IMPLEMENTAÇÃO DE RECURSOS DE ACESSIBILIDADE NO CANVAS DO HTML5

Diego Cândido Lima[1], João Bosco Ferreira Filho[2]

RESUMO

Para promover a acessibilidade na web, os desenvolvedores expõem elementos de páginas HTML5, permitindo que ferramentas de acessibilidade interpretem o conteúdo adequadamente. No entanto, componentes gráficos renderizados em elementos Canvas não são mapeados em código HTML, o que representa um desafio significativo para a análise semântica por essas ferramentas. Este artigo apresenta uma implementação que mapeia elementos gráficos dentro do Canvas para código HTML5 compreensível para ferramentas de acessibilidade, com o objetivo de aprimorar a acessibilidade e a usabilidade.

Palavras-chave: acessibilidade; acessibilidade na web; Canvas; HTML5.

ABSTRACT

To promote web accessibility, developers expose HTML5 page elements, allowing accessibility tools to interpret the content appropriately. However, graphical components rendered on Canvas elements are not mapped to HTML code, which poses a significant challenge for semantic analysis by these tools. This paper presents an implementation that maps graphical elements within the Canvas to HTML5 code understandable by accessibility tools, aiming to enhance accessibility and usability.

Keywords: accessibility; web accessibility; Canvas; HTML5.

1 Introdução

A acessibilidade na web refere-se à prática de garantir que os *websites*, aplicativos e outros recursos online possam ser utilizados e compreendidos por todas as pessoas, independente de suas habilidades físicas ou cognitivas. Isso significa que a web deve ser projetada e desenvolvida de forma a permitir o acesso igualitário a informações e funcionalidades para todos os usuários, incluindo aqueles com deficiências visuais, auditivas, motoras, cognitivas ou outras limitações. Segundo Conforto e Santarosa (2002), tornar a *web* acessível favorece não apenas pessoas com deficiência, mas também populações hipossuficientes e pessoas com poucos recursos de comunicação.

O Canvas é um elemento do HTML5 que fornece uma área de desenho definida em *pixels*, permitindo que os desenvolvedores criem gráficos, animações e outros tipos de conteúdo visual estático ou dinâmico diretamente no navegador. Ele utiliza um contexto de desenho em JavaScript para renderizar formas, texto, imagens e outras representações gráficas, oferecendo grande flexibilidade e controle sobre a manipulação de *pixels*.

Atualmente, apesar da importância da acessibilidade na web, a acessibilidade em elementos Canvas do HTML5 enfrenta desafios significativos que comprometem a experiência de usuários com deficiências. Diferentemente de outros elementos HTML, o Canvas não possui suporte nativo para tecnologias assistivas, como leitores de tela,

[1] Curso de Ciência da Computação, Universidade Federal do Ceará. Email: diegocndd4@gmail.com
[2] Doutor em Ciência da Computação, Departamento de Computação, Universidade Federal do Ceará. Email: bosco@dc.ufc.br

dificultando a interpretação de seu conteúdo para pessoas com deficiência visual. A ausência de semântica e a dificuldade em associar conteúdo alternativo significativo ao que é renderizado no canvas limitam a acessibilidade, exigindo soluções alternativas complexas e customizadas. Elementos como gráficos, animações e jogos, frequentemente desenvolvidos com Canvas, exigem a implementação manual de alternativas acessíveis, como a criação de descrições textuais dinâmicas. No entanto, essas soluções demandam conhecimento técnico mais avançado e são raramente implementadas pela complexidade, deixando muitos conteúdos baseados no Canvas inacessíveis para usuários com deficiência.

Isso ocorre porque elementos contidos no Canvas são puramente gráficos, não sendo representados na árvore DOM (Document Object Model). Em vez disso, o conteúdo do Canvas é gerado usando a API de desenho do Canvas HTML5, que fornece métodos para desenhar formas, texto e imagens no Canvas usando JavaScript. Na prática, o usuário consegue ver o desenho do Canvas, mas, ao analisar o código, verá apenas a *tag* Canvas. Por isso, o conteúdo desses elementos não é analisado semanticamente por ferramentas de acessibilidade na *web*, que são, em geral, incapazes de realizar uma análise visual e disparar eventos com precisão em elementos puramente gráficos. Dessa forma, nem todos os usuários podem utilizá-lo de maneira plena. Usuários cegos ou com baixa visão, por exemplo, não conseguem distinguir esses elementos gráficos, que também não são identificáveis ou focáveis pelas ferramentas de acessibilidade.

Como elementos Canvas não são descritíveis, quando desenvolvedores criam tais elementos, eles são motivados a criarem DOMs *fallbacks* (ou *subDOMs*), ou seja, elementos HTML5 internos ao Canvas que representam, ainda que de maneira simplória, o que é apresentado na tela (Figura 1). Contudo, o preço do retrabalho em replicar a mesma interface já desenvolvida desmotiva desenvolvedores a construírem essas subDOMs.

O World Wide Web Consortium (adiante, apenas W3C) propõe no artigo Canvas Hit Testing (W3C, 2024) a criação de dois novos métodos para a API do Canvas: *setElementPath* e *clearElementPath*, que criam e destroem, respectivamente, uma associação entre elementos gráficos do Canvas e elementos semânticos da DOM *fallback*. O objetivo deste artigo é apresentar uma implementação de uma classe com ambos os métodos sugeridos, mas que também permite a geração automática da *subDOM*.

Figura 1 – Elementos HTML dentro de fora da subDOM do Canvas.

```
<p>This is not part of Canvas subDOM</p>

<canvas>
  <p>This is part of Canvas subDOM</p>
  <button>subDOM button</button>
</canvas>
```

Fonte: Elaborada pelo autor.

2 Trabalhos relacionados

Abuaddous et al. (2016) destacam que a falta de motivação e treinamento entre as equipes de desenvolvimento de *websites* é um dos principais fatores que contribuem para a baixa conformidade com os padrões de acessibilidade. A ausência de conhecimento adequado e incentivo para a implementação de práticas de acessibilidade compromete

significativamente a qualidade e a acessibilidade dos *sites* desenvolvidos. Além disso, o estudo concluiu que seguir as diretrizes de acessibilidade pode tornar o próprio processo de avaliação de acessibilidade mais lento e mais caro. Essa situação sugere que, para garantir que a web seja acessível e, em particular, que o Canvas seja acessível, é crucial minimizar ao máximo o retrabalho e a reestruturação de *websites*.

