try *ad hoc* to understand top-down what is currently painted onto the canvas, and translate that into an accessibility tree. Given enough training data, this could possibly work. However, the AI model would always be constrained to analyzing what it can currently "see" on the canvas. Anything outside of the canvas viewport, or anything outside of the viewport of a widget (like a long scrollable list box of hidden items), would be completely hidden from the AI. As such, this method seems problematic for some potential use cases. The processing power required to resolve in a timely manner may also be a limitation. Low-powered devices might already struggle with rendering the canvas application in the first place, and not have enough spare CPU cycles to run the AI on top.

4 CONCLUSIONS

Opaque content has long existed on the Web, since the early days of Flash. Today, partial content like charts[20] is rendered opaquely to a canvas, and some apps consist of nothing more than a sole canvas element (supported by loads of JavaScript and WebAssembly). Questions on the desirability of screen reader detection have been consistently explored, from heuristics that some apps use based on keyboard navigation patterns, to Flutter's opt-in button, to Flash's `flash.accessibility.Accessibility.active` property. 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.[21] On one hand, browser API designers need to ensure that their API requires user consent, before a developer can detect that an assistive technology is being used. Early versions of the well-intentioned AOM did 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. Ensuring the accessibility of opaque Web content has always been a challenge worth solving. Ironically, 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