Experimental Web Applications for Multidimensional Visual Analysis

Project Report

Ožbej Golob

706.424 Seminar/Project Interactive and Visual Information Systems SS 2023
Graz University of Technology

27 May 2023

Abstract

As part of a future Master's thesis, it is intended to build a web application for Multidimensional Visual Analysis (MVA), which will allow analysts to explore large, multidimensional datasets using a number of linked views including scatterplot matrix, scatterplot, similarity map, and table view.

Within this project, various web technologies were explored and tried out, as a basis for making an informed choice of the technology stack to use in the planned Master's thesis. In particular, four frontend frameworks (Angular, React, Vue, and Svelte) and four built-in web rendering technologies (SVG-DOM, Canvas2D, WebGL, and WebGPU) were looked at, as well as the use of Offscreen Canvas. Furthermore, six web graphics libraries (Konva.js, Two.js, Pixi.js, Three.js, Babylon.js, and D3.js) were tried out and compared. Since it would be desirable to also build installable packages for the three common desktop platforms (Windows, MacOS, and Linux), the desktop development libraries Electron.js and Tauri were looked at.

Finally, three experimental web applications were developed using different technologies and frameworks to investigate performance and quality issues when drawing large numbers of records in a parallel coordinates visualization.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

This project is a step towards the larger goal of building a web application for Multidimensional Visual Analysis (MVA). MVA focuses on the use of visual representations to explore and analyze multidimensional data sets. The web final application is intended to allow analysts to explore large, multidimensional datasets using a number of linked views including scatterplot matrix, scatterplot, similarity map, and table view. A mock-up of the user interface for the final MVA application is shown in Figure 1.1.

The most important requirements for the future web application are:

- *Web Application*: The application should be a web application. Web applications offer cross-platform compatibility, straightforward development, updates, and maintenance, good web accessibility, and easy distribution.
- *Large Amount of Data*: The application should be able to cope with and render a large amount of data. Users are often analyzing large and complex datasets with up to 10,000 records and several dozen dimensions. The application should be able to handle that amount of data.
- *Desktop Application*: The application should also be available as a native desktop application. Despite the aforementioned benefits to web applications, desktop applications often provide better performance, better user experience, and better offline functionality.

Within this project, various web technologies were explored and tried out, as a basis for making an informed choice of technology stack to use in the planned Master's thesis. In particular, the following technologies were examined:

- *Frontend Development Frameworks*: Angular, React, Vue, and Svelte. A frontend development framework is a collection of pre-written code and tools that developers can use to build the user interface and functionality of a software application. They are typically written in JavaScript (or TypeScript).
- *Built-In Web Rendering Technologies*: SVG-DOM, Canvas2D, WebGL, and WebGPU, as well as the use of Offscreen Canvas. SVG-DOM and Canvas2D produce simple 2d graphics. WebGL and its successor WebGPU produce GPU-accelerated 2d and 3d graphics.
- *Web Graphics Libraries*: Konva.js, Two.js, Pixi.js, Three.js, Babylon.js, and D3.js. These libraries build on top of one or more of the built-in web rendering technologies.
- *Desktop Development Libraries*: Electron.js and Tauri. These allow a desktop executable package to be built for Windows, MacOS, and Linux.

Three experimental web applications were developed to gain experience with different technologies and frameworks and to investigate performance and quality issues when drawing large numbers of records in a parallel coordinates visualization:

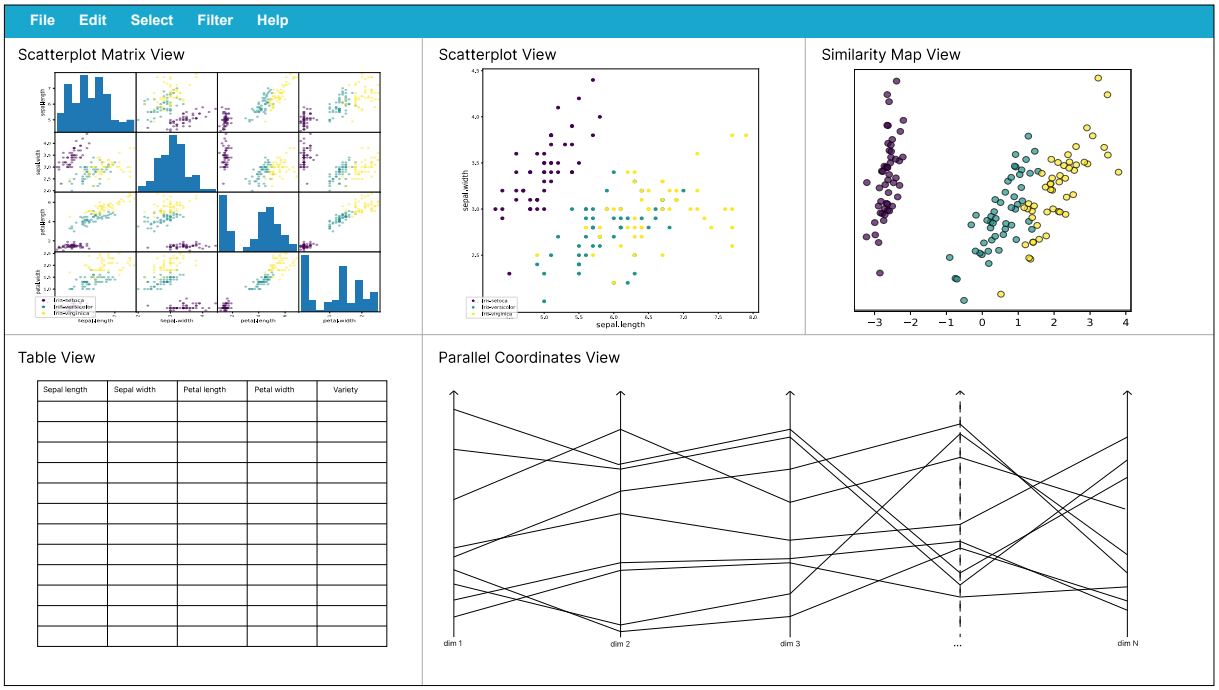

Figure 1.1: Mock-up of the user interface of the intended future MVA application. [Drawn by Ožbej Golob using Figma.]

- Angular with Canvas2D.
- React with Konva, Babylon, and Pixi.
- Plain JavaScript with Pixi.

Finally, a web technology stack was decided upon for the implementation of the future MVA web application: namely Svelte with a combination of Pixi.js, D3.js, and Offscreen Canvas.