Steiner (2024) pontua que uma das principais soluções para o problema da acessibilidade no Canvas é explicitar o conteúdo visual em um conteúdo alternativo acessível. Isso seria possível através de qualquer elemento HTML disponível na DOM. No entanto, usualmente sugere-se que este conteúdo alternativo esteja apresentado sob uma DOM *fallback* interna ao elemento Canvas. Além disso, o Accessibility Object Model (AOM) também pode fortalecer a acessibilidade nesses casos. O AOM é uma especificação em desenvolvimento pela comunidade W3C que visa aprimorar a acessibilidade das interfaces web, permitindo uma maior interação entre as tecnologias assistivas e as aplicações modernas baseadas em JavaScript.

Faulkner (2010) levantou uma das primeiras propostas para solucionar o problema, a *Issue 105*, que sugeria permitir o atributo *usemap* para elementos Canvas. O atributo *usemap* do HTML5 é utilizado em conjunto com a *tag img* para associar uma imagem a um mapa de imagem, isto é, um mapa de áreas clicáveis sobre uma imagem. Quando uma imagem é associada com um mapa de imagem, é possível especificar áreas dentro dela que são clicáveis e atribuir *URLs* ou ações a essas áreas específicas. A proposta levantada sugeria que permitir o *usemap* também ao Canvas forneceria um método para adicionar facilmente áreas interativas. No entanto, a decisão final do grupo de trabalho rejeitou a proposta, pois o *usemap* não cobre casos de uso suficientes, sendo muito limitado a exemplos simplórios de áreas poligonais, onde nem mesmo o uso do Canvas seria vantajoso, sendo mais favorável o uso de HTML e CSS (Cascading Style Sheets) puros.

Aghdam e Ravanmehr (2013) também apresentaram uma proposta para mapear os elementos do Canvas para uma DOM utilizando técnicas de CBIR (Content Based Image Retrieval) incorporadas em uma extensão para navegador Firefox. A extensão tem por objetivo substituir o Canvas por uma interface de usuário HTML equivalente, analisando os *pixels* do Canvas e, a partir de suas características visuais, gerando elementos equivalentes na DOM. No entanto, a solução apresentou certas limitações na geração desses elementos equivalentes, sendo capaz de converter interfaces mais simples, mas com falhas na identificação de elementos mais complexos, confundindo, por exemplo, desenhos circulares genéricos com *inputs* do tipo *radio*. Além disso, em jogos e animações, a extensão teria de aplicar CBIR e recriar a subDOM em todos os quadros, o que seria computacionalmente custoso, limitando a solução a telas estáticas.

Zen (2024) propôs outra solução que chegou ainda mais perto e foi apresentada no *framework* ZimJS para linguagem JavaScript, que fornece uma classe *Accessibility*, que permite adicionar alguns recursos de acessibilidade aos elementos do Canvas. O *framework* toma os elementos gráficos criados e replica em elementos na DOM, permitindo que o leitor de tela leia esses elementos. No entanto, essa abordagem, embora ofereça alguma acessibilidade aos conteúdos visuais do Canvas, não é tão eficaz quanto a estruturação em árvore na DOM e não replica os eventos entre o elemento gráfico e o elemento HTML. Essa falta de organização pode resultar em uma experiência menos coerente e menos eficaz para usuários de tecnologias assistivas, especialmente em casos mais complexos de interação ou apresentação de informações visuais.

3 Metodologia

Um *path*, no contexto do HTML5, refere-se a uma sequência de comandos de desenho que definem uma forma gráfica complexa ou um contorno. Esses comandos são utilizados para criar figuras geométricas, traçar linhas, curvas e outras formas dentro do elemento Canvas. Um *path* é, portanto, puramente gráfico e não possui associação alguma com qualquer elemento inteligível além do visual.

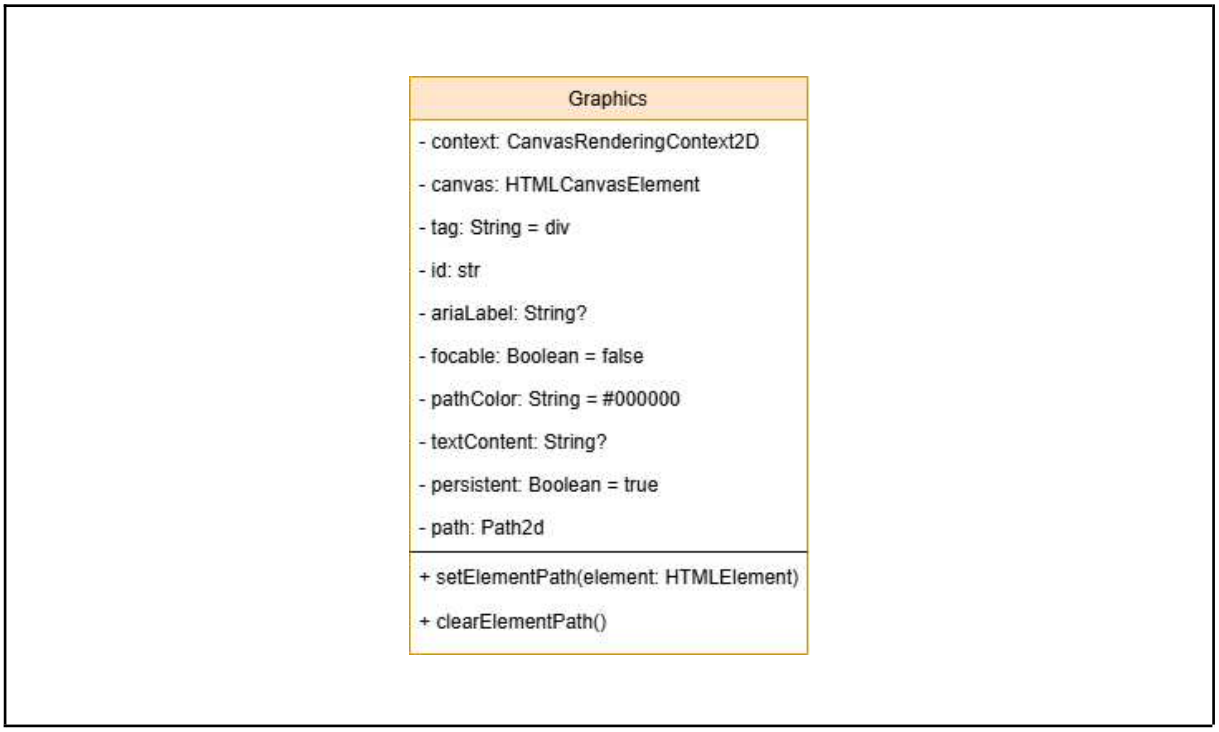
Em WHATWG (2024) há uma seção voltada para as melhores práticas de construção de um Canvas. Dentre elas, há a sugestão da criação de uma DOM *fallback* para o Canvas que o desenvolvedor programou. Além disso, deve haver uma correspondência direta entre as regiões interativas no Canvas e as áreas focáveis do *fallback*.

