BenchLines: Benchmarking Interactive Web Graphics for Parallel Coordinates

Michael Anderson, Jyothish Atheendran, and Filip Ljubotina

706.041 Information Architecture and Web Usability 3VU WS 2025/2026
Graz University of Technology

03 Feb 2026

Abstract

This report describes BenchLines, a benchmarking project designed to systematically evaluate and compare four core web rendering technologies: SVG-DOM, Canvas2D, WebGL, and WebGPU, in the context of drawing polylines for a parallel coordinates visualization. To provide deeper insights, BenchLines also incorporates two widely used graphics libraries built on these technologies, namely Pixi.js and Three.js for WebGL and WebGPU.

BenchLines enables standardised performance measurements, allowing for accurate benchmarking of rendering speed as well as common interactive features in a parallel coordinates visualization. This approach facilitates both quantitative and subjective evaluation of performance across multiple interactive features. Based on the results, the report presents recommendations for selecting appropriate rendering approaches depending on performance requirements and workload. These findings aim to guide developers in making informed decisions when building high-performance, visually rich web applications.

We have used Grammarly to improve the language in this paper.

Contents

Contents ii

List of Figures iii

List of Tables v

List of Listings vii

1 Introduction 1

2 Related Work 3
- 2.1 Parallel Coordinates . . . 3
- 2.2 Core Web Graphics Technologies . . . 3
  - 2.2.1 Canvas2D . . . 3
  - 2.2.2 SVG-DOM . . . 4
  - 2.2.3 WebGL . . . 4
  - 2.2.4 WebGL 1.0 . . . 4
  - 2.2.5 WebGL 2.0 . . . 4
  - 2.2.6 WebGPU . . . 5
- 2.3 Web Graphics Libraries . . . 5
  - 2.3.1 Three.js . . . 5
  - 2.3.2 Pixi.js . . . 5

3 BenchLines System Architecture 7
- 3.1 Background Layer . . . 7
- 3.2 Rendering Layer . . . 11
- 3.3 Interaction Layer . . . 11
  - 3.3.1 Dimension Interactions . . . 11
  - 3.3.2 Polyline Hover and Selection . . . 12
    - 3.3.2.1 Hover Detection . . . 12
    - 3.3.2.2 Selection . . . 15

4 Rendering Parallel Coordinates 19
- 4.1 Core Web Graphics Technologies . . . 19
  - 4.1.1 SVG-DOM . . . 19
  - 4.1.2 Canvas2D . . . 19
  - 4.1.3 WebGL . . . 19
  - 4.1.4 WebGPU . . . 22

4.2 Using Web Graphics Libraries 29
4.2.1 Three.js 29
4.2.2 Pixi.js 29

5 Benchmarking Results **33**

6 Future Work **35**

7 Concluding Remarks **37**

Bibliography **39**

List of Figures

1.1 BenchLines WebApp 2

3.1 Background Layer 8
3.2 Rendering Layer 11
3.3 Interaction Layer 12
3.4 Hover Detection: Distance from Point to Line 13
3.5 Polyline Selection: Line and Box Selection Modes 15

List of Tables

4.1 WebGPU Color Usage 22

5.1 BenchLines: Benchmark Results. 34

List of Listings

3.1 Background Layer: Canvas Initialization . . . 8
3.2 Background Layer: Rasterization . . . 9
3.3 Background Layer: Canvas2D Rendering . . . 10
3.4 Hover Detection: GPU-Based Point-to-Line Distance Calculation . . . 13
3.5 Hover Detection: CPU-Based Dimension Span Detection . . . 14
3.6 Hover Detection: CPU-Based Point-to-Line Distance Calculation . . . 14
3.7 Line Selection: GPU-Based Line Segment Intersection . . . 16
3.8 Line Selection: CPU-Based Line Segment Intersection . . . 16
3.9 Box Selection: GPU-Based Box Intersection . . . 17
3.10 Box Selection: CPU-Based Box Intersection . . . 17
3.11 Line and Box Selection: CPU-Based Multi-Dimensional Span Identification . . . 18