Chapter 2

Web Applications

A web application is an application program that is stored on a remote server, accessed over the internet, and does not need to be downloaded or installed. Users can access web apps through a web browser, such as Google Chrome, Mozilla Firefox, or Safari. Web applications offer several benefits, such as multiple users accessing the same version of an application, no need to install the app, access through various platforms like desktop, laptop, or mobile, and access through multiple browsers. Additionally, web applications often have shorter development cycles and smaller development teams, making them more cost-effective and efficient to build and maintain.

Web applications usually follow the client-server model, where the client is considered the frontend and the server is considered the backend [Berson 1996]. This is illustrated in Figure 2.1.

2.1 Frontend

The frontend of a web application is responsible for the user interface and user experience [Ahmed 2023b]. It is the part of the application which the user interacts with directly, and its purpose is to provide a visually appealing and intuitive interface that allows users to easily navigate and perform tasks. A well-designed frontend is crucial for the success of an application, as it can greatly influence user satisfaction and adoption rates. Frontend development involves the use of programming languages such as HTML, CSS, and JavaScript to create the visual components of an application, including buttons, menus, forms, and other graphical elements. It is important to consider factors such as accessibility, usability, and responsiveness when designing a frontend, to ensure that the application can be used by a diverse range of users on a variety of devices and platforms.

2.2 Backend

The backend of a web application is responsible for processing data and executing the logic of an application [Ahmed 2023a]. It provides the underlying functionality that supports the frontend, enabling it to interact with databases, servers, and other systems. Backend development involves the use of programming languages such as Java, Python, Ruby, and PHP to create the logic and functionality that drives an application. The backend also handles security and user authentication, ensuring that user data is protected from unauthorized access. Scalability and performance are key considerations when designing a backend, as it must be able to handle large amounts of data and support a large number of users. A well-designed backend is crucial for the success of an application, as it directly affects the reliability, performance, and security of the application.

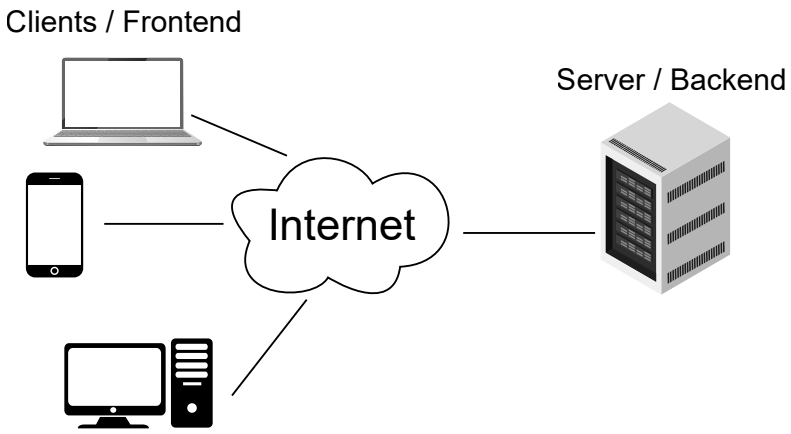

Figure 2.1: The client-server model. [Drawn by Ožbej Golob using Adobe Illustrator.]

Chapter 3

Frontend Development Frameworks

A frontend development framework is a collection of pre-written code and tools that developers can use to build the user interface and functionality of a software application. These frameworks are typically built using programming languages such as JavaScript, and provide a set of pre-designed components and functions that developers can use to create an application's frontend quickly and efficiently.

Frontend development frameworks can include a wide range of tools, such as user interface components like buttons, forms, and menus, as well as more complex features like data visualization, animations, and dynamic content loading. This chapter reviews a number of popular frontend development frameworks.

3.1 Angular

Angular is an open-source framework for building dynamic, single-page web applications using HTML, CSS, and JavaScript or TypeScript [Google 2016; Murray et al. 2018]. It was developed and is maintained by Google.

Some of the pros of using Angular include its comprehensive set of tools and features, which can help developers build complex web applications quickly and efficiently. Angular also provides a strong focus on code organization and architecture, making it easier to maintain and update applications over time. Additionally, Angular offers a large community of developers and resources, making it easier to find support and learn new skills.

However, some of the cons of using Angular include a steep learning curve, as it requires knowledge of multiple programming languages and concepts. Additionally, Angular can be more verbose than other frameworks, which can lead to larger codebases and slower performance. Finally, Angular can be more difficult to integrate with other technologies and frameworks, as it has a unique architecture and approach to web development.

3.2 React

React is an open-source framework for building web and mobile applications using JSX (a syntax extension for JavaScript), CSS, and JavaScript or TypeScript [Meta 2013; Accomazzo et al. 2017]. It was developed and is maintained by Meta (formerly Facebook).

Some of the pros of using React include its simplicity and flexibility, which make it easy to learn and use, and its focus on reusability, which can save developers time and effort. React also offers a virtual Document Object Model (DOM), which can improve application performance by reducing the number of updates needed to the actual DOM. Additionally, React has a large and active community of developers, with many resources and tools available to help developers learn and use the framework effectively. React is also highly customizable, allowing developers to use it in conjunction with other libraries and frameworks to build complex, scalable applications.

However, some of the cons of using React include the lack of built-in functionality, which can require developers to use additional libraries or tools to add functionality to their applications. React can also have a steep learning curve. Additionally, React's reliance on JSX can be challenging for some developers to learn and use effectively.

3.3 Vue

Vue is an open-source framework for building web applications using HTML, CSS, and JavaScript or TypeScript [You 2014; Filipova 2016]. It was developed by Evan You and is maintained by Vue core team members.

Some of the pros of using Vue include its simplicity, which makes it easy to learn and use for developers of all skill levels, and its flexibility, which allows developers to use it with other libraries and frameworks to build complex, scalable applications. Vue also offers a range of features, including reactive data binding, computed properties, and built-in directives, that can help developers build applications quickly and efficiently. Additionally, Vue has a growing and active community of developers, with many resources and tools available to help developers learn and use the framework effectively. Vue also has a relatively small learning curve compared to other popular frameworks, such as React and Angular.

However, some of the cons of using Vue include its smaller community and ecosystem compared to other frameworks, which can make it more difficult to find support and resources. Vue also has fewer third-party plugins and tools available compared to other frameworks, which can limit its functionality in some cases.