O W3C propõe no artigo *Canvas Hit Testing* (W3C, 2024) a criação de dois novos métodos que auxiliariam a criação de uma DOM *fallback*: *setElementPath* e *clearElementPath*, que respectivamente criam e destroem uma associação entre o *path* atual e o elemento HTML passado como argumento. No entanto, o *path* não é acessível fora da implementação interna do Canvas, que por sua vez não é acessível ao usuário desenvolvedor. Essa falta de transparência limita a manipulação eficaz dos elementos, dificultando a exploração e o aproveitamento de todas as suas capacidades, tornando o Canvas acessível apenas pelos seus pixels.

Para contornar essa limitação, criamos uma classe *Graphics*, que armazena e gerencia os *paths* de cada desenho. Nessa classe, implementamos tambem os dois metodos sugeridos pelo W3C.

Dessa forma, ao nível do código, torna-se possível armazenar os *paths* e reutilizá-los quando necessário. A Figura 2 apresenta um diagrama parcial da classe Graphics. Destacamos que a classe recebe uma referência ao contexto e ao Canvas em que será desenhado o *path*. Este, por sua vez, é inicializado dentro da classe e pode ser utilizado pelo usuário desenvolvedor ao chamar o atributo *path* da classe.

Figura 2 – Diagrama parcial da classe *Graphics*.



Fonte: Elaborada pelo autor.

A principal vantagem de armazenar o *path* no objeto da classe é permitir que um código anterior que já utilizava os métodos do Path2D possa ser facilmente adaptado a um código que utilize a classe Graphics. Na prática, as instâncias da classe ficam responsáveis por armazenar o *path* e realizar as mudanças necessárias sobre ele.

Nos próximos subtópicos, apresentamos a implementação dos métodos *setElementPath* e *clearElementPath*.

3.1 setElementPath

Para associar um *path* a um elemento HTML, utilizamos o conceito de mapeamento. Por mapeamento queremos dizer que ambos os elementos semântico e gráfico possuem os mesmos ouvintes de eventos. Um ouvinte de eventos em JavaScript é uma forma de configurar uma resposta automática a ações específicas que ocorrem em elementos da página *web*. Na prática, isso implica que disparar um evento no *path* é o mesmo que disparar um evento no elemento HTML associado e vice-versa.

A classe *Graphics* fornece duas formas para esse mapeamento: automático ou manual. No mapeamento automático, o usuário passa os comandos para o *path* ao desenhar a figura e depois chama um método *draw*, que é responsável por criar o elemento HTML5 e realizar o mapeamento. No mapeamento manual, o usuário deve criar o elemento semântico como filho do Canvas e, após isso, chamar o método *setElementPath*, passando o elemento como parâmetro.

Tal mapeamento entre o elemento e o path baseia-se, portanto, em dois princípios: representação do *path* na *subDOM* do Canvas e propagação dos eventos.

3.1.1 Representação do path na subDOM

Assistentes de acessibilidade e leitores de telas utilizam a estrutura em árvore do HTML5 para reconhecer a prioridade de leitura entre os elementos. Por ser um elemento baseado em *pixels*, o Canvas não estabelece uma relação de parentesco entre os *paths* desenhados da mesma forma que a estrutura em árvore da DOM. De fato, isso dificultaria a compreensão da estrutura semântica da subDOM dos elementos visuais. Visando sanar essa carência, a classe *Graphics* permite a criação de relação entre os elementos visuais, relação tal que é refletida na subDOM que está sendo manipulada.

Essa relação é construída a partir de uma árvore de elementos que cada objeto possui. Quando um objeto é tornado filho de outro, sua árvore de filho torna-se sub-árvore deste e, por consequência, todos os seus eventos são clonados para seus filhos.

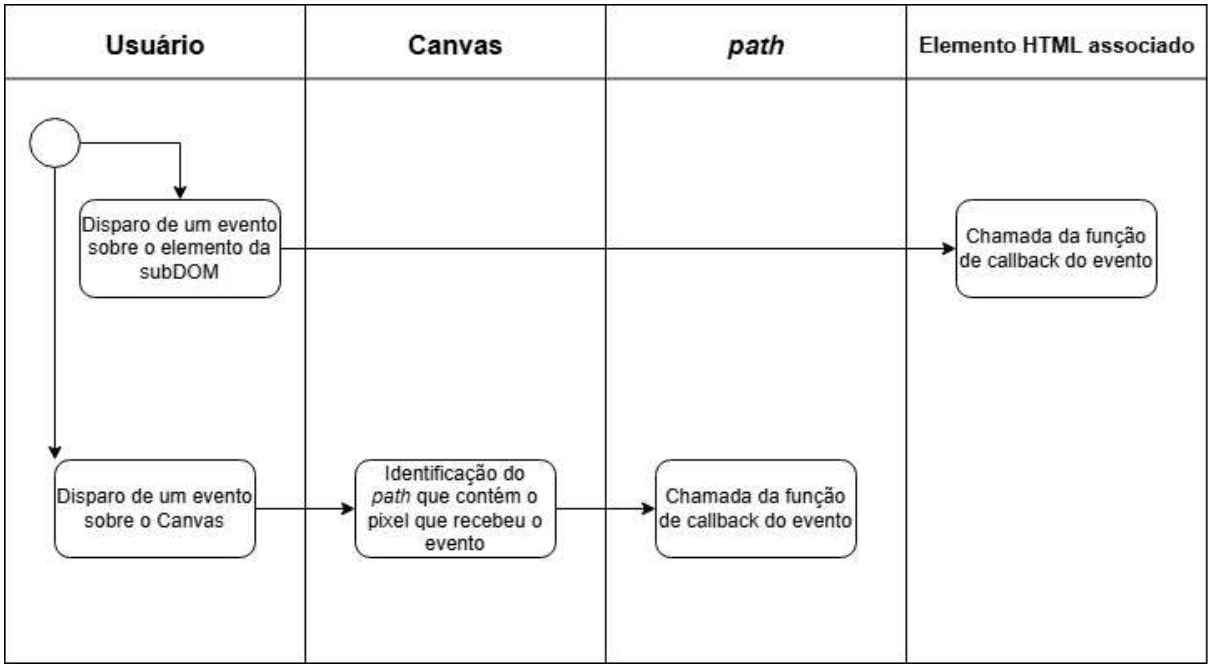
Em Fulton (2013, p. 32-33) é apresentado o problema de uma solução que crie elementos HTML associados aos *paths*: aplicações interativas que utilizem o Canvas podem fazer uso de uma série de quadros e animações que exigem a renderização contínua da tela, o que pode gerar um elevado custo de memória e desempenho de CPU para gerar esses elementos HTML associados em cada frame. Portanto, para gerar a estabilidade na *subDOM*, cada elemento HTML gerado pela classe *Graphics* possui um *drawer-id*, que identifica o elemento que já foi desenhado na tela, e um atributo booleano *persistent*, que indica se é o elemento é ou não persistente, ou seja, se é necessário preservar esse elemento nas múltiplas renderizações do Canvas. Se o elemento for persistente, no próximo *frame*, não será necessário reconstruí-lo na *subDOM*.