4.1 SVG-DOM: Polyline Rendering . . . 20
4.2 Canvas2D: Polyline Rendering . . . 20
4.3 WebGL: Vertex Shader . . . 21
4.4 WebGL: Fragment Shader . . . 21
4.5 WebGL: Program . . . 22
4.6 WebGL: Polyline Rendering . . . 23
4.7 WebGPU: Vertex and Fragment Shaders . . . 24
4.8 WebGPU: Render Pipeline . . . 25
4.9 WebGPU: Vertex Buffer layout . . . 26
4.10 WebGPU: Bind Groups and Uniform Buffers . . . 26
4.11 WebGPU: Lines with Custom Thickness . . . 27
4.12 WebGPU: Polyline Rendering . . . 28
4.13 Three.js: Line Rendering . . . 30
4.14 Pixi.js: Polyline Rendering . . . 31

Chapter 1

Introduction

The landscape of web rendering has undergone significant evolution over the past decade, offering a diverse set of technologies for creating interactive and visually rich web applications. Modern web applications rely on rendering engines not only for displaying graphics, but also for handling complex interactions and visualisations efficiently. The four core web rendering technologies, SVG-DOM, Canvas2D, WebGL, and WebGPU each offer unique capabilities and trade-offs in terms of performance, flexibility, hardware requirements, and the development effort needed to leverage them effectively. Understanding these differences is crucial for developers to optimize user experience, development workflow, and hardware utilization.

This report describes BenchLines, an open-source benchmarking tool designed to evaluate the performance of web rendering technologies in the context of drawing polylines for a parallel coordinates visualization [Anderson et al. 2026]. Parallel coordinates is a widely used visualization technique for exploring multidimensional datasets. The BenchLines project builds upon earlier work called SPCD3 [Andrews et al. 2026].

BenchLines implements parallel coordinates with each of the featured technologies, so that performance can be measured and compared. It provides numerous datasets with differing numbers of dimensions and records. It is possible both to run trials measuring rendering performance and to manually evaluate common interactive features. Figure 1.1 shows the user interface of BenchLines.

In addition to the core technologies, BenchLines also includes support for two widely used libraries built on top of the WebGL and WebGPU platforms: Pixi.js and Three.js. These libraries simplify GPU utilization by providing higher-level abstractions, allowing developers to implement complex graphics, without delving into low-level programming details.

The primary contributions of this work are:

1. Comprehensive benchmarking of web rendering technologies in an interactive visualization scenario.
2. Analysis of the performance and interactivity of each rendering technology.
3. Practical recommendations for selecting an appropriate rendering technology or library based on performance, interactivity, and development requirements.