3.4 Svelte

Svelte is an open-source framework for building web applications using HTML, CSS, and JavaScript [Harris 2016; Holthausen 2022]. It was developed by Rich Harris and is maintained by Svelte core team members. It takes a different approach to building web applications compared to other popular frameworks, by moving much of the work of generating code from the browser to the build process.

One of the primary benefits of Svelte is its small file size and fast performance. Svelte compiles the application code into highly optimized, plain JavaScript code that runs more efficiently in the browser, resulting in faster load times and improved performance. Another advantage of Svelte is its simplicity and ease of use. The framework has a small API surface area, and its syntax is intuitive and easy to learn. Svelte also has a rich ecosystem of plugins and tools that make it easy to integrate with other technologies.

However, one of the disadvantages of Svelte is its relatively new status, which means that it has a smaller community and fewer resources compared to more established frameworks like Angular, React, and Vue. This can make it more difficult to find support and resources for developers who are just starting with Svelte. Another disadvantage of Svelte is its limited compatibility with older browsers. Since Svelte relies on newer web technologies like JavaScript modules and the CSS Grid Layout, it may not work well on older browsers that do not support these features.

Chapter 4

Web Graphics Rendering Technologies

Web graphics rendering technologies are used to create and display graphics, images, and multimedia content in the web browser. This chapter reviews popular web graphics rendering technologies and libraries and compares their performance.

4.1 Built-In Web Rendering Technologies

Modern web browsers support a number of ways to draw graphics, which are built into the web browser. These built-in web graphics rendering technologies can be split into two categories:

- *Simple 2D Graphics*: Either vector graphics with SVG nodes injected into the DOM using JavaScript (SVG-DOM), or raster-based graphics created with the HTML `<canvas>` element and `2d` context (Canvas2D).
- *GPU-Enabled 2D and 3D Graphics*: WebGL and its successor WebGPU, which are based on constructing and manipulating a scene graph.

4.1.1 SVG-DOM

Scalable Vector Graphics (SVG) is a standard markup language for 2d vector graphics [W3C 2001]. Based on XML, it contains elements such as lines, rectangles, circles, polygons, and text, out of which more complex graphics can be constructed. SVG graphics are scalable, which means that they can be resized without losing quality. SVG graphcs are also editable, which means that they can be easily modified or updated. Modern browsers support SVG 1.1 and some elements of SVG 2.0. This means that SVG elements can be intermingled with HTML elements in a web document, to provide embedded vector graphics. The SVG elements can be styled with CSS.

JavaScript can be used to dynamically insert SVG nodes into the browser's Document Object Model (DOM) and to later manipulate or remove them [Whitney 2013]. Thus, data-driven vector graphics can be created on-the-fly with JavaScript. This technique will be referred to as SVG-DOM.

Performance issues can arise when rendering large datasets with SVG-DOM, since each data point or shape requires its own SVG element be inserted into the DOM.

4.1.2 Canvas2D

Raster-based 2d graphics can be created from JavaScript using the Canvas API and the HTML `<canvas>` element with a `2d` rendering context [MDN 2023]. This will be referred to as Canvas2D. Canvas2D is used to create dynamic and interactive graphics on the web. Canvas2D graphics are created using a raster-based approach, where pixels are directly manipulated to create the graphics.

Canvas2D offers improved performance compared to SVG-DOM and can handle larger datasets and more complex animations. Canvas2D graphics are highly dynamic and interactive, which means that

they can be manipulated and animated in real-time using JavaScript. However, they are not naturally scalable or responsive and have to be redrawn if the dimensions change. Canvas2D is supported by all modern browsers, making it a reliable and cross-browser compatible graphics technology.

4.1.3 WebGL

WebGL [Khronos Group 2011] is a 3d web graphics rendering technology used to create high-performance graphics on the web. WebGL leverages the hardware of the Graphics Processing Unit (GPU) of a device to render graphics in real-time, resulting in visually stunning and highly performant graphics. Graphics are drawn by constructing a scene graph through the WebGL API. WebGL also enables users to manipulate and interact with the graphics in real-time. Additionally, WebGL is supported by all modern browsers, making it a reliable and cross-browser compatible graphics technology.

However, WebGL has a steep learning curve that requires a high level of expertise and knowledge of both JavaScript and computer graphics. There exist a number of WebGL libraries that simplify the development of web applications with WebGL, some of which are described in Section 4.2.

4.1.4 WebGPU

WebGPU [W3C 2021] is the successor to WebGL. It is designed to be closer to the architecture of modern GPUs, and is similar in style to low-level graphics APIs like Vulkan, DirectX 12, and Metal. WebGPU aims to improve performance and efficiency by allowing developers to write code that can take advantage of the latest hardware features, while still maintaining compatibility with a wide range of devices and browsers. It provides a unified and standardized interface for accessing GPU resources, including rendering, computing, and data transfer operations.

WebGPU is still in the early stages of development and is not yet supported by all major web browsers [Deveria 2023], but it has the potential to revolutionize web-based graphics and computing in the future.

4.1.5 Offscreen Canvas

The Offscreen Canvas API [MDN 2018] is a relatively new feature in web development that allows for the creation of a canvas rendering context that can be rendered in a separate thread from the main JavaScript thread. In traditional web development, all canvas rendering occurs on the main thread, which can cause performance issues if the canvas is rendering complex graphics or animations. The Offscreen Canvas API helps alleviate this issue by offloading the rendering process to a separate thread, thus freeing up the main thread for other tasks.

Using the Offscreen Canvas API, developers can create and manipulate a canvas rendering context in a separate thread, while still being able to transfer the rendered output to the main thread for display on the web page. This can result in faster rendering times, smoother animations, and a more responsive user experience.

The Offscreen Canvas API is supported by all modern web browsers. Canvas2D, WebGL, and WebGPU can all make use of and benefit from Offscreen Canvas. However, since SVG-DOM is rendered via the DOM rather than a canvas, it cannot benefit from Offscreen Canvas.

4.2 Web Graphics Libraries

Web graphics rendering libraries have become increasingly popular in recent years, as they enable developers to create visually appealing and interactive graphics on web pages. These libraries provide pre-built functions and tools for creating and manipulating 2d and 3d graphics, animations, and visualizations. They offer a powerful and flexible platform for creating dynamic and responsive web graphics that can be used for a wide range of applications.

4.2.1 Konva.js

Konva.js [Lavrenov 2012] is a 2d drawing and animation library which builds upon Canvas2D. In effect, it extends the utility of Canvas2D with many extra features. With Konva.js, developers can easily create and manipulate complex scenes composed of various shapes, images, text, and animations.