3.1.2 Propagação de eventos

Quando ocorre o mapeamento manual, os eventos do elemento são copiados para a área do *path* do Canvas. Quando ocorre o mapeamento automático, os eventos devem ser adicionados à instância da classe *Graphics*, sendo então copiados para o elemento HTML e para o *path*. Para copiar um evento para o elemento, apenas chamamos o método HTML5 e passamos a função de evento definida como um *callback*. Para copiar um evento para o *path*, criamos um ouvinte geral de eventos no Canvas e, sempre que esse evento é disparado, checamos se a área em que ele foi disparado corresponde a algum *path* com elemento HTML associado.

Os eventos propagados na implementação atual da classe *Graphics* são: *onclick*, que captura um clique no *path*, *mousemove*, que captura o movimento do mouse sobre o *path*, e *onfocus*, que captura o foco no *path*. A Figura 3 representa visualmente o que ocorre quando um usuário dispara um evento. Ao clicar no elemento HTML associado, a função é chamada instantaneamente. Ao clicar no Canvas, o *path* é identificado, bem como seu elemento associado, e então a mesma função é chamada.

Figura 3 – Diagrama de atividade com a propagação de um evento disparado pelo usuário.



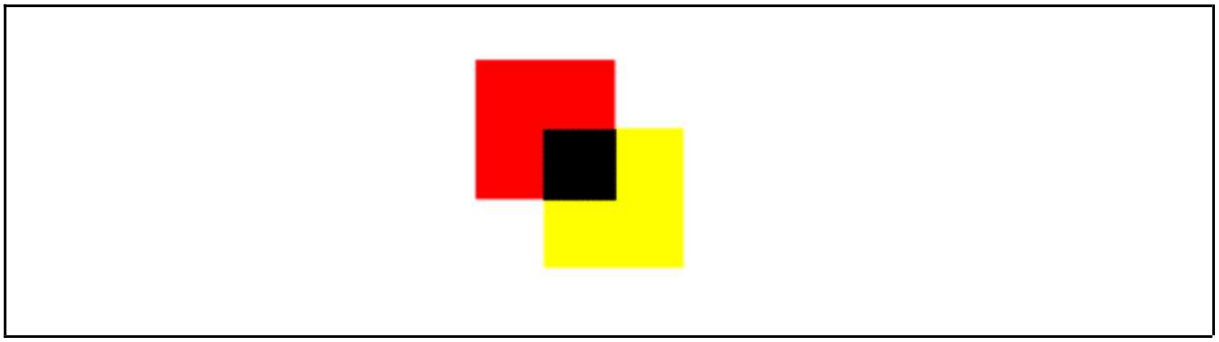
Fonte: Elaborada pelo autor.

3.1.2.1 onclick

O clique no elemento pode ocorrer de duas formas: através do mouse ou do teclado. No primeiro caso, o usuário clica sobre um *pixel* no Canvas, que recebe o evento e verifica qual elemento está associado ao *path* daquele *pixel* clicado. Identificado o elemento, chama-se sua função *callback* de clique. Isso implica que clicar no elemento gráfico e chamar o método de clique do elemento HTML disparam o mesmo evento. Caso o usuário não tenha acesso ao *mouse*, o clique pode ocorrer pelo teclado. Para tal, ele precisaria, utilizando a tecla Tab, transitar entre os elementos da tela até que a área desejada seja focada, o que será descrito adiante no tópico 3.1.2.3. Após o foco, o usuário pressiona a tecla Enter, que dispara o evento de clique.

Na Figura 4, a área preta indica uma área de sobreposição de dois *paths*. Quando uma área desse tipo é clicada, atendemos ao critério sugerido pelo W3C[1] e consideramos o último *path* que chamou o método *setElementPath* como o elemento efetivamente clicado.

Figura 4 – Sobreposição de dois *paths* representada na cor preta.



Fonte: Elaborada pelo autor.

3.1.2.2 mousemove

Segundo Gaver (1991), a mudança do cursor do mouse é importante para identificar áreas interativas. Para botões e *links*, por exemplo, a mudança do cursor para o tipo *pointer* fortalece a *affordance* do elemento gráfico.

Para identificar em que áreas o cursor será modificado, utilizamos a mesma estratégia para clique. Criamos um *event listener* em todo o Canvas, que capta o pixel sobre o qual o mouse está e identifica o elemento correspondente à sua região. Se o elemento for clicável (propriedade definida pelo usuário desenvolvedor), sua área recebe um cursor *pointer*.

3.1.2.3 onfocus

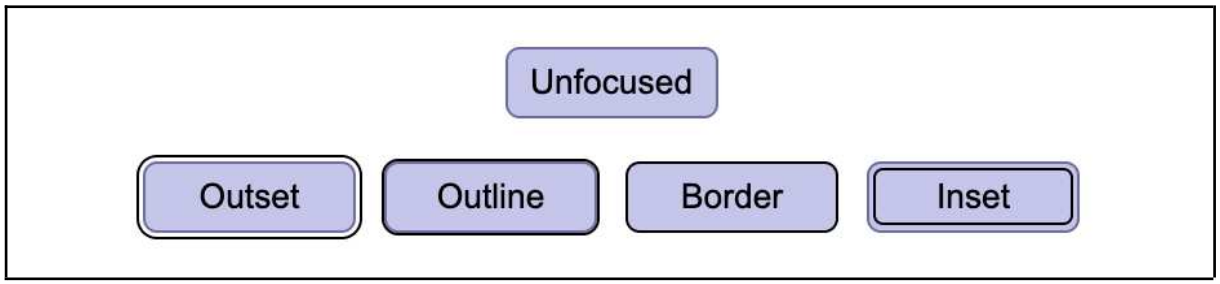
Um elemento focado é um elemento que está preferencialmente destacado dentre os demais e, além disso, recebe os eventos do teclado disparados pelo usuário. O Web Content Accessibility Guidelines (adiante, apenas WCAG) no tópico 2.4.7 da versão 2.2 sugere como critério de sucesso que elementos operáveis pelo teclado tenham um foco visível. Este critério ajuda os usuários a identificar o elemento da página *web* que receberá os eventos do teclado.

Esses focos são implementados em elementos HTML, em boa parte, de forma nativa por navegadores. A única implementação nativa desses anéis para o elemento Canvas se dá pelo método *drawFocusIfNeeded* do objeto nativo *context*. Este método é chamado sempre que um *path* é desenhado no Canvas e é responsável por desenhar um anel de foco em torno do *path* caso o elemento associado esteja focado.

