A Taxonomy of HTML5 Canvas Bugs

Finlay Macklon, Markos Viggiato, Natalia Romanova, Chris Buzon, Dale Paas, Cor-Paul Bezemer

Abstract—The HTML5 `<canvas>` is widely used to display high quality graphics in web applications. However, the combination of web, GUI, and visual techniques that are required to build `<canvas>` applications, together with the lack of testing and debugging tools, makes developing such applications very challenging. To help direct future research on testing `<canvas>` applications, in this paper we present a taxonomy of `<canvas>` bugs. First, we extracted 2,403 `<canvas>`-related bug reports from 123 open source GitHub projects that use the HTML5 `<canvas>`. Second, we constructed our taxonomy by manually classifying a random sample of 332 bug reports. Our manual classification identified five broad categories of `<canvas>` bugs, such as Visual and Performance bugs. We found that Visual bugs are the most frequent (35%), while Performance bugs are relatively infrequent (5%). We also found that many `<canvas>` bugs that present themselves visually on the `<canvas>` are actually caused by other components of the web application. Our taxonomy of `<canvas>` bugs can be used to steer future research into `<canvas>` bugs and testing.

Index Terms—html5 canvas, web applications, bug taxonomy, issue reports

1 INTRODUCTION

The HTML5 `<canvas>` allows the rendering of high quality graphics in web applications. The Canvas API and WebGL API each provide JavaScript methods for drawing graphics on the HTML `<canvas>` element. The `<canvas>` is widely-used to develop web applications such as data visualizations, animations, and web games. The use of `<canvas>` in web applications is expected to grow even further, because its main alternative, Adobe Flash, is no longer supported in modern browsers as of 2021 [1], [2].

However, developers face challenges when working with the `<canvas>`. For example, a study of web development Q&A forum posts found that the `<canvas>` can cause confusion for developers, which is attributed to a lack of API documentation [3]. In terms of web applications, API documentation is arguably particularly important for the `<canvas>`, as `<canvas>` applications operate differently from traditional web applications. Traditional web applications are controlled through the Document Object Model (DOM), which represents the web page as nodes and objects [4]. Testing methods for traditional web applications often rely on the DOM to test the page. However, instead of being represented in the DOM, `<canvas>` contents are represented as a bitmap which is directly manipulated using the Canvas API or WebGL API [5]. The lack of a DOM representation of `<canvas>` contents makes it less intuitive and much harder to test `<canvas>` applications.

To better understand the issues that developers face with the `<canvas>`, we address the following research question in our study: What types of bugs do developers encounter when creating web applications with the HTML5 `<canvas>`?

In this paper, we construct a taxonomy of `<canvas>` bugs to give better insights on how `<canvas>` bugs are different from generic web application bugs, and to direct future research on testing the HTML5 `<canvas>`. In our study, we build an understanding of the types of `<canvas>` bugs by manually analyzing the contents of 332 `<canvas>`-related bug reports in 123 open source GitHub projects. We combine an existing taxonomy of GUI bugs [6] and an existing taxonomy of web bugs [7] to provide a baseline taxonomy that contains an initial set of bug types. Then, we perform a manual classification process to classify bugs into the existing types and define new bug types as necessary. The main contribution of our paper is a taxonomy of HTML5 `<canvas>` bugs, which allows future researchers to better target their efforts on testing methods for the `<canvas>`.

The remainder of our paper is structured as follows. In Section 2 we provide background information to motivate our study. In Section 3 we discuss related work. We present our methodology in Section 4. We present our taxonomy of `<canvas>` bugs in Section 5. In Section 6 we compare `<canvas>` bugs to generic web bugs and GUI bugs. In Section 7 we discuss future research directions. In Section 8 we discuss threats to validity. Section 9 concludes our paper.

2 BACKGROUND AND MOTIVATION

2.1 HTML5 `<canvas>` applications

The HTML5 `<canvas>` is particularly useful for web applications which require dynamic graphics [8], such as animations, interactive data visualizations, or web games. A `<canvas>` application can be made interactive by making it listen and respond to browser events such as mouse clicks [9]. After the occurrence of such an event, its associated JavaScript code is executed, which changes the `<canvas>` bitmap. This interactive behaviour allows developers to create ‘complete’ applications inside the `<canvas>` element. As a result, many `<canvas>` applications have their own GUI, which is rendered on the `<canvas>` bitmap as well. Because `<canvas>` applications have a GUI, deal
The existing web testing techniques are severely limited in Selenium DOM, such as used test automation frameworks that rely on analyzing the contents are not represented in the DOM. As web testing is typically done through several widely-used test automation frameworks, making development challenging.

2.2 Difficulties in \texttt{<canvas>} testing

To illustrate the challenges of testing the HTML5 \texttt{<canvas>}, we provide a simple motivating example. Figure 1a shows an empty \texttt{<canvas>} element on a web page. Figure 1b shows the updated \texttt{<canvas>} element after executing the JavaScript code shown in Listing 1. Figure 1c shows the DOM representation of the \texttt{<canvas>} element for both Figure 1a and Figure 1b. Clearly, the \texttt{<canvas>} contents have changed, but the DOM representation of the \texttt{<canvas>} element has not changed, showing that the \texttt{<canvas>} contents are not represented in the DOM.

As web testing is typically done through several widely-used test automation frameworks that rely on analyzing the DOM, such as Selenium\textsuperscript{[1]} Cypress\textsuperscript{[2]} and Playwright\textsuperscript{[3]}, existing web testing techniques are severely limited in \texttt{<canvas>} testing. There are many websites which use the \texttt{<canvas>}, yet there is no good way to test the \texttt{<canvas>}, meaning these websites either use manual testing, or do not test the \texttt{<canvas>} at all. With a lack of available \texttt{<canvas>} testing tools, there are opportunities for future research. A bug taxonomy provides a detailed classification of possible bug types, and a detailed classification is necessary to build testing tools that target different types of bugs\textsuperscript{[4]}. Additionally, challenges in brainstorming useful approaches to testing can be mitigated by using a bug taxonomy\textsuperscript{[5]}. Bug taxonomies can help identify gaps in our knowledge and direct future research\textsuperscript{[6]}. Therefore, our taxonomy of \texttt{<canvas>} bugs can be used to guide future research on \texttt{<canvas>} testing.