4.2.2 Two.js

Two.js [Brandel 2014] is a 2d drawing library for the web that provides an intuitive and easy-to-use API for creating complex vector graphics, animations, and interactive visualizations. It was originally spun off from the Three.js project, to focus solely on 2d graphics. Two.js supports WebGL, Canvas2D, and SVG-DOM web rendering technologies. It has a wide range of shapes and primitives, and allows developers to apply various styles and effects to them, such as gradients, shadows, and opacity. It also supports animation and interactivity, with features like tweening and easing functions, mouse and touch events, and drag-and-drop functionality. Two.js is highly performant thanks to its use of WebGL for rendering graphics and has a modular architecture that allows developers to use only the parts of the library that they need. It has a large community of contributors and users.

4.2.3 Pixi.js

Pixi.js [Groves 2013] is a powerful 2d drawing library for the web, which provides a fast and efficient way to create high-performance interactive applications and games. Pixi.js supports WebGL and Canvas2D web rendering technologies, as well as Offscreen Canvas. Pixi.js provides an easy-to-use API that allows developers to create complex graphics and animations with ease. Pixi.js supports a wide range of features, including sprite sheets, filters, masking, and particle systems. It also has built-in support for animations and interactivity, including keyboard and mouse events, touch events, and physics simulations. Pixi.js is highly optimized for performance, which makes it suitable for creating complex graphics and animations that run smoothly even on mobile devices. It has a large and active community of contributors and users who continue to improve and extend its functionality.

4.2.4 Three.js

Three.js [Cabello 2010] is a 3d drawing library for generating graphics and animations on the web. Three.js supports WebGL, WebGPU, Canvas2D, and SVG-DOM web rendering technologies, as well as Offscreen Canvas. Three.js provides an easy-to-use API for building complex 3d scenes, which can include a wide variety of objects, materials, lights, and effects. Three.js also includes support for advanced materials, physics simulations, and VR. With a large and active community of contributors, Three.js is constantly evolving and improving, making it one of the most popular and widely used 3D graphics libraries for the web.

4.2.5 Babylon.js

Babylon.js [Catuhe 2013] is a 3d drawing library for generating graphics and animations on the web, similar to Three.js. Babylon.js supports WebGL and WebGPU web rendering technologies, as well as Offscreen Canvas. Babylon.js provides an easy-to-use API for building complex 3d scenes, which can include a wide variety of objects, materials, lights, and effects. With a large and active community of contributors, Babylon.js is constantly evolving and improving, making it one of the most popular and widely used 3d graphics libraries for the web.

4.2.6 D3.js

D3.js [Bostock et al. 2011] is a lower-level 2d drawing library for manipulating and visualizing data using web technologies, based on SVG-DOM rendering. D3.js allows developers to create dynamic, interactive, and customizable data visualizations by composing a series of marks. It includes a powerful

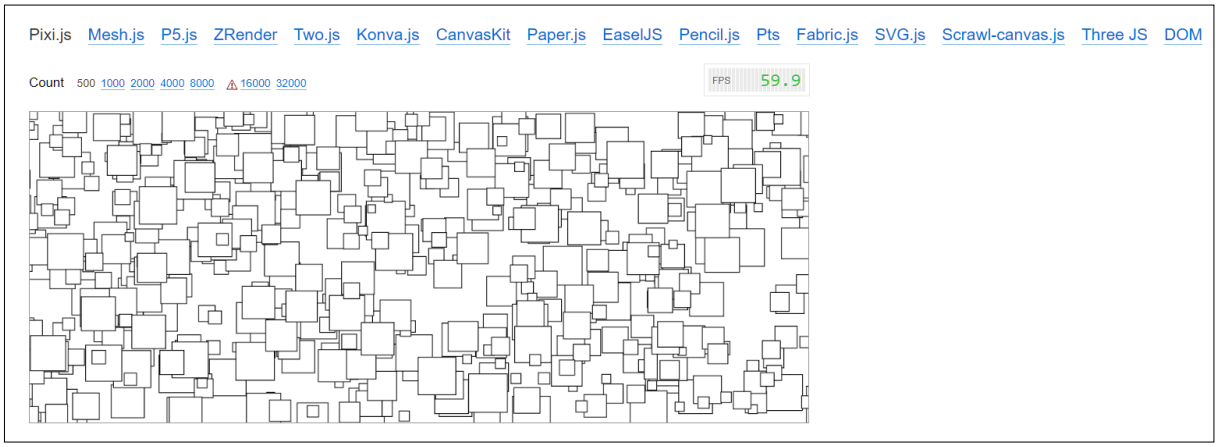

Figure 4.1: The Slay Lines benchmark web application displaying 500 rectangles with Pixi.js.

set of examples and recipes for visualizations such as bar charts, scatterplots, line charts, and more. In addition, D3.js provides a wide range of utilities for working with data, including data binding, selection, filtering, transformation, reading CSV files, and creating scales. These tools can be combined and customized to create complex interactive visualizations that are tailored to specific data sets and use cases.

One of the key strengths of D3.js is its flexibility. It does not impose a specific visualization style or framework, but rather provides a set of building blocks that can be combined in creative ways. This makes it suitable for a wide range of applications, from simple charts and graphs to complex data-driven applications.

4.3 Comparison

The company Slay Lines created a benchmark web application, which compares a number of popular web graphics rendering libraries [Slay Lines 2020]. The application consists of up to 32,000 different rectangles moving on a canvas at various speeds. Figure 4.1 shows the Slay Lines application displaying 500 rectangles with Pixi.js. The Slay Lines web application was used to compare the following graphics web rendering libraries: SVG.js [Fierens 2012] (which uses SVG-DOM web rendering technology), Konva.js [Lavrenov 2012] (which uses Canvas2D web rendering technology), Two.js [Brandel 2014], Pixi.js [Groves 2013], and Three.js [Cabello 2010] (which all use WebGL web rendering technology).

The aforementioned libraries were benchmarked by their frame rate in frames per second (FPS). FPS was measured as an average over ten seconds. A custom Python script, shown in Listing 4.1, was written for the purpose of measuring the average FPS. The script uses the Selenium WebDriver [Huggins 2004] to programmatically open a Chrome web browser and navigate to the Slay Lines application. Then, Selenium executes a JavaScript function to measure the average FPS over ten seconds and reports it into the console logs.