Integrando o *drawFocusIfNeeded* na classe Graphics, conseguimos criar o anel de foco sempre que o evento de foco é disparado no elemento HTML. No entanto, ao chamar o evento *blur*, que desfaz o foco, redesenhamos o *path*, apagando o anel de foco desenhado internamente, mas não conseguimos apagar os resquícios externos do foco. Isso significa que para apagar esses resquícios, teríamos que chamar a função que renderiza todo o Canvas novamente. Em muitas aplicações, como jogos e animações, cada renderização pode ser vital e exigir uma série de cálculos que podem ser indesejados.

A API do Canvas não fornece métodos para especificar o tipo de anel de foco. Portanto, optou-se por construir um método próprio para o desenho do anel de foco.

Figura 5 – Tipos de anéis de foco.



Fonte: World Wide Web Consortium (2024).

Em W3C (2024), a WCAG fornece alguns tipos de contornos possíveis para a construção de um anel de foco, conforme ilustrado na Figura 5. Optamos por construir o tipo *border* com espessura de 2 *pixels* CSS. Um pixel CSS é a unidade básica de medida no CSS usada para definir tamanhos, espaçamentos e outras propriedades de estilo. Ele é uma unidade relativa que pode variar de tamanho dependendo das configurações de exibição, resolução de tela e do dispositivo.

Para criação da borda, o único método útil que o *context* do Canvas provê é o método *isPointInPath*. Este método avalia se um determinado pixel, especificado por coordenadas cartesianas, está dentro do *path*. Em nossa implementação, para definir a borda do *path*, percorre-se todos os *pixels* dele e, para cada *pixel*, se um de seus oito vizinhos não faz parte do *path*, então esse *pixel* é considerado um *pixel* de borda. Sendo um pixel de borda, ele é colorido com a cor escura.

Isso garante que a borda terá 1 *pixel* CSS de largura. Para que o segundo pixel mais interno seja colorido, utilizamos a seguinte estratégia: para cada *pixel* do *path*, se um de seus oito vizinhos for um *pixel* de borda, colorimos ele com a cor escura também.

Figura 6 – Representação dos *pixels* internos e externos de um *path*.

e	e	b_3
e	b_2	i
e	b_1	i

Fonte: Elaborada pelo autor.

Na figura 6, ilustramos o procedimento. Seja (x_1, y_1) o par de coordenadas do pixel b_2. Ao analisarmos b_2, verificamos inicialmente que ele é vizinho de um *pixel* externo e com coordenadas $(x_2 = x_1 - 1, y_2 = y_1)$. Portanto, ele é classificado como *pixel* de borda. Dessa forma, consideramos o *pixel* $(x_1 + 1, y_1)$ como o segundo *pixel* de borda caso ele esteja no *path*, pois ele está a uma distância de 1 *pixel* de b_2.

Dessa forma, como todos os *pixels* de borda fazem parte do *path*, podemos inserir e apagar o anel de foco sem chamar a função que reconstrói todo o Canvas novamente.

3.2 clearElementPath

Para desfazer a associação entre o elemento entre o *path* e o elemento do HTML, o desenvolvedor pode optar por excluir o elemento HTML. Neste caso, as associações de evento são naturalmente perdidas. Caso não deseje destruir o elemento, a desassociação é feita removendo os eventos do *path*.

4 Aplicabilidade

A implementação apresentada neste trabalho está disponível na linguagem JavaScript[3] e é suportada pela maioria dos navegadores atuais. Os navegadores Internet Explorer, nas versões 5.5 a 8, inclusive, necessitam de *polyfill* para assegurar retrocompatibilidade. O uso da classe pode facilmente ser incorporada a um código JavaScript Vanilla, isto é, JavaScript puro sem bibliotecas ou *frameworks*, apenas inserindo todo o código-fonte da implementação ou carregando-o de um código minificado[4] através de uma URL diretamente no elemento *script* do HTML.

A aplicação desta ferramenta é ampla e impacta diretamente a usabilidade de recursos visuais em diversas plataformas web e é voltada, sobretudo, a *frameworks* e bibliotecas que desejem implementar os recursos de acessibilidade já aqui apresentados, podendo ser facilmente incorporados em projetos existentes e novos desenvolvimentos.

Na Figura 7, apresentamos o código necessário para criar um quadrado azul na tela utilizando a API pura do Canvas.

Na Figura 8, apresentamos o código com o mesmo objetivo, mas utilizando a classe Graphics e indicando que o quadrado azul é um botão clicável já existente na *subDOM* (no código, referenciado por *element*).

Na Figura 9, apresentamos o código também com esse objetivo e utilizando a classe Graphics, mas permitindo que o objeto gere o elemento da *subDOM* automaticamente. Dessa forma, o botão passa a ser acessível para ferramentas de leitura de tela. Além disso, passamos para o botão duas funções de *callback* (no exemplo, vazias): *onClick* e *onFocus*, que definem os ouvintes de evento de clique e foco.

Figura 7 – Código para criação de um quadrado azul sem acessibilidade com a API do Canvas.

```
const square = new Path2D();
square.rect(x, y, 100, 100);
ctx.fillStyle = "#0000FF";
ctx.fill(square);
```

Fonte: Elaborada pelo autor.

Figura 8 – Código para criação de um quadrado azul acessível com a classe *Graphics*.

```
const square = new Graphics({
  id: "quadrado azul",
  context: ctx,
  canvas: canvas,
  pathColor: "#0000FF",
  focable: true,
});
```

[3] Código-fonte disponível em https://graphics-pied.vercel.app/Graphic.js
[4] Código-fonte minificado disponível em https://graphics-pied.vercel.app/graphics.min.js

```
const path = square.path;
path.rect(x, y, 100, 100);
square.setElementPath(element);
square.draw();
```

Fonte: Elaborada pelo autor.

Figura 9 – Código para criação de um quadrado azul acessível com a classe *Graphics* sem referenciar um elemento da *subDOM*.

```
const square = new Graphics({
  id: "quadrado azul",
  context: ctx,
  canvas: canvas,
  pathColor: "#0000FF",
  tag: "button",
  focable: true,
});
const path = square.path;
path.rect(startX, startY, 100, 100);
square.draw();
square.onFocus(() => {});
square.onClick(() => {});
```

Fonte: Elaborada pelo autor.

No caso da Figura 9, precisamos explicitar qual será a *tag* do elemento HTML associado. Conforme a Figura 2 apresenta, se nenhuma *tag* for passada, consideramos que a *tag* do elemento será a *div*.