3 RELATED WORK

3.1 HTML5 \texttt{<canvas>} bugs

While there is extensive prior work on web bugs and web testing, there is limited prior work that investigates \texttt{<canvas>} bugs. As a discussion of related work on non-\texttt{<canvas>} web testing is outside the scope of our paper, we refer to the systematic literature review by Do˘gan et al.\textsuperscript{[7]} and the grey literature review by Ricca and Stocco\textsuperscript{[8]} for an overview of related work about web testing.

Bajammal and Mesbah\textsuperscript{[9]} propose an approach for automated visual testing of the \texttt{<canvas>}, and report high accuracy. However, their approach is evaluated by injecting only a single type of \texttt{<canvas>} bug. In our work, we investigate the types of \texttt{<canvas>} bugs that developers encounter, so that future research on \texttt{<canvas>} testing can target a wider range of relevant bugs.

Hoetzlein\textsuperscript{[10]} conducted a study on graphics performance in web applications, finding HTML5 \texttt{<canvas>} showed different performance across different browsers. However, a difference in performance may not constitute a bug, and our study is concerned with many types of \texttt{<canvas>} bugs, rather than just performance bugs on \texttt{<canvas>}.

Bajaj et al.\textsuperscript{[11]} investigated Stack Overflow questions asked by web developers, and found that developers faced implementation issues with the \texttt{<canvas>}. They reason that this is due to a lack of clear API documentation. While this does suggest that developers have problems with the \texttt{<canvas>}, our work further investigates problems with the \texttt{<canvas>} by identifying the types of bugs developers face.

3.2 Related bug taxonomies

There are some existing bug taxonomies that are related to the GUI, web, and graphics rendering characteristics of the \texttt{<canvas>}. Lelli et al.\textsuperscript{[12]} constructed a taxonomy of GUI bugs that allows for the evaluation of GUI testing tools. Marchetto et al.\textsuperscript{[13]} constructed a taxonomy of web bugs that focuses on web architecture bugs. Our work differs in that we focus specifically on \texttt{<canvas>} bugs, allowing us to determine whether these taxonomies are useful in describing \texttt{<canvas>} bugs, and define new bug types for the \texttt{<canvas>} as required.

Ocariza et al.\textsuperscript{[14]} constructed a taxonomy of client-side JavaScript bugs, finding that most client-side JavaScript bugs are DOM-related. As explained in Section 2, the \texttt{<canvas>} contents are not represented in the DOM, meaning the taxonomy of client-side JavaScript bugs has limited relevance to \texttt{<canvas>} bugs.

Woo et al.\textsuperscript{[15]} provide a taxonomy of problems in rendering algorithms. While their taxonomy may be complimentary to our taxonomy (in describing the causes of Rendering bugs), our work takes a higher-level and broader view of \texttt{<canvas>} bug types.

4 METHODOLOGY

In this section, we describe our methodology for constructing a taxonomy of \texttt{<canvas>} bugs. Table 1 provides an overview of how we used taxonomy construction guidelines provided by Usman et al.\textsuperscript{[16]} and Ralph\textsuperscript{[17]} to facilitate the

---

1. \url{https://www.selenium.dev/}
2. \url{https://www.cypress.io/}
3. \url{https://playwright.dev/}

---

Fig. 1: \texttt{<canvas>} contents are not represented in the DOM.

![Canvas API example](image-url)
construction of our taxonomy. Bug taxonomies have been defined in a variety of ways, which is mainly due to differences in objectives \[43\]. However, bug taxonomies have commonly been constructed via the manual classification of relevant bug reports \[14\]. Our goal was to guide future research on \(<\text{canvas}>\) testing by uncovering the bug types (based on their symptoms) that are reported in \(<\text{canvas}>\) projects, and manually classifying bug reports provides the insight we require to build a taxonomy of \(<\text{canvas}>\) bugs. Therefore to construct our taxonomy of \(<\text{canvas}>\) bugs, we collected and manually classified \(<\text{canvas}>\) bug reports from GitHub. We took an initial set of labels from existing taxonomies, based on known aspects of the HTML5 \(<\text{canvas}>\): the \(<\text{canvas}>\) is used as a GUI, for graphics rendering, and is a web technology. An overview of our methodology can be seen in Figure 2.

### 4.1 Collecting \(<\text{canvas}>\) bug reports

To analyze the types of bugs that developers face when using the \(<\text{canvas}>\), we looked at \(<\text{canvas}>\)-related bug reports in popular open source projects that use the \(<\text{canvas}>\). Bug reports in popular open source projects that use the \(<\text{canvas}>\) are appropriate to study because they can contain detailed descriptions and discussions of real-world bugs encountered by software developers, as shown by previous studies that mine bug reports in popular open source projects \[4-6,14,46\]. We have made our collected data available on Zenodo at the following link: https://zenodo.org/record/6886143

#### 4.1.1 Extracting \(<\text{canvas}>\) projects.

We created a custom crawler that utilized the GitHub API to search GitHub for open source projects that use the \(<\text{canvas}>\). We searched for open source projects on GitHub that matched the keywords ‘canvas’, ‘html5-canvas’, or ‘html5-canvas’ in their title, description, readme file, or topic tags. We limited our search to projects which use the following languages: HTML, JavaScript, or TypeScript. To ensure only mature, popular projects were considered in this study, inclusion criteria of at least 1,000 stars and at least 2 contributors were set empirically, similar to prior work \[12,18,21\]. We avoid immature projects because bug reports for immature projects are not necessarily representative of real bugs, for example because the report is simply for a feature that is not finished or well-tested yet. Focusing on mature projects should improve the likelihood of bug reports being more representative. With our inclusion criteria applied, we had a list of 375 open source projects that matched our search criteria. We then manually filtered out false positives. Some false positives were due to our use of keyword matching in project readme files. For example, one false positive was \texttt{microsoft/\textasciitilde\texttt{Web-Dev-For-Beginners}}, which contains some guides for the \(<\text{canvas}>\) but does not utilize it in a specific software application. Other false positives were projects which utilized a non-HTML5 canvas object, such as \texttt{node-canvas} for Canvas API compatibility with \texttt{Node.js} desktop applications. For projects with a non-HTML5 canvas object, it was difficult to automatically ascertain whether bug reports discuss the \(<\text{canvas}>\) of interest, or the other canvas object. With the false positives removed, we were left with a list of 180 projects from which \(<\text{canvas}>\)-related bug reports could be extracted.