The following experimental setup was used:

- Asus Vivobook S510U laptop (with Nvidia GeForce MX150 GPU).
- Google Chrome Version 112.0.5615.50.
- Selenium ChromeDriver 112.0.5615.49.
- Python 3.9.13.

Figure 4.2 shows the resulting performance measurements in terms of average FPS with respect to the number of animated rectangles. Table 4.2 reports the exact measurements.

```
from selenium import webdriver
import time

# Create a new Chrome web driver
options = webdriver.ChromeOptions()
browser = webdriver.Chrome(options=options, executable_path='./chromedriver.exe')

# Navigate to the webpage you want to measure the FPS for
browser.get("https://benchmarks.slaylines.io")

# Wait for the page to load
time.sleep(10)

# Execute JavaScript to measure FPS
browser.execute_script("""
  window.performance.mark('start');
  console.log("Performance measurement started");
  let startTime = performance.now();
  let lastTime = startTime;
  let fps = 0;
  let fpsArray = [];

  function update() {
    let currentTime = performance.now();
    let timeElapsed = currentTime - lastTime;
    lastTime = currentTime;
    fps = 1000 / timeElapsed;
    fpsArray.push(fps);

    // End measurement at 10 seconds
    if (currentTime - startTime > 10000) {
      console.log("Performance measurement finished");
      window.performance.mark('end');
      window.performance.measure('measure', 'start', 'end');
      let sum = fpsArray.reduce((a, b) => a + b);
      let average = sum / fpsArray.length;
      console.log("Average FPS:", average);
      return fpsArray;
    }
    else window.requestAnimationFrame(update);
  }
  window.requestAnimationFrame(update);
""")
```

Listing 4.1: Python code using Selenium WebDriver to measure average FPS.

	500 R	1,000 R	2,000 R	4,000 R	8,000 R	16,000 R	32,000 R
SVG.js	52.2	26.2	13.4	8.3	4.4	1.5	0.7
Konva.js	64.5	57.0	31.6	16.0	9.2	6.9	3.5
Two.js	60.2	70.0	47.2	26.2	13.6	5.1	/
Pixi.js	60.1	60.2	60.3	60.5	56.4	31.8	17.0
Three.js	27.2	14.7	7.6	3.4	1.8	0.6	/

Table 4.1: Comparative performance of web graphics rendering libraries. Columns indicate the number of animated rectangles in the benchmark. Cells show the average FPS of a specific library. A slash (/) indicates that Chrome crashed.

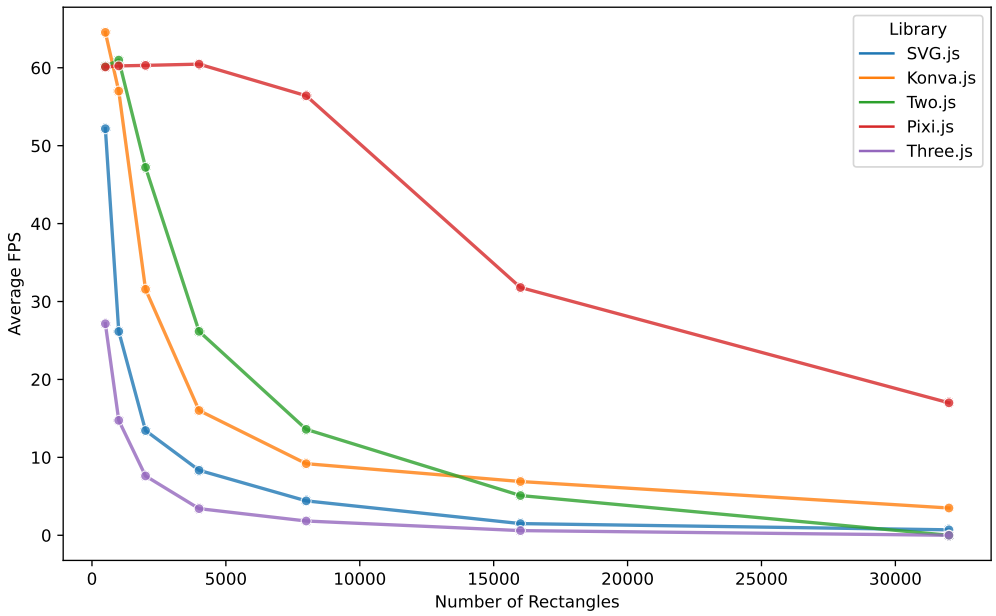

Figure 4.2: Comparative performance of web graphics rendering libraries, measured using the Slay Lines benchmark web application.

Chapter 5

Desktop Development Libraries

Desktop development libraries are software libraries and frameworks that enable developers to create web applications that also run on common desktop operating systems like Windows, macOS, and Linux. These libraries provide developers with tools and APIs to build graphical user interfaces (GUIs), handle user input and output, manage data, interact with system resources, and perform other tasks required for building desktop applications.

The goal is to be able to build the future MVA web application as a desktop application too. This chapter describes two desktop development libraries using HTML, CSS, and JavaScript.

5.1 Electron.js

Electron.js [OpenJS 2013] (also known as Electron) is a popular desktop development library. Electron works by combining two main components: Chromium and Node.js. Chromium is an open-source web browser project that powers Google Chrome and other popular browsers. It provides Electron with the ability to render web content and handle user interactions. The user sees the same UI on Windows, Linux and macOS.

Node.js is a popular JavaScript runtime that allows developers to run JavaScript on the desktop and access native APIs. In essence, Electron allows web applications to be packaged as desktop applications, by bundling Chromium, Node, and the web application itself into a single executable package. This comes at the cost of rather large executable packages, often 100 MB or more.

Electron also provides developers with a set of APIs for building desktop applications, including APIs for handling file system operations, creating menus and dialogs, and managing window events. However, using them means that the application can no longer be used as a web application.

5.2 Tauri

Tauri [Tauri 2019] is a relatively new desktop development library. Tauri works by combining Rust, JavaScript, and the platform's local Webview. Rust is a systems programming language known for its memory safety and performance. Tauri uses the local system's Webview: Edge Webview2 (Chromium) on Windows, WebKitGTK on Linux, and WebKit on macOS, making Tauri apps much lighter than Electron apps. Tauri apps also typically have better performance, launch time, and memory consumption than Electron apps.

Chapter 6

Experimental Implementations

In the scope of the project, three experimental web applications were developed using different technologies and frameworks to draw parallel coordinates and compare performance.