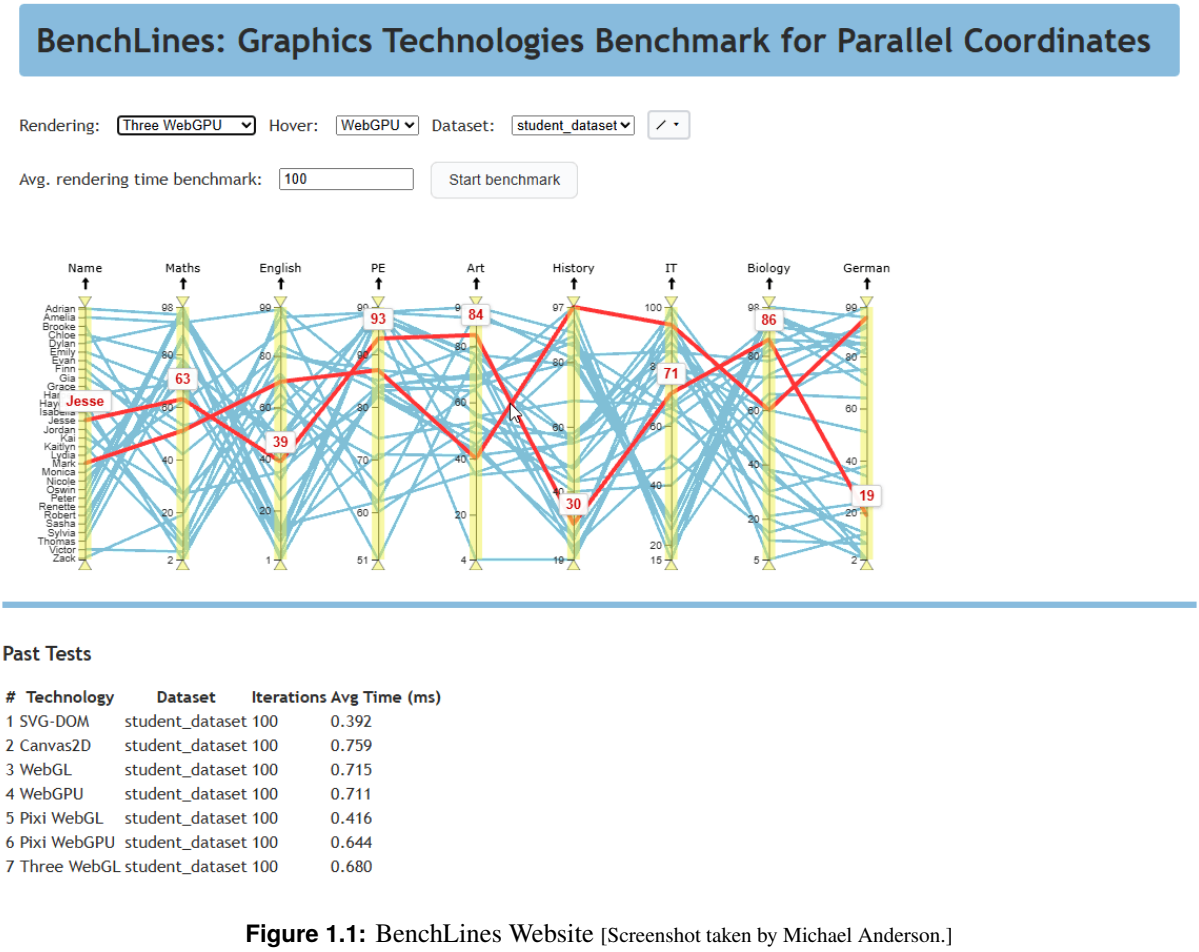


Figure 1.1: BenchLines Website [Screenshot taken by Michael Anderson.]

Chapter 2

Related Work

This chapter describes some of the techniques and technologies involved in the BenchLines project.

2.1 Parallel Coordinates

Parallel coordinates visualizations are widely used for visualising and exploring multidimensional data. Unlike traditional plots, where a dimension is typically represented by an axis in two or three-dimensional space, parallel coordinates represent each dimension as a vertical axis arranged in parallel. Individual data points are represented as a polyline that intersects each axis at the corresponding value. The technique was popularised by Inselberg [2009].

These visualisations are particularly useful for exploratory data analysis [Weitz 2020]. They allow patterns, correlations, and outliers to be identified across multiple dimensions simultaneously [Gruber and Andrews 2026]. However, the complexity of rendering increases with the number of dimensions and data points, which can lead to performance challenges, especially for interactive applications.

2.2 Core Web Graphics Technologies

Modern web-based visualizations rely on a range of graphics technologies that differ in rendering models, performance characteristics, and interaction support. Understanding the historical development and core principles of these technologies provides essential context for the design decisions behind BenchLines.

2.2.1 Canvas2D

The Canvas2D API can be accessed via a `<canvas>` element with a `"2d"` rendering context. It provides an interface for rendering two-dimensional graphics directly in the browser. Unlike retained-mode graphics systems such as SVG, Canvas2D operates using immediate-mode rendering, meaning the canvas does not "remember" objects, but every frame must be explicitly redrawn [Taivalsaari et al. 2017]. This makes Canvas2D ideal for dynamic graphics, games, and animations where content changes frequently [Aircada 2024]. The API provides methods for direct pixel manipulation, such as `getImageData()` and `putImageData()`, which allow control over individual pixels for custom rendering.

Performance-wise, Canvas2D is mostly CPU-based, and performance degrades with complex or high-volume drawing operations [PixelFree 2024]. Since Canvas2D utilizes the CPU to render content directly to the DOM, browsers have optimized Canvas2D to be very performative and even offload some rendering tasks to the GPU in the background [Larabel 2023]. In this sense, in certain scenarios, like plotting lines, Canvas2D remains competitive in terms of rendering performance.

2.2.2 SVG-DOM

The SVG Document Object Model (SVG-DOM) refers to the use of JavaScript to programmatically create, manipulate, and interact with Scalable Vector Graphics (SVG) elements in the browser's Document Object Model (DOM). Since SVG describes vector graphics objects, there is no loss of quality when the graphics are scaled up or down. However, since SVG elements are stored in the DOM, graphics containing many hundreds or thousands of elements can significantly strain the CPU.

According to Wikipedia [2025], the World Wide Web Consortium (W3C) began developing the SVG standard in 1998 after six competing submissions for vector graphics standards were submitted. SVG 1.0 became a W3C Recommendation on 04 Sep 2001. The specification continued to evolve, with SVG 1.1 Second Edition released on 16 Aug 2011. Most current web browsers support SVG 1.1. A W3C Candidate Recommendation for SVG 2 was published on 15 Sep 2016, with the latest draft on 14 Sep 2025, but browser support for SVG 2 is feature-based rather than version-based, and no modern browser fully implements the entire SVG 2 spec.