#### 4.1.2 Extracting \(<\text{canvas}>\) bug reports.

To collect bug reports for our analysis, we modified our custom crawler to search GitHub for relevant issue reports in each of the 180 projects. To ensure we study only relevant bug reports, we only include closed bug reports in our analysis, similar to prior work \[14,40\]. Closed bug reports contain the information required to understand the main symptom of the reported bug \[14\], for example, developer responses that confirm that the reported issue was indeed a bug and other relevant discussion in the comments of the bug report. We searched for closed issue reports with

| TABLE 1: Mapping of taxonomy construction guidelines \[34,41\] to our methodology. |
|-----------------------------|----------------|
| USMAN ET AL. \[41\] | RALPH  \[34\] |
| Planning | Choose a strategy | 4 |
| Identification and extraction | Site selection | 4.1 and 4.2 |
| Design and construction | Data collection | 4.1.1, 4.1.2 and 4.2.1 |
| Validation | Conceptual evaluation | 4.4 |
| Collecting \(<\text{canvas}>\) bug reports | Extract \(<\text{canvas}>\) bug reports | 180 mature, popular projects |
| Google Scholar | Select GUI & web papers | Taxonomies of GUI & web bugs |
| | Take random sample | Baseline taxonomy |
| | Perform pilot analysis | Baseline taxonomy |
| Constructing the taxonomy of \(<\text{canvas}>\) bugs | Take representative random sample | 332 \(<\text{canvas}>\) bug reports |
| | Perform manual classification | Taxonomy of \(<\text{canvas}>\) bugs |

Fig. 2: Study methodology overview.
4.2 Selecting the baseline taxonomy

To reduce subjectivity during the manual bug classification, we started our classification from a baseline taxonomy. To select the baseline taxonomy, a pilot analysis was conducted with a taxonomy of GUI bugs [22] and a taxonomy of web bugs [23].

4.2.1 Selecting papers with relevant taxonomies

As a web technology, we expected the <canvas> to exhibit bugs that are present in web applications. Also, the <canvas> is often used for both the GUI and graphics of web applications. Therefore, we also expected the <canvas> to exhibit bugs that are present in the GUIs of desktop applications. To find relevant bug taxonomies, we searched Google Scholar with the following queries: ‘web bug taxonomy’ and ‘gui bug taxonomy’. The use of Google Scholar to find relevant software engineering works is suggested by Brereton et al. [13], and provides a wide coverage of search results as discussed by Landman et al. [20]. The search results were manually analyzed, and two papers providing a relevant taxonomy were selected (one for GUI bugs [22] and one for web bugs [23]).

4.2.2 Performing the pilot analysis

A pilot analysis was conducted by the first author to verify that each of the selected taxonomies were useful for our purposes. Manual classification was performed with a random sample of 100 bug reports. Listing 2 shows the process for the pilot analysis. Once the pilot analysis was complete, we were able to define the baseline taxonomy using the selected taxonomies. For the GUI characteristic of the <canvas>, most bug types from the taxonomy provided by Lelli et al. [22] were adopted into the baseline taxonomy. For the web characteristic of the <canvas>, one of the bug types from the taxonomy provided by Marchetto et al. [23] was adopted into the baseline taxonomy. The baseline taxonomy can be seen in Table 2.

4.3 Constructing the taxonomy of <canvas> bugs

To construct our taxonomy of <canvas> bugs, we manually classified a sample of the bug reports that we collected. We randomly selected a statistically representative sample of 332 of our collected bug reports with a confidence level of 95% and a confidence interval of 5%. We used stratified sampling to reduce bias [24] towards projects with a higher number of bug reports in our dataset. To manually classify the bug reports, we performed a manual labelling process over two stages. First, we performed multi-label classification to determine which bug reports contained bug types already defined in the baseline taxonomy. Then, we performed card sorting primarily to elicit and define new bug types. When assigning labels to bug reports, we considered not only the title and description provided by the reporter, but also discussion by developers and users in the comments of the bug reports.

4.3.1 Independent multi-label classification

In the first stage of manual classification, the first and second authors each independently performed a multi-label classification of the sampled bug reports. Listing 3 shows the process we followed to perform multi-label classification. There were cases where no bug types were assigned to a bug report, because the baseline taxonomy was not fully descriptive of <canvas> bug types. Similar to prior work [42], to calculate agreement when performing multi-label classification of the bug reports, we calculated the percentage of agreement on a per label basis, and then averaged across the agreement measures. Table 2 shows the agreement measure for each bug type from the baseline taxonomy, and our overall (averaged) agreement was 86%. However, the first and second authors agreed that the GUI structure and aesthetics and Data presentation labels were not mutually exclusive, as these labels often had to be grouped together as if they were the same label. This is because the <canvas> can act as both a GUI and a graphics container for web applications. So, to improve our classification, we merged the labels from the User interface characteristic of

---

Table 2: The baseline taxonomy.

| Characteristic          | Type                          |
|-------------------------|-------------------------------|
| User interface          | Data presentation             |
|                         | GUI structure and aesthetics  |
| User interaction        | Action                        |
|                         | Behaviour                     |
| Web architecture        | Browser incompatibility        |

---

Listing 2: Steps to perform the pilot analysis.

For each bug report:
1. Manually analyze the contents of the bug report.
2. If the bug report matches an existing bug type:
   - Add the bug type(s) to the baseline taxonomy.
3. If the bug report does not match any existing bug type:
   - Merged the labels from the User interface.

Output = the baseline taxonomy.
bugs. After the first stage of manual classification, we were able to define new bug types for our taxonomy that are specific to the <canvas> bugs. Additionally, in Section 6 of our paper, we conducted an explicit benchmarking exercise with the two aforementioned taxonomies, including discussion of new bug types in our taxonomy that are specific to the <canvas> application.

4.4 Evaluating our taxonomy of <canvas> bugs

After completing the construction of our taxonomy, we evaluated our final taxonomy using established quality criteria for taxonomies [34], [41].

Utility demonstration: The utility of a taxonomy is demonstrated by classifying subject matter examples [38], [41], such as bug reports. To evaluate the utility of our taxonomy, the first and second author each independently performed a multi-label classification of a random sample of 25 bug reports from a proprietary <canvas> application. We found that we did not require any new labels to classify these bug reports, and attained an agreement measure of 98% (with the lowest agreement for a category being 88% and the highest 100%). We calculated agreement for our validation exercise the same way as described in Section 4.3.1. This result shows that our taxonomy is useful in describing <canvas> bugs found in real, industrial <canvas> applications, and therefore meets the utility criterion.