6.1 Angular with Canvas2D

The application developed with Angular served as a starting point. It incorporated a simple parallel coordinates plot developed with Canvas2D, as shown in Figure 6.1. The plot displays 100 records with 20 dimensions. The problem with Angular is that some of the web rendering libraries do not support Angular. Hence, further work on the Angular web application was abandoned and the React application was developed.

6.2 React with Konva, Babylon, and Pixi

The application developed with React incorporated parallel coordinates plots developed with the Konva.js, Babylon.js, and Pixi.js libraries. The application also incorporated an FPS counter component, which was used to compare performance.

Figure 6.2 shows the parallel coordinates plot developed with React and the Konva.js library. The plot displays 100 records with 20 dimensions and supports filtering of the records with draggable triangles. Results showed that the plot can render around 1000 records without a performance drop-off. However, when filtering the records, performance drops significantly after a few hundred records (say 500) to around 10 FPS.

Figure 6.3 shows the parallel coordinates plot developed with React and the Babylon.js library. The plot displays 100 records with 20 dimensions. Results showed that the rendering performance drops significantly after a few hundred records (say 500) to around 3 FPS.

Figure 6.4 shows the parallel coordinates plot developed with React and the Pixi.js library. The plot displays 100 records with 20 dimensions. Results showed that the plot can render around 10,000 records without a performance drop-off. However, the rendering quality of lines and text with Pixi.js left much to be desired: lines are jagged and text is blurry. Additionally, the React implementation of the Pixi.js library is not as well maintained as the plain JavaScript Pixi.js library. For that reason, an experimental plain JavaScript web application was also developed.

6.3 Plain JavaScript with Pixi

The plain JavaScript web application incorporated a parallel coordinates plot with the plain JavaScript Pixi.js library, as shown in Figure 6.5. The results showed promise, as the application is able to render a large amount of data (say 10,000 records) without any performance drop-off and provides good quality rendering of lines.

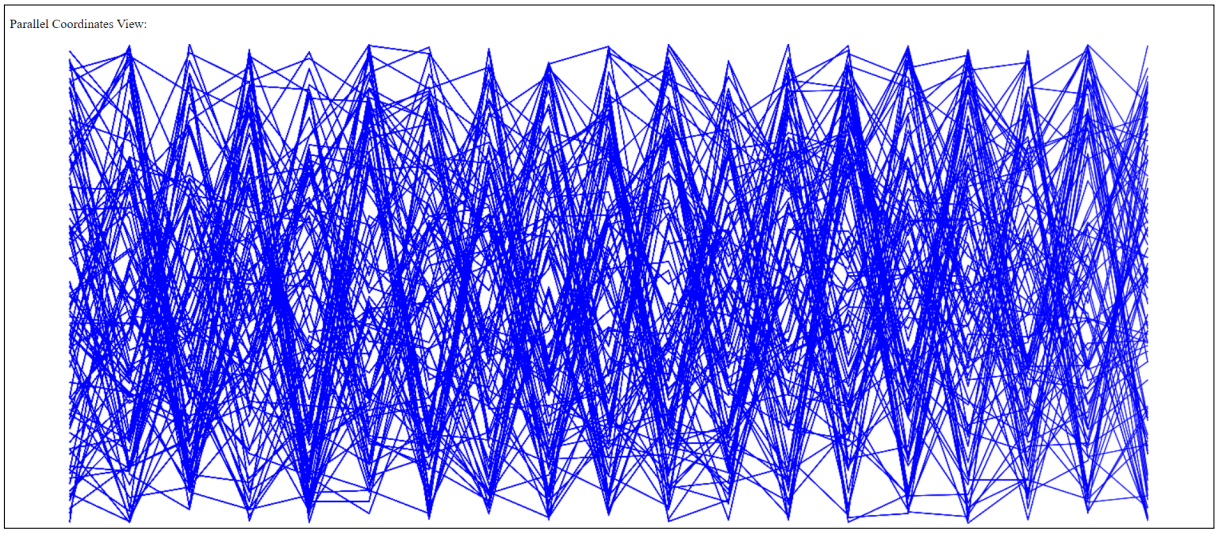

Figure 6.1: Angular and Canvas2D parallel coordinates plot.

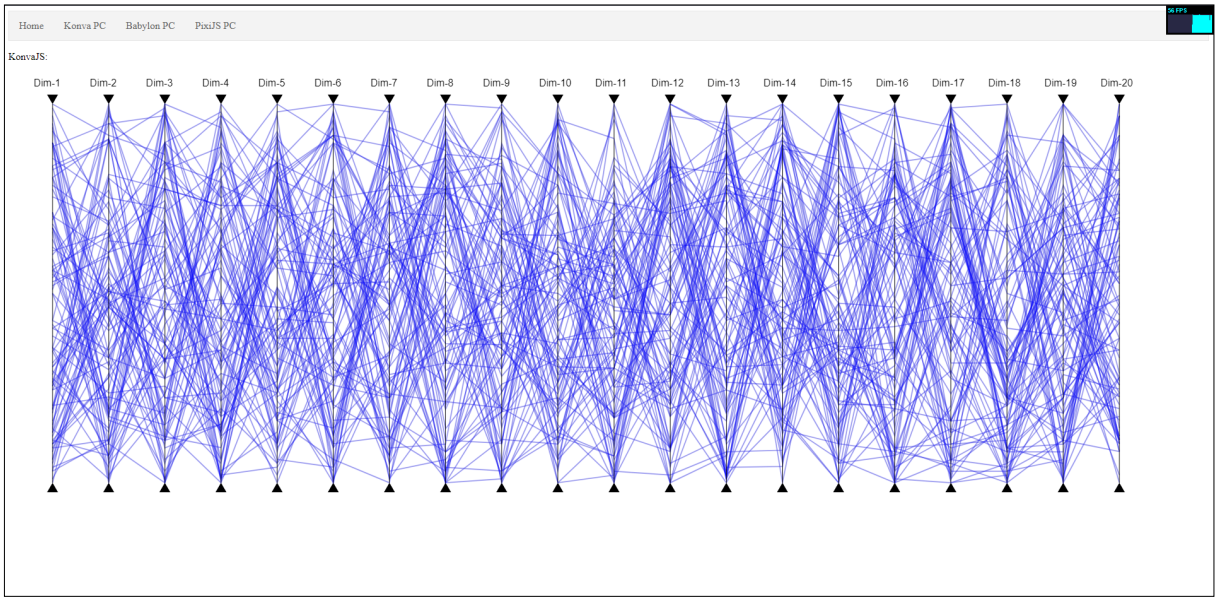

Figure 6.2: React and Konva.js parallel coordinates plot.