5 Avaliação

A avaliação da ferramenta desenvolvida neste artigo se deu pelo estudo de três casos diferentes. Em cada um dos casos, analisamos: a capacidade de foco em *paths* definidos como focáveis; se um *path*, quando focado, é capaz de receber eventos do teclado e se o conteúdo do elemento semântico associado ao *path* é lido por uma ferramenta de acessibilidade quando este *path* é focado. Este último teste se deu pelo uso de três leitores de telas em três navegadores distintos. No navegador Microsoft Edge, utilizamos a ferramenta Microsoft Narrator, uma ferramenta de leitura de tela integrada ao Windows. No navegador Google Chrome, utilizamos a extensão Screen Reader Extension. E no navegador Firefox, utilizamos a ferramenta NVDA (NonVisual Desktop Access).

5.1 Estudo de Caso: construção de um gráfico de barras

Gráficos de dados são frequentemente construídos utilizando o Canvas HTML5 devido à sua flexibilidade e desempenho em renderização gráfica. Neste estudo de caso, nos baseamos na biblioteca Chart.js, que também utiliza Canvas para renderização de gráficos. A biblioteca Chart.js é uma ferramenta de fácil utilização para a criação de gráficos interativos e visualizações de dados na web. Para o teste, replicamos um mesmo gráfico desenvolvido com Chart.js (a princípio, não acessível), mas utilizando a ferramenta desenvolvida neste artigo.

A Figura 10 apresenta o gráfico desenvolvido com a biblioteca Chart.JS; a Figura 11, o gráfico desenvolvido sem biblioteca e com a ferramenta desenvolvida neste artigo.

Figura 10 – Gráfico desenvolvido com Chart.JS.

Fonte: Chart.JS (2024).

Figura 11 – Gráfico desenvolvido utilizando a classe *Graphics*.

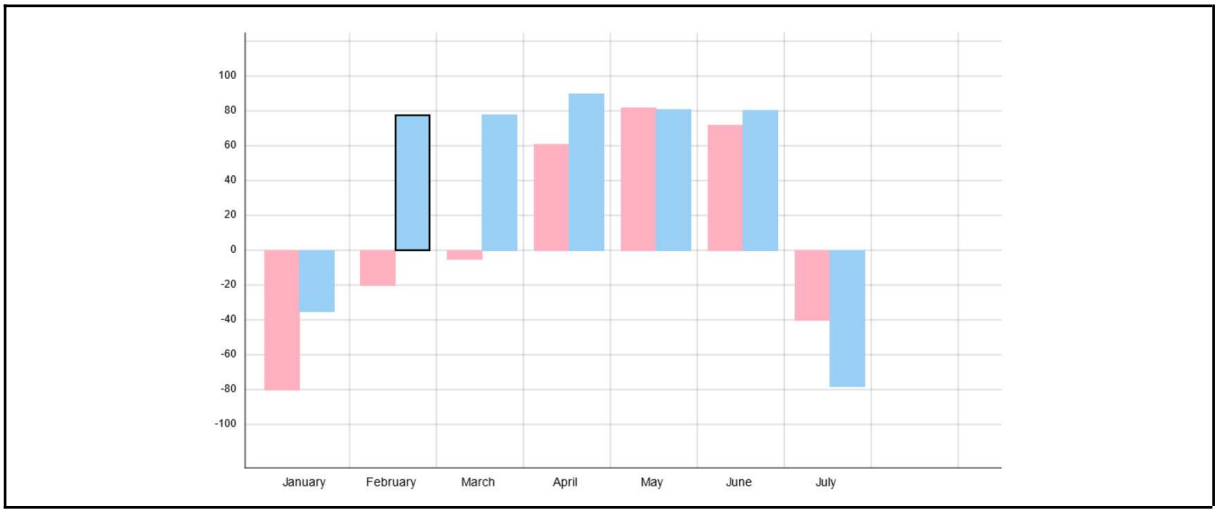
Fonte: Elaborada pelo autor.

No caso desenvolvido com Chart.JS, o Canvas não era acessível. Nenhuma das ferramentas foi capaz de ler as informações descritas nele. Ao passar o *mouse* sobre as barras, era exibido um *modal* flutuante indicando o ponto exato do eixo *y* que a barra em questão atingia.

No caso desenvolvido com nossa ferramenta, os eixos não eram lidos, mas todas as barras eram focáveis e, ao focar em cada um delas, a ferramenta lia a informação da barra em questão. A Figura 12 ilustra o caso em que uma das barras foi focada utilizando o teclado. Neste ponto, as três ferramentas leram a informação do ponto no eixo *y*. No caso da Figura 12, as ferramentas leram o número 78.

Em suma, todos os *paths* classificados como focáveis foram focados e suas informações foram lidas pelas três ferramentas.

Figura 12 – Barra azul do mês de fevereiro focada.



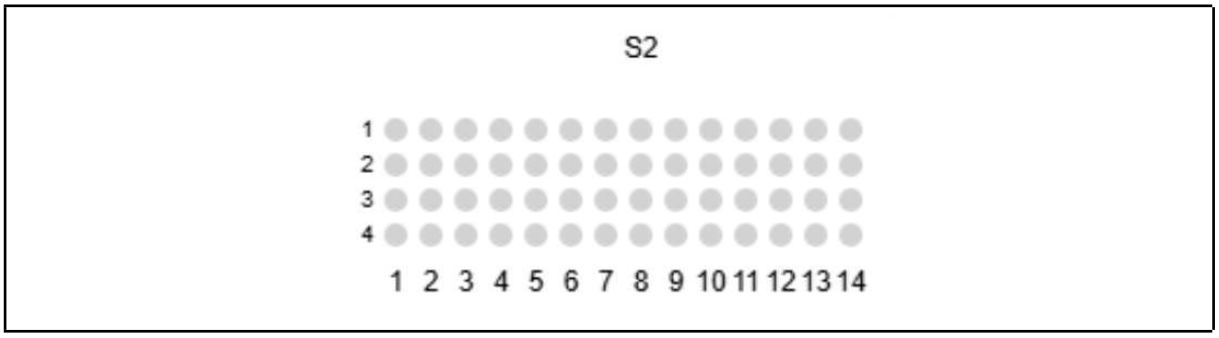
Fonte: Elaborada pelo autor.

5.2 Estudo de Caso: reserva de assentos

Painéis de reserva de assentos também costumam ser construídos com o Canvas. Neste estudo, emulamos um caso de uso da biblioteca Konva. A biblioteca Konva é uma ferramenta para criação de gráficos e animações 2D em aplicações web, oferecendo uma gama de recursos para manipulação de gráficos vetoriais e formas. Este exemplo, apesar de parecer trivial, oferece o desafio de possuir múltiplas renderizações do Canvas com atualizações do estado a cada *input* do usuário.