Orthogonality demonstration/Reflects (dis)similarities between instances: The orthogonality of a taxonomy, i.e., that the labels in the taxonomy do not overlap with each other, is implied by the design of the taxonomy [38], [41]. Our results in Section 5.6 indicate that there is a difference in resolution times between our bug types, which supports that there are differences between the bug types in our taxonomy. However, during the evaluation of the utility of our taxonomy, we observed that the proprietary issue reports sometimes required multiple labels. The need for multiple labels suggests that symptoms can occur together, and that the bug types in our taxonomy are not completely mutually exclusive. At the same time, the symptoms do not always occur together - indicating that the bug types in our taxonomy are dissimilar enough to warrant not merging them. As the goal of our taxonomy is to identify directions for research on <canvas> testing, we do not consider it problematic that the taxonomy is not completely orthogonal.

Benchmarking: Benchmarking involves comparing the constructed taxonomy with similar classification schemes [38], [41]. As described in Section 4.2.2, we conducted a pilot analysis to determine the relevance of labels in similar taxonomies (for GUI [22] and web [23] bugs) when constructing our taxonomy of <canvas> bugs, meaning our methodology for taxonomy construction contains an implicit benchmarking exercise with the two aforementioned taxonomies. Additionally, in Section 5 of our paper, we provided detailed discussion of the (dis)similarities between our final taxonomy and the GUI [22] and web [23] bug taxonomies, including discussion of new bug types in our taxonomy that are specific to the <canvas>.

Fit-for-purpose: To determine if our taxonomy is fit-for-purpose, we consider the degree to which our taxonomy is effective at the specific purpose it was designed for [34]. As defined in Section 1, our guiding question in constructing our taxonomy was "What types of bugs do developers encounter..."
when working with the HTML5 `<canvas>`?”, and our underlying motivation is to direct future research on testing the `<canvas>`. As discussed in Section 6.3, our taxonomy answers our research question by providing a detailed classification of bug types reported in popular open-source projects that utilize the `<canvas>`, including bug types defined in existing GUI [22] and web [23] bug taxonomies, as well as new bug types specific to the `<canvas>`.

Furthermore, in Section 7 we identified several future lines of research based on our taxonomy of `<canvas>` bugs. Based on these characteristics, we conclude that our taxonomy is fit-for-purpose.

### 4.5 Determining bug resolution time

In addition to constructing a taxonomy of `<canvas>` bugs, we also quantified bug resolution time for our set of classified bug reports to better understand how developers deal with different types of `<canvas>` bugs. To calculate the bug resolution time, we calculated the difference between the issue open time and the final issue close time for each bug report, similar to prior work [12]. We calculated the median resolution time for each bug characteristic in our taxonomy by taking the median of the resolution times for all bug reports associated with each bug characteristic.

#### 4.5.1 Collecting data for comparison

To provide a baseline for comparison for bug resolution times, we collected non-`<canvas>` bug reports from the projects in our studied population. We followed a similar process to how we identified `<canvas>` bug reports in the projects, except we collected only bug reports that did not mention the `<canvas>`.

### 5 Our taxonomy of `<canvas>` bugs

In this section, we provide a detailed description of each of the bug types in our taxonomy of `<canvas>` bugs. Table 4 shows our taxonomy of `<canvas>` bugs, including the frequency of each bug type in the studied bug reports.

#### 5.1 Visual bugs

**Visual** bugs are problems in the presentation of objects on the `<canvas>` bitmap. We discovered four types of Visual bugs: Rendering, Layout, State, and Appearance.

Rendering bugs are often related to the incorrect scaling of objects or images on the `<canvas>`, with example instances such as distortion or blurriness being common. Some instances of Rendering bug reports described unexpected visual results due to more specific logical errors in using the Canvas API (or WebGL API) to display specific shapes, such as a waveform [69]. Figure 3a shows an example instance of a Rendering bug [68]. We noticed that some Rendering bug reports discussed how the incorrect scaling occurred due to a failure in accounting for differences in device resolutions [66], [67], [82], [84], or changes in `<canvas>` dimensions [59], [65]. We also found an instance of a Rendering bug where artifacts appeared on the screen only when using WebGL on devices with an outdated graphics driver [52].

**Appearance** bugs are related to the incorrect aesthetics of objects on the `<canvas>`, such as incorrect color, transparency, or font. Figure 3d shows an example instance of an Appearance bug, with the legend of the graph having incorrect colors [62]. We found some instances of Appearance bugs where the incorrect font was used when rendering text on the `<canvas>` [56], [57].

**Layout** bugs are related to incorrect positioning and sizing of objects on the `<canvas>`. Figure 3b shows an example instance of a Layout bug, with misaligned objects or the incorrect coordinates (like \([x,y]\)) for an object [58], or the incorrect layering (like z-index) for an object [50] on the `<canvas>`.

In one instance, a data visualization was not fully viewable because some objects had been assigned positions outside of the `<canvas>` viewport [49]. In another instance, a `<canvas>` animation was not fully viewable due to the `<canvas>` dimensions being too small [77].

**State** bugs are those in which an object (or text) is displayed in a different state than it should be. For example, an object that is visible on the `<canvas>` should be invisible, or vice versa. Figure 3c shows an example instance of a State bug, with circles appearing when they should not be visible [55]. In one State bug report, game controls were not removed when changing views on the `<canvas>` to a text box for chat [80]. In another more complicated case, multiple HTML `<canvas>` elements were used to render different objects, and specific HTML `<canvas>` elements were not being hidden when they should have been [55].

**User interaction** bugs are related to DOM events that are triggered on the HTML `<canvas>` element. We discovered two types of User interaction `<canvas>` bugs. Action bugs occur when a single action fails to execute correctly. Meanwhile, Behaviour bugs are those which consist of multiple actions that together provide an incorrect result. Some example instances of Action bug reports we encountered include: clicking the cursor to zoom does not work when the cursor moves outside of the bounds of the HTML `<canvas>` element [58], and a tooltip is not showing when the mouse hovering over a graph rendered to the `<canvas>` [81]. Other instances of Action bug reports were similar in that a specific action was failing [47], [57], [78], which prevented user interaction functionality in web applications built with the `<canvas>`. For Behaviour bugs, some example instances we encountered include: drawing functionality does not work correctly while simultaneously scrolling within the `<canvas>` [63], and dragging an object on the `<canvas>` does not function as expected after scrolling within the `<canvas>` [71]. In general, Behaviour bugs involve two or more actions that are performed simultaneously or in succession, which ultimately provide incorrect functionality with the `<canvas>`. Although User interaction bugs are related to DOM events, they were often described in bug reports by comparing the expected and buggy visual results.