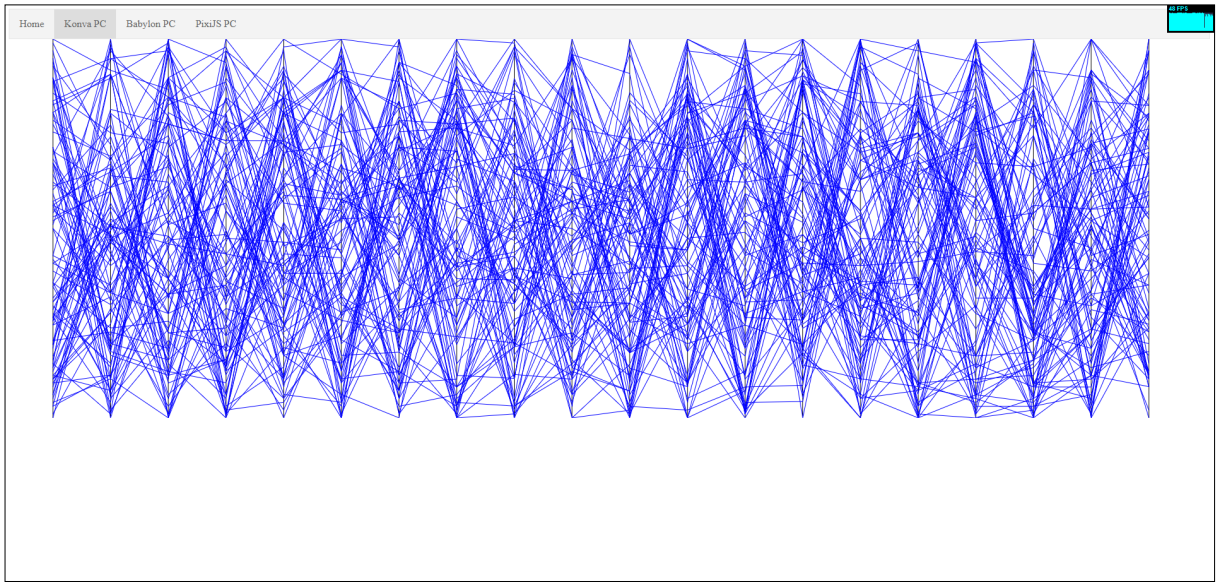

Figure 6.3: React and Babylon.js parallel coordinates plot.

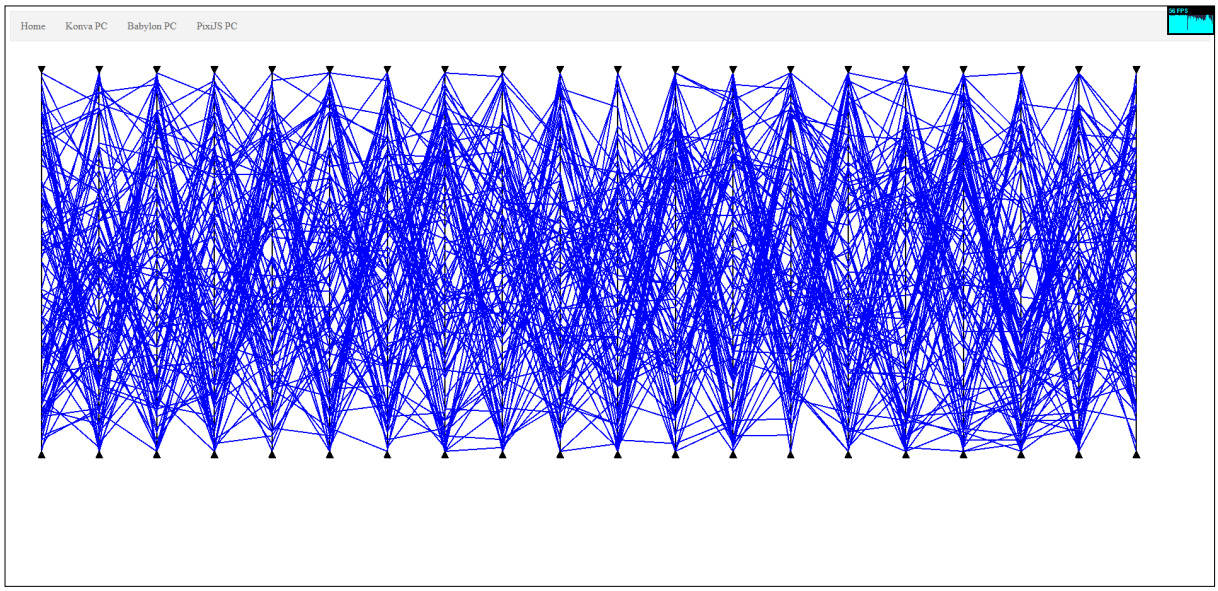

Figure 6.4: React and Pixi.js parallel coordinates plot.

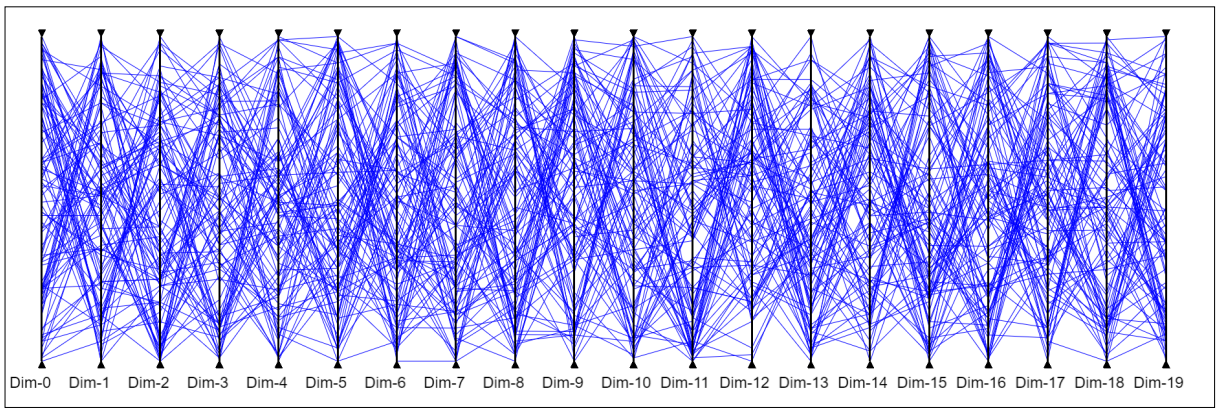

Figure 6.5: Plain JavaScript and Pixi.js parallel coordinates plot.