2.2.3 WebGL

The Web Graphics Library (WebGL) is a JavaScript API that for rendering interactive 2d and 3d graphics directly in the browser [MDN 2025] using the HTML `<canvas>` element. The first version of WebGL was released at the Game Developers Conference in San Francisco in 2011 [Khronos 2011]. The technology is based on the OpenGL ES (Embedded Systems) standard, which was designed for efficient graphics rendering on mobile or other embedded devices [Gilbert 2025]. As such, WebGL works natively in all major browsers [Deveria 2025], meaning users do not need to enable anything to be able to utilize WebGL.

2.2.4 WebGL 1.0

WebGL 1.0 is built on the OpenGL ES 2.0 standard and uses the `<canvas>` element with the `"webgl"` rendering context. Using the GPU means graphics can be rendered quicker and more efficiently than on the CPU, without the need for additional plugins or adjusting browser settings. WebGL 1.0 supports programmable shaders and 3d geometry, which allows developers to make complex visualizations, games, or simulations entirely within the browser.

WebGL 1.0 also has its drawbacks. Since it is based on OpenGL ES 2.0, some of the advanced features available in later versions of OpenGL ES are not supported, such as multiple render targets, 3d textures, and some instancing. This can limit the ability to create more complex visualizations or simulations as compared to WebGL 2.0. Also, WebGL exposes low-level GPU operations, meaning developers need a solid understanding or background in graphics programming, including the use of shader programming and buffer management, to fully utilize WebGL 1.0 to its full capabilities. To make this easier, some web graphics libraries, such as Three.js and Pixi.js, have been developed. These frameworks simplify working with WebGL, providing easier-to-understand and utilize methods for accessing the GPU powered graphics rendering while obfuscating some of the more complicated details.

2.2.5 WebGL 2.0

WebGL 2.0 was released in 2017 [Mo 2017] and is based on the OpenGL ES 3.0 standard. Like WebGL 1.0, it provides a JavaScript API for hardware-accelerated rendering of 2d and 3d graphics directly in the browser. WebGL 2.0 is initialized via the `<canvas>` element with the `"webgl2"` rendering context. WebGL 2.0 also allows developers access to the GPU which leads to improved rendering performance.

WebGL 2.0 extends the features available in WebGL 1.0 by offering several features made possible by OpenGL ES 3.0. These include support for 3d textures, instanced rendering, multiple render targets, and the ability to query objects directly. These additional features make it possible for developers to create more complex graphics and visualizations while improving the performance. Developers can

now implement complex particle systems and large-scale data visualizations more efficiently than with WebGL 1.0.

Despite this, WebGL 2.0 is not without limitations. While it is supported by most modern browsers, it is slightly less supported than WebGL 1.0, which can affect compatibility with older devices. WebGL 2.0 is also a low-level API, meaning developers must have a strong understanding of graphics programming to take full advantage of the additional features. Libraries such as Three.js and Pixi.js continue to be useful to simplify this process, allowing developers a simpler method to utilize these advanced features without directly interfacing with the GPU.

Overall, WebGL 2.0 allows for better performance and additional features not available in WebGL 1.0. This makes it one of the more common choices for modern, high-performance web graphics applications.

2.2.6 WebGPU

The WebGPU API started as an initial idea by Google called "Explicit Web Graphics API" presented by Corentin Wallez [Wallez 2016] on 08 Jun 2016. The main aim was to propose a "WebGL Next" that could replace WebGL. Later, with subsequent evolutions, W3C published their first public working draft of WebGPU on 18 May 2011 [Malyshau and Ninomiya 2021]. The WebGPU API can be accessed by web browsers using JavaScript. This provides access to the Vulkan [Vulkan 2026], Metal [Apple 2026], or Direct3D 12 [Microsoft 2026] technologies of the Linux, Apple or Windows system, respectively.

WebGPU is available in all major desktop browsers [Deveria 2026]. However, in certain Linux systems, it has to be enabled by setting a browser flag. There has been an effort to develop a translation layer that can convert WebGL calls to WebGPU calls [Han et al. 2025]. Although there are some limitations (such as a lack of one-to-one calls from WebGL to WebGPU), and even with the translation overhead, converting code from WebGL to WebGPU was shown to yield better performance.