#### 5.2 User interaction bugs

**User interaction** bugs are related to DOM events that are triggered on the HTML `<canvas>` element. We discovered two types of User interaction `<canvas>` bugs. Action bugs occur when a single action fails to execute correctly. Meanwhile, Behaviour bugs are those which consist of multiple actions that together provide an incorrect result. Some example instances of Action bug reports we encountered include: clicking the cursor to zoom does not work when the cursor moves outside of the bounds of the HTML `<canvas>` element [58], and a tooltip is not showing when the mouse hovering over a graph rendered to the `<canvas>` [81]. Other instances of Action bug reports were similar in that a specific action was failing [47], [57], [78], which prevented user interaction functionality in web applications built with the `<canvas>`. For Behaviour bugs, some example instances we encountered include: drawing functionality does not work correctly while simultaneously scrolling within the `<canvas>` [63], and dragging an object on the `<canvas>` does not function as expected after scrolling within the `<canvas>` [71]. In general, Behaviour bugs involve two or more actions that are performed simultaneously or in succession, which ultimately provide incorrect functionality with the `<canvas>`. Although User interaction bugs are related to DOM events, they were often described in bug reports by comparing the expected and buggy visual results.

#### 5.3 Web architecture bugs

We discovered two types of Web architecture `<canvas>` bugs: Different behaviour across browsers, and Cross-origin resource sharing.
Different behaviour across browsers bugs occur when a `<canvas>` application works on some browsers but not others. Many such bugs were caused by the use of an outdated browser, such as IE8 [76], or an outdated version of a modern browser, such as Firefox 51.0 when the latest stable release was Firefox 63.0 [64]. However, we also found instances of Different behaviour across browsers bugs where `<canvas>` functionality was incorrect for a latest stable release version of a modern browser. For example, we found some instances where a `<canvas>` application did not function correctly on Safari, despite working on Firefox and Chrome [61], [74]. In another case, a specific browser bug in Chrome prevented a `<canvas>` application from functioning [60].

Cross-origin resource sharing (CORS) bugs are related to the incorrect use of CORS policies with `<canvas>`. CORS policies provide a secure method for loading `<canvas>` resources (such as images or WebGL textures) from foreign domains, but if they are not used correctly, the resources will either not be loaded [29], or prevent the `<canvas>` data from being saved to a file [26]. In one instance, an AWS bucket was not correctly configured with CORS policies, which prevented the loading of those resources to a `<canvas>` application [75]. In some instances, CORS headers were not used in the HTTP requests to foreign domains, preventing resources from being loaded [48], [53].

5.5 Integration bugs

When connecting the `<canvas>` with other parts of a web application, Integration bugs can occur. We discovered two types of Integration bugs: Saving `<canvas>` data and Browser runtime error.

Saving `<canvas>` data bugs can occur when saving the `<canvas>` bitmap to a file using custom JavaScript methods or JavaScript methods built into the browser. One instance described how saving the `<canvas>` bitmap using a built-in browser method produced a file with a black background that was not present in the browser [72]. This indicates that built-in methods such as `getImageData`, `toBlob`, or `toDataURL` may not produce the same result as what is seen on the `<canvas>`. Other instances of Saving `<canvas>` data bugs were due to errors in custom methods for saving the `<canvas>` bitmap, for example we found an instance where the saved data was unexpectedly compressed [70].

We discovered bug reports that discussed `<canvas>` functionality despite the bug being located in a different part of the web application, and classified such bug reports as Browser runtime error. While these bug reports did mention the `<canvas>`, these bugs were not actually directly related to `<canvas>` problems. Example instances of Browser runtime error bugs include: referencing an HTML `<canvas>` element that had not been created [79], and using incorrect syntax for a JavaScript library method that added objects to

![Image](image.png)
TABLE 5: Median bug resolution time per characteristic in our taxonomy.

| Characteristic in our taxonomy | Median time to resolve (days) |
|--------------------------------|-----------------------------|
| Integration                    | 4                           |
| Visual                         | 8                           |
| User Interaction               | 13                          |
| Performance                    | 32                          |
| Web Architecture               | 48                          |

a data visualization on the `<canvas>` [51]. It appears that most Browser runtime error bugs are reported by developers who are not experienced with web development or the `<canvas>`.

5.6 Results for bug resolution time
As described in Section 4.5, we quantified the median bug resolution times for each bug characteristic in our taxonomy to better understand how developers deal with different types of `<canvas>` bugs. Table 5 shows the median bug resolution times per bug characteristic in our taxonomy. Our results indicate that developers of open source projects that utilize the `<canvas>` may, for example, prioritize bug reports belonging to Visual bug types over bug reports belonging to Web Architecture bug types.

5.6.1 Resolution times for non- `<canvas>` bug reports
As described in Section 4.5.1, we also collected non- `<canvas>` bug reports to compare bug resolution times. For the non- `<canvas>` bug reports in the 123 studied projects, we found that the median resolution time was 6 days. Comparing this result with our results in Table 5, we can conclude that developers may either prioritize non- `<canvas>` bug reports over most `<canvas>` bug reports, or that `<canvas>` bug reports are harder to resolve. However, Integration bugs (from our taxonomy) are usually resolved more quickly than non- `<canvas>` bug reports.

6 DISCUSSION
Prior to performing classification, we hypothesized that `<canvas>` bugs are similar to the bugs found in GUIs and web applications. While we did discover some overlap, we found that some GUI bug types and many web bug types are not relevant to `<canvas>` applications. We also discovered some new types of bugs that are specific to the `<canvas>`. In this section, we discuss the differences between our taxonomy of `<canvas>` bugs and the taxonomies that contributed to the baseline taxonomy (that we selected in Section 4.2). A mapping of the selected taxonomies to our taxonomy of `<canvas>` bugs can be seen in Figure 4.