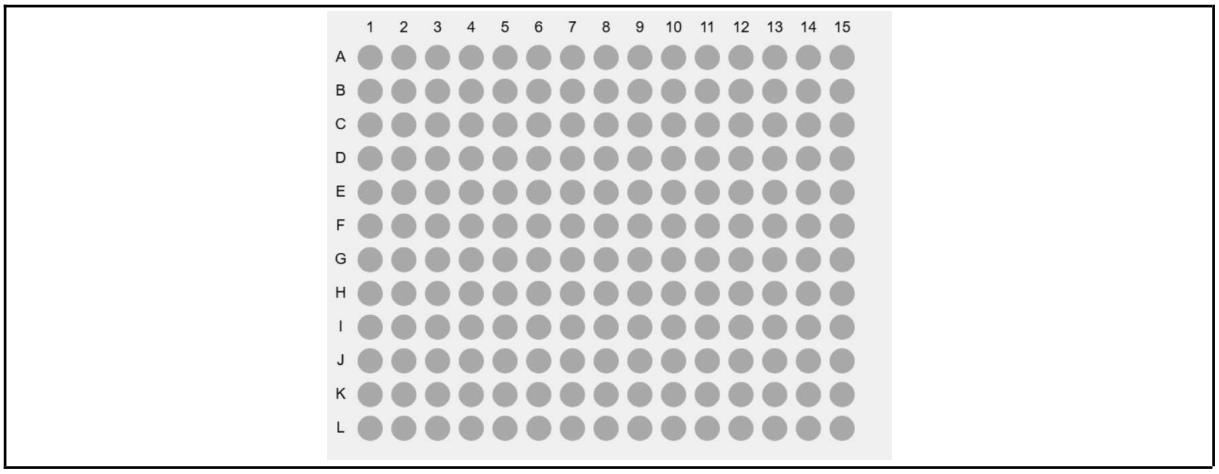
A Figura 13 apresenta o painel desenvolvido pela biblioteca Konva (adiante, Painel 1) e a Figura 14, o painel desenvolvido sem biblioteca e com a ferramenta desenvolvida neste artigo (adiante, Painel 2).

Figura 13 – Painel desenvolvido com a biblioteca Konva.



Fonte: Konva.JS.

Figura 14 – Painel desenvolvido com a classe *Graphics*.

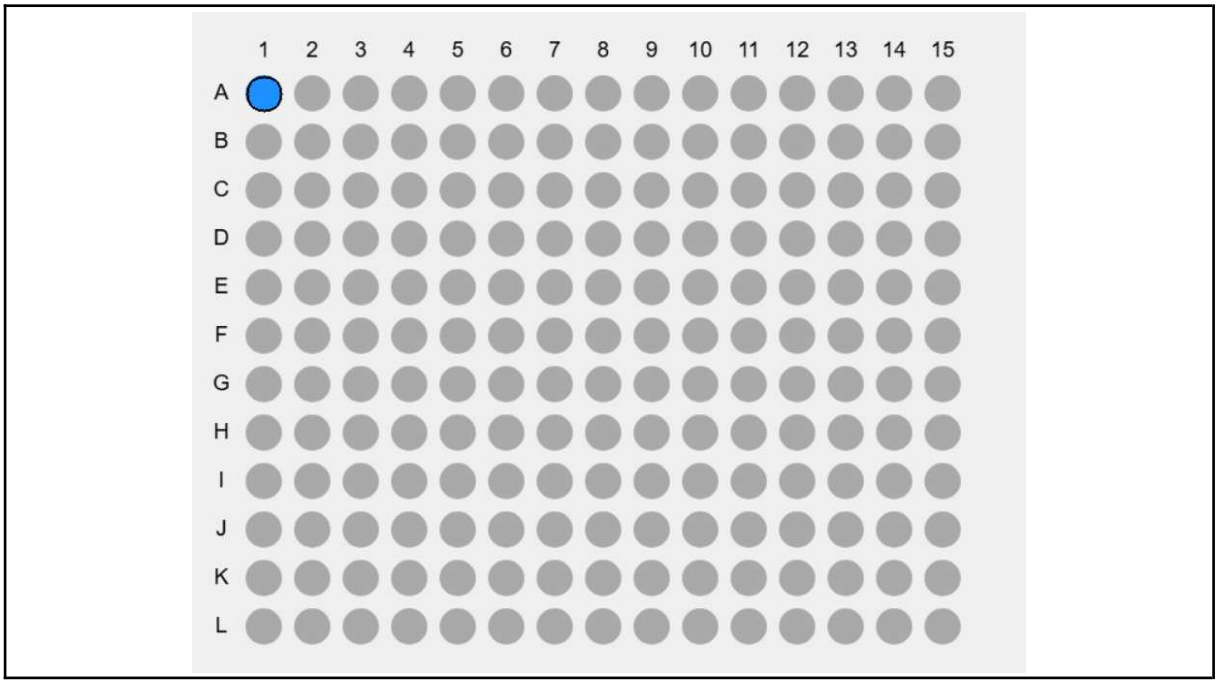


Fonte: Elaborada pelo autor.

O Painel 1 realizava a mudança de cursor do *mouse* quando este estava sobre áreas clicáveis ou não-clicáveis. No entanto, não foi lido por nenhuma das três ferramentas. Nenhuma informação interna ao Canvas estava exposta no código HTML.

O Painel 2, ao contrário, permitiu a leitura da informação do assento quando um círculo cinza era focado. Ao ser focado, o usuário poderia pressionar a tecla *Enter* e disparar o evento, que selecionava o assento e mudava a cor do círculo, conforme ilustrado pela Figura 15.

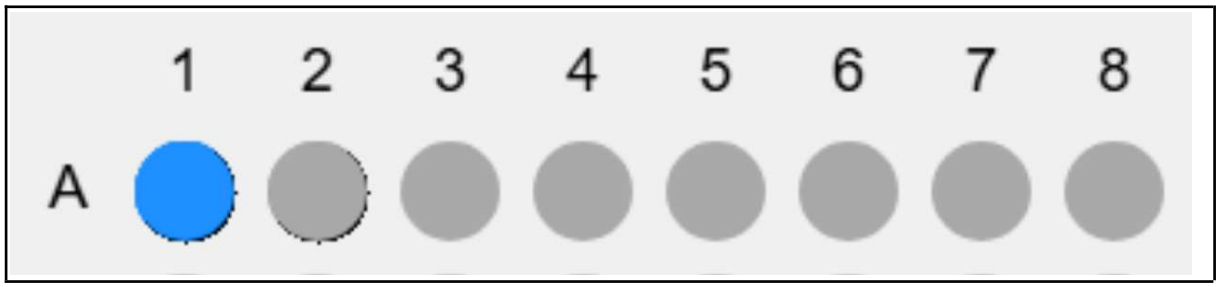
Figura 15 – Assento focado após um clique utilizando apenas o teclado.



Fonte: Elaborada pelo autor.