2.3 Web Graphics Libraries

Working directly with low-level web graphics technologies such as WebGL and WebGPU can be challenging. They require deep knowledge of the graphics pipeline and involve large amounts of code. This is why higher-level graphics libraries have become essential in modern web development. They sit on top of WebGL or WebGPU and provide clean, intuitive APIs that simplify the process.

2.3.1 Three.js

Three.js [Danchilla 2012] is a popular open-source JavaScript library that simplifies rendering graphics using WebGL and WebGPU. It abstracts away the lower-level programming, thus making it easy and faster for programmers to work on the underlying technologies. Three.js was created by Ricardo Cabello (also known as Mr.doob) and was first released in Apr 2010 on GitHub [Cabello 2026]. When launched, it supported WebGL 1.0. However, through generous community contributions, it was able to support WebGL 2.0 in the r95 version update on 31 Jul 2018. Since version r118 (released on 27 Jun 2020), Three.js uses WebGL 2.0 by default, with WebGL 1.0 as a fallback. Since version r171 published on 29 Nov 2024, WebGPU was made available production-ready with zero-config imports. As of 2026, Three.js is used for several projects such as web graphics rendering, game development, WebXR Expansion and much more [Lecamus 2026].

2.3.2 Pixi.js

Pixi.js is an open-source 2D web graphics rendering library [Pixi 2026]. It was developed by Mathew Groves, where the initial commit of the project was made on 21 Jan 2013. It is a fast HTML5 Creation Engine that by default leverages WebGL for hardware-accelerated 2D rendering and uses Canvas as a fallback if certain older platforms cannot handle WebGL. PixiJS v8 introduced WebGPU support along

with many other features and improvements. This is an actively maintained project and is used in gaming, data visualization, interactive web apps, and so on.

Chapter 3

BenchLines System Architecture

BenchLines utilises three canvases (layers) for drawing and interaction. The lowermost canvas, the background layer, contains a background raster image, showing all polyline rendered in the inactive color (gray). The middle canvas serves as the rendering layer, where active polylines are drawn. The third, foreground, layer is an overlay interaction layer that handles mouse input, such as hovering over polylines and manipulating a filter control.

3.1 Background Layer

The background layer consists of a raster image placed beneath all other layers. The idea is to draw all possible polylines in the inactive color (gray) onto a canvas, and then only draw the currently active polylines in blue on top of them. An example of a background canvas is shown in Figure 3.1.

Parallel coordinate visualisations can contain thousands of polylines, and rendering all of them in real time can be computationally expensive. By creating a rasterised background image during scene initialisation, the inactive lines do not need to be re-rendered during normal operations. Some interactive operations, such as reordering or inverting dimensions, require that the background canvas be re-rendered. This process allows for inactive background lines to be displayed when filtering the dataset without having to continuously re-render inactive lines while the user filters the data. During a benchmarking trial, only the rendering of the full set of active polyline is considered.

To implement the background image rasterisation, BenchLines uses an offscreen WebGL canvas that matches the height and width of the main background canvas. This canvas is not visible to users. The code used to initialize it is shown in Listing 3.1. Using WebGL, the GPU is employed to render the inactive lines in the same manner as the active lines. Once all polylines are rendered to the offscreen canvas, the content is rasterised into an image and placed onto the designated background canvas, as shown in Listing 3.2. If WebGL is unavailable, Canvas2D is used to render the background lines directly onto the background canvas, as shown in Listing 3.3.