6.1 Comparing GUI bugs and `<canvas>` bugs
The key difference between GUI bugs and `<canvas>` bugs is that it is difficult to distinguish GUI structure and aesthetics and Data presentation for the `<canvas>`. Additionally, while several of the specific User interface bug types provided in the taxonomy of GUI bugs [22] were descriptive of Visual bugs on the `<canvas>`, we did not use Type/format or Properties in our classification. The absence of such bug types in our taxonomy may be explained by the fact that the `<canvas>` renders a bitmap rather than a structured markup language (e.g. HTML, XML) that contains separate properties for the display of raw numerical and textual data. With regards to User interaction bug types, we did not find any Feedback or Reversibility bugs in `<canvas>` bug reports. Together, these comparisons indicate that the `<canvas>` is like a specific kind of GUI that deals heavily with graphics rendering. However, our taxonomy of `<canvas>` bugs contains bug types that are not GUI-related, meaning `<canvas>` testing has additional complexities that are not seen in desktop GUIs.

6.2 Comparing web bugs and `<canvas>` bugs
Our taxonomy of `<canvas>` bugs contains very few bug types seen in the taxonomy of web bugs [23], indicating there is not a lot of overlap between `<canvas>` bugs and generic web application bugs. For the Web architecture aspect of `<canvas>`, the only obvious similarity between the taxonomy of web bugs [23] and our taxonomy of `<canvas>` bugs is the presence of the Different behaviour across browsers type. Given that `<canvas>` is a client-side technology (i.e. it is implemented by browsers), the existence of Different behaviour across browsers bugs is not surprising. In our taxonomy of `<canvas>` bugs, the Cross-origin resource sharing type is somewhat similar to the Server environment type in the taxonomy of web bugs [23]. Knowing that `<canvas>` is a client-side technology, one might not expect Server environment issues to affect `<canvas>` in web applications. However, the specific challenges posed by CORS means that there are some `<canvas>` bugs that could also be described...
by Server environment. We did not use Web components’ data exchanged, Extracting from database, or Form construction types, meaning <canvas> bugs do not range the full scope of generic web architecture bugs. The smaller scope of web-related <canvas> bugs compared to generic web application bugs is also indicated by the lack of many other types that were proposed in the taxonomy [23] from which we took initial Web architecture types.

### 6.3 Summary of findings

To complete the discussion of our constructed taxonomy of <canvas> bugs, below we summarize our findings with reference to the research question defined in the introduction section of our paper: What types of bugs do developers encounter when creating web applications with the HTML5 <canvas>?

**Summary:** As our taxonomy demonstrates, developers encounter a wide variety of bug types when creating web applications with the HTML5 <canvas>. These bug types span various characteristics of the <canvas>, such as Visual, Performance, and Web Architecture bugs. It is unlikely that there exists a silver bullet testing approach to cover this variety of <canvas> bugs. Therefore, different <canvas> testing approaches must be developed to test these different types of bugs.

### 7 Future research directions

#### 7.1 Detecting Visual bugs

The most frequently reported <canvas> bugs in the open source projects are Visual bugs, as can be seen in Table 4. Visual bugs may be hard to diagnose, because the <canvas> state is not exposed through the DOM. The only existing research [9] that addresses testing the <canvas> focuses on detecting what we have defined as State bugs in our taxonomy. Although there is extensive prior work regarding Visual GUI testing [3], [10], this testing paradigm is not synonymous with detecting Visual bugs [17]. Visual bugs would interfere with Visual GUI testing methods that leverage the visual aspect of the application to drive test automation. Snapshot testing is a Visual GUI testing approach that could target Visual bugs on the <canvas>, however snapshot testing has many inherent drawbacks that limit its utility as an automated testing strategy [9]. And future research is required to understand what extent snapshot testing is useful in catching Visual bugs on the <canvas> in web applications. This emphasizes that research on <canvas> testing is still in an early stage, and that there are many opportunities for future research on <canvas> testing. Given the high frequency and notable variety of Visual bugs, new approaches for testing the <canvas> must be developed.

#### 7.2 Detecting correct usage of the <canvas>

Using the <canvas> can lead to new challenges in terms of performance. While the <canvas> opens the door to many interesting use-cases in web applications, it also may lead to Performance issues that are specific to the <canvas>. As seen in Table 4, one of our newly defined types of <canvas> bugs is Inefficient memory usage, which can lead to memory leaks or high CPU usage. While much of the graphics rendering with the <canvas> is optimized underneath, particularly when using WebGL, it is still up to developers to utilize the provided APIs in a performant manner. The same <canvas> animation might be produced through several different approaches, and some approaches outperform others. In lieu of performance testing tools for the <canvas>, developers may be able to utilize performance profiling tools built into a browser, such as Chrome DevTools [11], to analyze the runtime performance of web applications built with the <canvas>. Alternatively, an approach to profiling graphics performance such as the one detailed by Hoetzlein [16] could also prove useful. However, performance analysis is not an automated testing approach, and automated performance testing would be very difficult with performance profiling tools due to their computational overhead. Therefore, future research should investigate the detection of Inefficient memory usage bugs.

As shown in Table 4, one of our newly defined bug types is Cross-origin resource sharing. To render resources (such as images or WebGL textures) from foreign domains to the <canvas>, the foreign domains must be configured to allow CORS requests, and any HTTP request for these resources must contain a valid set of headers as defined in the CORS protocol [44]. Failure to correctly implement either end of the communication will lead to these resources not being rendered to the <canvas>. Future research should develop testing approaches for the <canvas> that detect incorrect usage of CORS policies with the <canvas>.

Many bugs that appear to affect the <canvas> are bugs that are Browser runtime errors. As shown in Table 4, many bug reports in the open source projects discussed <canvas> bugs that were not actually due to problems with <canvas>, but instead runtime errors that consequently affected <canvas> functionality. When debugging web applications that use the <canvas>, it would be useful to be able to automatically identify bugs that are Browser runtime errors, so that developers do not waste time trying to locate <canvas> bugs that originate in a different part of the web application. Therefore, future research should aim to save developer time and effort by developing testing tools for the <canvas> that help decide whether a bug originated in or outside of the <canvas>.

#### 7.3 Detecting browser bugs

Some browsers may provide incorrect <canvas> functionality. As can be seen in Table 4, a fairly frequent type of <canvas> bugs is Different behaviour across browsers. However, widely-used cross-browser testing frameworks rely on analyzing the DOM, meaning new forms of cross-browser testing are required for the <canvas>. Future research should investigate cross-browser testing methods for the <canvas>.