No entanto, o anel de foco do *path* apresentou alguns resíduos ao ser removido o foco. A Figura 16 apresenta tais resíduos. No entanto, esses *pixels* residuais não interferiram na acessibilidade dos *paths*, que foram corretamente lidos pelas três ferramentas e focados pelo teclado.

Figura 16 – *Pixels* residuais do anel de foco nos assentos A1 e A2.



Fonte: Elaborada pelo autor.

5.3 Estudo de Caso: manipulação de mapas

Mapas digitais frequentemente utilizam o Canvas HTML5 para renderizar e manipular visualizações geoespaciais com alto desempenho e flexibilidade. A abordagem baseada em Canvas permite desenhar mapas vetoriais e rasterizados diretamente no navegador, fornecendo um controle maior sobre a aparência do mapa. Neste estudo de caso, nosso objetivo é demonstrar a possibilidade de adicionar detalhes e descrições a mapas de forma acessível utilizando o Canvas.

Para tal, utilizamos o API Maps JavaScript, uma API do Google Maps adaptada para JavaScript. Além disso, utilizamos também um conjunto de dados[5] da The Meteoritical Society, que contém 45716 registros com informações sobre todos os pousos de meteoritos conhecidos. Renderizamos o gráfico básico utilizando esta API e, a partir de parte deste conjunto de dados, renderizamos, utilizando a ferramenta aqui desenvolvida, *paths* que representam os locais em que esta queda de meteoritos ocorreu. Este exemplo foi baseado em uma demonstração da biblioteca Mappa.js. Cada um destes *paths* está associado a um elemento HTML, cujo conteúdo textual indica o nome do meteorito e seus dados de queda (longitude e latitude). A Figura 17 apresenta este caso.

Figura 17 – Recorte do mapa-múndi com indicação de locais de quedas de meteoritos (em vermelho).

[5] Disponível em https://data.nasa.gov/Space-Science/Meteorite-Landings/gh4g-9sfh/about_data

Fonte: Elaborada pelo autor.

As três ferramentas de acessibilidade foram capazes de ler dados relativos à posição em que os meteoritos caíram. No entanto, pela quantidade de dados, tornou-se difícil identificar qual dos locais estava sendo focado. Alguns elementos foram focados sem que estivessem na *viewport*.

Além disso, utilizando todo o *dataset*, as renderizações do Canvas tornaram-se muito lentas, o que nos obrigou a reduzir para um décimo do tamanho original do *dataset*.

6 Conclusão e trabalhos futuros

O Canvas, apesar de sua flexibilidade e capacidade de criar gráficos dinâmicos, apresenta desafios significativos para a acessibilidade devido à sua natureza puramente gráfica e à ausência de representação semântica na árvore DOM. Isso dificulta a interação para usuários com deficiências visuais e limita a eficácia das ferramentas de acessibilidade convencionais.

Para superar esses desafios, a proposta de adicionar os métodos *setElementPath* e *clearElementPath* à API do Canvas do W3C representa um avanço importante. Esses métodos visam criar uma ponte entre o Canvas e elementos da DOM *fallback*, proporcionando uma alternativa acessível para o conteúdo gráfico, além de oferecer uma forma de criar a subDOM automaticamente.

Apenas dos bons resultados aqui descritos, o anel de foco implementado na classe Graphics, por ser do tipo *border*, sofre da desvantagem de remover parte do conteúdo do *path* em que está inserido. Em *paths* com imagens, a diferença se torna mais evidente. Para contornar isso, sugere-se a implementação do anel de foco do tipo *outline* ou *outset*, sendo necessário armazenar na memória os *pixels* apagados para reconstruí-los depois que o foco tiver sido removido do elemento.

Além disso, trabalhos posteriores podem também estudar o aumento do desempenho da ferramenta em cenários que requerem mais memória e mais desempenho do navegador, reduzindo o tempo de processamento para criar os elementos semânticos na subDOM, o que se mostrou um gargalo no terceiro estudo de caso.

REFERÊNCIAS

ABUADDOUS, H. Y.; JALI, M. Z.; BASIR, N. **Web Accessibility Challenges**. International Journal of Advanced Computer Science and Applications (IJACSA), v. 7, n. 10, 2016.

AGHDAM, Paniz Alipour; RAVANMEHR, Reza. **A novel approach for canvas accessibility problem in HTML5**. IJCSI International Journal of Computer Science Issues, v. 10, p. 116, 2013.

CHART.JS. **Vertical Bar Chart**. Disponível em: https://www.chartjs.org/docs/latest/samples/bar/vertical.html. Acesso em: 28 jul. 2024.

CONFORTO, Débora; SANTAROSA, Lucila Maria Costi. **Acessibilidade à Web: Internet para Todos**. *Informática na Educação: Teoria e Prática*, Porto Alegre, v. 5, n. 2, p. 90-91, nov. 2002.

FAULKNER, S. **ISSUE-105: Allow image maps on the canvas element**. World Wide Web Consortium (W3C) HTML Working Group. Disponível em:

https://www.w3.org/html/wg/tracker/issues/105. Acesso em: 15 mai. 2024. Data de criação: 31 mar. 2010.

FULTON, Steve; FULTON, Jeff. **HTML5 Canvas: Native Interactivity and Animation for the Web**. 2. ed. O'Reilly Media, 2013. p. 32-33.

GAVER, William W. **Technology affordances**. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. 1991.

KONVA.JS. **Seats Reservation**. Disponível em: https://konvajs.org/docs/sandbox/Seats_Reservation.html. Acesso em: 02 ago. 2024.

STEINER, Thomas. **Toward making opaque web content more accessible: accessibility from Adobe Flash to canvas-rendered apps**. Companion Proceedings of the ACM on Web Conference 2024. WWW '24, p. 1111-1114, 2024.

W3C. **Web Content Accessibility Guidelines (WCAG) 2.2. Success Criterion 2.4.13: Focus Appearance**. Disponível em: https://www.w3.org/WAI/WCAG22/Understanding/focus-appearance.html. Acesso em: 20 jun. 2024.

W3C. **Canvas Hit Testing**. Disponível em: https://www.w3.org/wiki/Canvas_hit_testing. Acesso em: 20 mai. 2024.

WHATWG. **HTML Living Standard: Canvas**. Disponível em: https://html.spec.whatwg.org/multipage/canvas.html#best-practices. Acesso em: 11 jul. 2024.

WORLD WIDE WEB CONSORTIUM (W3C). **Understanding Success Criterion 2.4.11: Focus Appearance (Minimum) | Understanding WCAG 2.2.** Disponível em: https://www.w3.org/WAI/WCAG22/Understanding/focus-appearance.html. Acesso em: 4 mai. 2024.

ZEN, D. **ZIM - JavaScript Framework**. Disponível em: https://zimjs.com/docs.php?view=30.5&title=Accessibility&line=18456. Acesso em: 18 mai. 2024.