Chapter 7

Concluding Remarks

In the scope of this project, various web technologies were explored and compared, as a basis for making an informed choice of the technology stack to use in the planned MVA application.

The final web application for MVA will not follow the standard client-server model, since the application has no need for a typical backend. The application will handle all user interactions on the client side. User settings will be stored in local storage, which allows web applications to store data persistently on the client side [W3C 2010]. However, local storage has a size limitation of 5 MB and is not sufficient for storing large datasets. The datasets will thus be stored with IndexedDB. IndexedDB is a low-level API for client-side storage of significant amounts of structured data [W3C 2015].

The Svelte frontend development framework will be used for developing the web application, as it is minimal, lightweight, and fast. The data will be rendered with a combination of Pixi.js, Konva.js, and Offscreen Canvas. The main graphics elements (for example parallel coordinates lines and scatterplot data points) will be rendered with Pixi.js since it offers the best performance for rendering. However, text and other similar elements (for example draggable filters) will be rendered with Konva.js, as it produces better quality text and shapes, and performance is not crucial for these elements. Offscreen Canvas web workers will be used in combination with Pixi.js to render large amounts of data simultaneously (in parallel).

When the MVA application development is finished, the application will also be packaged as a desktop application with Tauri. Tauri was chosen because it is minimal and lightweight. However, since Tauri is a relatively new library, it could cause some unforeseen issues. In that case, Electron.js will be used to convert the web application into a desktop application.

Bibliography

Accomazzo, Anthony, Ari Lerner, Nate Murray, Clay Allsopp, David Guttman, and Tyler McGinnis [2017]. *Fullstack React*. Fullstack.io, 12 Sep 2017. ISBN 0991344626 (cited on page 5).

Ahmed, Kamran [2023a]. *Backend Developer Roadmap*. 2023. https://roadmap.sh/backend (cited on page 3).

Ahmed, Kamran [2023b]. *Frontend Developer Roadmap*. 2023. https://roadmap.sh/frontend (cited on page 3).

Berson, Alex [1996]. *Client/Server Architecture*. McGraw-Hill, 29 Mar 1996. ISBN 0070056641 (cited on page 3).

Bostock, Mike, Jason Davies, Jeffrey Heer, and Vadim Ogievetsky [2011]. *D3.js*. 18 Feb 2011. https://d3js.org/ (cited on page 9).

Brandel, Jono [2014]. *Two.js*. 08 Jan 2014. https://github.com/jonobr1/two.js (cited on pages 9–10).

Cabello, Ricardo [2010]. *Three.js*. 24 Apr 2010. https://threejs.org/ (cited on pages 9–10).

Catuhe, David [2013]. *Babylon.js*. 17 Aug 2013. https://babylonjs.com/ (cited on page 9).

Deveria, Alexis [2023]. *Can I Use: WebGPU*. 27 May 2023. https://caniuse.com/webgpu (cited on page 8).

Fierens, Wout [2012]. *SVG.js*. 15 Dec 2012. https://svgjs.dev/docs/3.0/ (cited on page 10).

Filipova, Olga [2016]. *Learning Vue.js 2*. Packt Publishing, 13 Dec 2016. ISBN 1786469944 (cited on page 6).

Google [2016]. *Angular*. 14 Sep 2016. https://angular.io/ (cited on page 5).

Groves, Matt [2013]. *Pixi.js*. Feb 2013. https://pixijs.com/ (cited on pages 9–10).

Harris, Rich [2016]. *Svelte*. 26 Nov 2016. https://svelte.dev/ (cited on page 6).

Holthausen, Simon [2022]. *Svelte: A Beginner's Guide*. SitePoint, 10 Feb 2022. ISBN 1925836487 (cited on page 6).

Huggins, Jason [2004]. *Selenium*. 2004. https://selenium.dev/ (cited on page 10).

Khronos Group [2011]. *WebGL*. 03 Mar 2011. https://webgl.org/ (cited on page 8).

Lavrenov, Anton [2012]. *Konva.js*. 04 Mar 2012. https://konvajs.org/ (cited on pages 9–10).

MDN [2018]. *Offscreen Canvas*. Mozilla Developer Network, Sep 2018. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (cited on page 8).

MDN [2023]. *Canvas API*. Mozilla Developer Network, 19 Feb 2023. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (cited on page 7).

Meta [2013]. *React*. 29 May 2013. https://react.dev/ (cited on page 5).

Murray, Nate, Felipe Coury, Ari Lerner, and Carlos Taborda [2018]. *ng-book: The Complete Guide to Angular 5th Edition*. CreateSpace Independent Publishing Platform, 06 Feb 2018. ISBN 1985170280 (cited on page 5).

OpenJS [2013]. *Electron.js*. 15 Jul 2013. https://electronjs.org/ (cited on page 13).

Slay Lines [2020]. *Canvas Engines Comparison*. 28 Apr 2020. https://github.com/slaylines/canvas-engines-comparison (cited on page 10).

Tauri [2019]. *Tauri*. 18 Dec 2019. https://tauri.app/ (cited on page 13).

W3C [2001]. *SVG*. World Wide Web Consortium, 04 Sep 2001. https://w3.org/Graphics/SVG/ (cited on page 7).

W3C [2010]. *Web Storage API*. World Wide Web Consortium, 18 Nov 2010. https://w3.org/TR/webstorage/ (cited on page 19).

W3C [2015]. *IndexedDB*. World Wide Web Consortium, 08 Jan 2015. https://w3.org/TR/IndexedDB/ (cited on page 19).

W3C [2021]. *WebGPU*. World Wide Web Consortium, 18 May 2021. https://w3.org/TR/webgpu/ (cited on page 8).

Whitney, Justin [2013]. *Surviving the Zombie Apocalypse: Manipulating SVG with JavaScript*. 24 Jun 2013. https://sitepoint.com/surviving-the-zombie-apocalypse-manipulating-svg-with-javascript/ (cited on page 7).

You, Evan [2014]. *Vue*. Feb 2014. https://vuejs.org/ (cited on page 6).