Browser-provided JavaScript methods for saving <canvas> data may not always provide the expected result. As can be seen in Table 4, one of our newly defined

---

**Table 4:** Summary of findings with reference to the research question defined in the introduction section of our paper: What types of bugs do developers encounter when creating web applications with the HTML5 <canvas>?
types of `<canvas>` bug is `Saving <canvas> data`. Although there are JavaScript methods built into browsers for saving the `<canvas>` bitmap to a file (such as `toDataURL` and `getImageData`), we found one bug report where a built-in browser method produced a different result than what was rendered on the web page. This means that even if such methods usually work, we may not be able to fully rely on them to know what was rendered to a specific frame on the `<canvas>`. This finding has particular relevance for the development of `<canvas>` testing tools that rely on visual analysis, which would require high confidence in the input data. Further research should seek to understand why browser-provided JavaScript methods for saving the `<canvas>` bitmap can provide unexpected results.

8 Threats to Validity

Internal validity: Our project selection relied on the direct mention of `<canvas>`-related keywords in each GitHub project’s title, description, topic tags, or readme file. While this provided a wide coverage of search results that had to be manually filtered for false positives, we may still have missed some `<canvas>` projects, producing a possible bias in our results. We mitigated this threat mostly by including a wide range of `<canvas>` projects and using stratified sampling to reduce bias towards specific projects. However, further studies may be required to confirm that our results generalize to all open source `<canvas>` projects.

We used a custom set of keywords to identify bug reports in the set of unlabelled GitHub issue reports, which poses a threat to internal validity. While we may have missed some `<canvas>` bug reports, our set of keywords reflects terms often found in GitHub issue reports that report a bug. More research is required to better automatically identify bug reports in a set of GitHub issue reports.

Potential subjectivity in manually classifying bug reports is a threat to internal validity. While automatic classification could have reduced such subjectivity, manual classification was required to discover new types of bugs. Additionally, we did not perform the second stage of our classification process independently, as this was necessary to perform card sorting and ensure we elicited all relevant labels for our taxonomy of `<canvas>` bugs. We addressed this threat first by selecting a baseline set of bug types from empirically validated taxonomies that are related to known characteristics of the `<canvas>`. Then, the first two authors initially independently classified the bug reports, which helps reduce bias and mitigate the potential subjectivity. Our independent classification had a reasonably high overall agreement of 86% (with the lowest agreement for a category being 75% and the highest 94%). We then leveraged the results of our independent classification to perform card sorting with three authors. Having more than one author participate in card sorting also helped reduce potential subjectivity. Finally, we performed a validation exercise as described in Section 4.4 of our paper, in which we performed multi-label classification of 25 `<canvas>` bug reports from a proprietary `<canvas>` application, and had very high agreement of 98% (with the lowest agreement for a category being 88% and the highest 100%). Also, because we used a statistically representative random sample during classification, our results should be valid for the full set of 2,403 collected bug reports.

A threat to internal validity is our choice of agreement measure in our independent multi-label classification. We calculated the percentage of agreement on a per label basis, and then averaged the agreement measures to yield our overall agreement score, which may overestimate agreement as we did not account for coder bias [33]. Although the use of Krippendorf’s alpha and variations of Cohen’s kappa (which account for coder bias) have been proposed to calculate agreement for multi-label classifications [2, 33], it is difficult to compare these agreement scores across different manual classification exercises [7]. For example, differences between choice of distance measure in Krippendorf’s alpha can lead to greater variability in agreement scores than coder bias alone in multi-label classifications [7], and there is no consensus on which distance measure to use [7]. To complicate matters further, there is a lack of consensus on how to interpret the agreement values provided by Cohen’s kappa and Krippendorf’s alpha [7]. Therefore, while our choice of agreement measure may be subject to overestimation, we avoid the variability that would result from choices made within the chosen agreement measure, and it is simple to interpret our overall agreement value as how often both coders assigned the same label to a sample on average (across all assigned labels).

External validity: Our findings are only valid for open source `<canvas>` projects on GitHub, and may not generalize to other `<canvas>` projects. Further studies should validate whether our results generalize to other `<canvas>` projects.

To automatically identify `<canvas>` bug reports in our set of collected `<canvas>` projects, we manually filtered out projects that utilized a canvas object that was not related to HTML5. Examples of such projects include desktop Node.js applications that require node-canvas, or web applications that use react-canvas to render React components to `<canvas>` instead of the DOM. Further research would be required to validate that our findings generalize to projects that use non-HTML5 canvas objects.

9 Conclusion

In this paper, we study the types of bugs that developers face when using the HTML5 `<canvas>` in their applications. We collected 2,403 `<canvas>` bug reports from 123 open-source GitHub projects that use the `<canvas>`. We selected a baseline taxonomy based on known aspects of the `<canvas>`, which also provided a set of initial bug types during our manual analysis. We conducted an iterative manual classification process using a random sample of 332 bug reports. Through our study, we empirically constructed a taxonomy of `<canvas>` bugs that provides the basis for future research on `<canvas>` testing. Our study also provided several insights for developers when working with the `<canvas>`:

- **Visual** bugs are the most frequently encountered `<canvas>` bugs in the studied bug reports.
• Often, bugs which appear to be caused by `<canvas>` are in fact due to issues in other parts of the web application.
• Browsers do not necessarily provide the same `<canvas>` functionality, and this is particularly true for outdated browsers.
• Use of the `<canvas>` brings specific performance concerns in web applications.
• JavaScript methods built into browsers for saving `<canvas>` data may not always provide an expected result.
• Cross-origin resource sharing policies must be correctly used to load `<canvas>` resources from foreign domains.

ACKNOWLEDGMENTS

The research reported in this article has been supported by Prodigy Education and the Natural Sciences and Engineering Research Council of Canada under the Alliance Grant project ALLRP 550309.

REFERENCES

Works Cited

[1] Adobe. (Jul. 25, 2017). “Flash & the future of interactive content,” [Online]. Available: https://blog.adobe.com/en/publish/2017/07/25/adobe-flash-update.html (visited on 09/29/2021).

[2] ——. (Jan. 13, 2021). “Adobe flash player EOL general information page,” [Online]. Available: https://www.adobe.com/products/flashplayer/end-of-life.html (visited on 09/29/2021).

[3] E. Alégroth, K. Feidt, and P. Koistirö, “Maintenance of automated test suites in industry: An empirical study on visual gui testing,” Information and Software Technology, vol. 73, pp. 66–80, 2016.

[4] J. Anvik, L. Hiew, and G. C. Murphy, “Coping with an open bug repository,” in Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange, 2005, pp. 35–39.

[5] ——, “Who should fix this bug?” In Proceedings of the 28th international conference on Software engineering, 2006, pp. 361–370.

[6] J. Anvik and G. C. Murphy, “Determining implementation expertise from bug reports,” in Fourth International Workshop on Mining Software Repositories (MSR '07: ICSE Workshops 2007), IEEE, 2007, pp. 2–2.

[7] R. Artstein and M. Poesio, “Inter-coder agreement for computational linguistics,” Computational linguistics, vol. 34, no. 4, pp. 555–596, 2008.

[8] K. Bajaj, K. Pattabiraman, and A. Mesbah, “Mining questions asked by web developers,” in Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 112–121.

[9] M. Bajammal and A. Mesbah, “Web canvas testing through visual inference,” in 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2018, pp. 193–203.

[10] M. Bajammal, A. Stocco, D. Mazinian, and A. Mesbah, “A survey on the use of computer vision to improve software engineering tasks,” IEEE Transactions on Software Engineering, 2020.

[11] K. Basques. (Apr. 6, 2017). “Analyze runtime performance,” [Online]. Available: https://developer.chrome.com/docs/devtools/evaluate-performance/ (visited on 08/30/2022).

[12] J. Bogner and M. Merker, “To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github,” arXiv preprint arXiv:2203.11115, 2022.

[13] P. Breton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, “Lessons from applying the systematic literature review process within the software engineering domain,” Journal of systems and software, vol. 80, no. 4, pp. 571–583, 2007.

[14] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, “Not all bugs are the same: Understanding, characterizing, and classifying bug types,” Journal of Systems and Software, vol. 152, pp. 165–181, 2019.

[15] S. Doğan, A. Betin-Can, and V. Garousi, “Web application testing: A systematic literature review,” Journal of Systems and Software, vol. 91, pp. 174–201, 2014.

[16] R. C. Hoetzlein, “Graphics performance in rich internet applications,” IEEE computer graphics and applications, vol. 32, no. 5, pp. 98–104, 2012.

[17] A. Issa, J. Sillito, and V. Garousi, “Visual testing of graphical user interfaces: An exploratory study towards systematic definitions and approaches,” in 2012 14th IEEE International Symposium on Web Systems Evolution (WSE), IEEE, 2012, pp. 11–15.

[18] A. V. Kamierski, L. Faléchon, C.-P. Bezemer, and A. Hindle, “Pyssubs: Characterizing single-statement bugs in popular open-source python projects,” in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 520–524.

[19] R.-M. Karampatsis and C. Sutton, “How often do single-statement bugs occur? the manyssubs-4j dataset,” in Proceedings of the 17th International Conference on Mining Software Repositories, (2020), pp. 573–577.

[20] D. Landman, A. Serebrenik, and J. J. Vinju, “Challenges for static analysis of java reflection-literature review and empirical study,” in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 507–518.

[21] P. Leitner and C.-P. Bezemer, “An exploratory study of the state of practice of performance testing in java-based open source projects,” in Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, 2017, pp. 373–384.

[22] V. Lelli, A. Blouin, and B. Baudry, “Classifying and qualifying gui defects,” in 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2015, pp. 1–10.

[23] A. Marchetto, F. Ricca, and P. Tonella, “An empirical validation of a web fault taxonomy and its usage for web testing,” Journal of Web Engineering, pp. 316–345, 2009.

[24] R. M. de Mello, P. C. Da Silva, and G. H. travassos, “Investigating probabilistic sampling approaches for large-scale surveys in software engineering,” Journal of Software Engineering Research and Development, vol. 3, no. 1, pp. 1–26, 2015.

[25] K. Moot. (Jul. 23, 2012). “Handling user input in html5 canvas-based games,” [Online]. Available: https://developer.ibm.com/tutorials/wa-games/?mhsrc=ibmsearch_a&men=induction (visited on 09/29/2021).

[26] Mozilla. (Sep. 15, 2021). “Allowing cross-origin use of images and canvas,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/CORS_enabled_image (visited on 09/29/2021).

[27] ——. (Sep. 20, 2021). “Canvas: The graphics canvas element,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Canvas/Element/canvas (visited on 09/29/2021).

[28] ——. (Sep. 14, 2021). “CanvasRenderingcontext2d.arc,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/arch (visited on 09/29/2021).

[29] ——. (Sep. 5, 2021). “Cross-origin resource sharing (CORS),” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/DocumentObject_Model/Introduction (visited on 09/29/2021).

[30] N. Nurmuliani, D. Zowghi, and S. P. Williams, “Using card sorting technique to classify requirements change,” in Proceedings. 12th IEEE International Requirements Engineering Conference, 2004., IEEE, 2004, pp. 240–248.

[31] F. S. Ocarriza, K. Bajaj, K. Pattabiraman, and A. Mesbah, “A study of causes and consequences of client-side javascript bugs,” IEEE Transactions on Software Engineering, vol. 43, no. 2, pp. 128–144, 2016.

[32] ——. (Sep. 14, 2021). “Introduction to the DOM,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/DocumentObject_Model/Introduction (visited on 09/29/2021).

[33] R. Passonneau, “Measuring agreement on set-valued items (masi) for semantic and pragmatic annotation,” 2006.

[34] P. Ralph, “Toward methodological guidelines for process theories and taxonomies in software engineering,” IEEE Transactions on Software Engineering, vol. 45, no. 7, pp. 712–735, 2018.

[35] F. Ricca and A. Stocco, “Web test automation: Insights from the grey literature,” in International Conference on Current Trends in Theory and Practice of Informatics, Springer, 2021, pp. 472–485.
[85] Vincenttbk, *Halo does not disappear on hiding series (mobile)*, Oct. 23, 2014. [Online]. Available: https://github.com/highcharts/highcharts/issues/3533 (visited on 09/29/2021).

[86] vincenttheeten, *Cannot get font() to work*, Nov. 9, 2012. [Online]. Available: https://github.com/jasondavies/d3-cloud/issues/3 (visited on 09/29/2021).

[87] williamngan, *Canvasform.font doesn’t set font*, Nov. 14, 2018. [Online]. Available: https://github.com/williamngan/pts/issues/32 (visited on 09/29/2021).