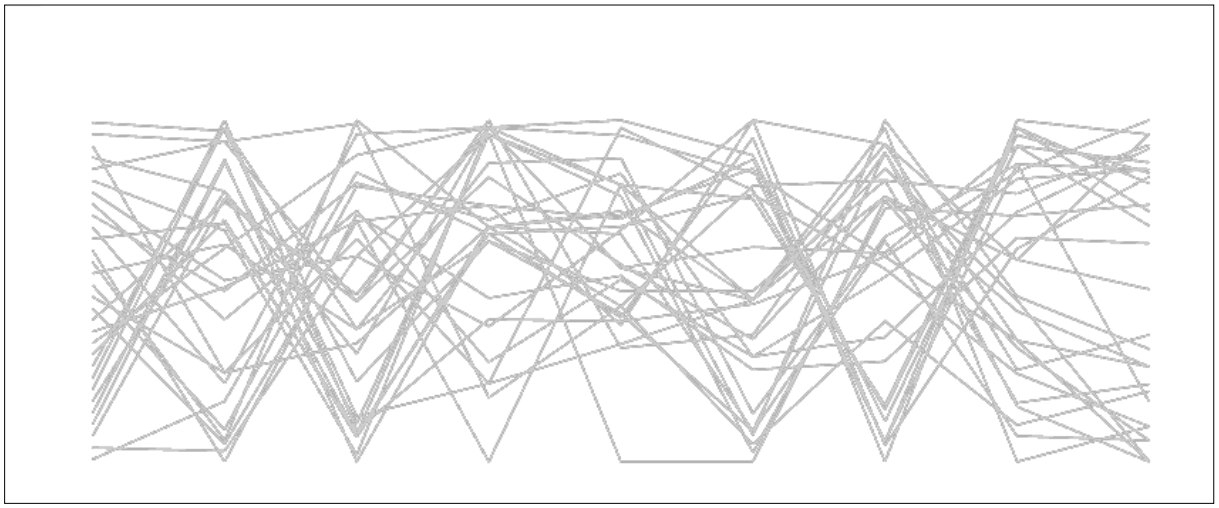


Figure 3.1: Background Layer: Every polyline is drawn in the inactive color (gray) into the background canvas. [Screenshot taken by Michael Anderson.]

```
1  //create background lines image
2  inactiveLinesCanvas = document.createElement("canvas");
3  inactiveLinesCanvas.width = canvasEl.width;
4  inactiveLinesCanvas.height = canvasEl.height;
5  inactiveLinesCanvas.style.position = "absolute";
6  inactiveLinesCanvas.style.top = canvasEl.style.top;
7  inactiveLinesCanvas.style.left = canvasEl.style.left;
8  inactiveLinesCanvas.style.pointerEvents = "none";
9  inactiveLinesCanvas.style.width = canvasEl.style.width;
10 inactiveLinesCanvas.style.height = canvasEl.style.height;
11
12 // Insert behind the main canvas
13 canvasEl.parentNode?.insertBefore(inactiveLinesCanvas, canvasEl);
14
15 redrawWebGLBackgroundLines(dataset, parcoords);
```

Listing 3.1: Background Layer: Typescript code to initialize a background canvas for background lines to be rendered with WebGL.

```typescript
export function rasterizeInactiveLinesToCanvas(targetCanvas: HTMLCanvasElement) {
  if (!gl) {
    return;
  }

  const sourceCanvas = gl.canvas as HTMLCanvasElement;
  const w = sourceCanvas.width;
  const h = sourceCanvas.height;

  const pixels = new Uint8Array(w * h * 4);
  gl.readPixels(0, 0, w, h, gl.RGBA, gl.UNSIGNED_BYTE, pixels);

  const ctx = targetCanvas.getContext("2d");
  if (!ctx) return;

  const imageData = ctx.createImageData(w, h);

  // Flip Y (WebGL bottom-left -> Canvas top-left)
  for (let y = 0; y < h; y++) {
    for (let x = 0; x < w; x++) {
      const src = ((h - y - 1) * w + x) * 4;
      const dst = (y * w + x) * 4;

      imageData.data[dst]     = pixels[src];
      imageData.data[dst + 1] = pixels[src + 1];
      imageData.data[dst + 2] = pixels[src + 2];
      imageData.data[dst + 3] = pixels[src + 3];
    }
  }

  ctx.putImageData(imageData, 0, 0);
}
```

Listing 3.2: Background Layer: Typescript code to rasterize (draw) every polyline in the inactive color (gray) to the background canvas.

```
function drawInactiveLinesCanvas2D(dataset: any[], parcoords: any,
  targetCanvas: HTMLCanvasElement) {
  const ctx = targetCanvas.getContext("2d");
  if (!ctx) return;

  const dpr = window.devicePixelRatio || 1;

  // reset transforms
  ctx.setTransform(dpr, 0, 0, dpr, 0, 0);
  ctx.clearRect(0, 0, targetCanvas.width, targetCanvas.height);

  ctx.strokeStyle = "rgba(200,200,200,1)";
  ctx.lineWidth = 1;

  for (const d of dataset) {
    const id = getLineNameCanvas(d);
    if (lineState[id]?.active) continue;

    ctx.beginPath();
    let first = true;
    parcoords.newFeatures.forEach((name: string) => {
      const x = (parcoords.dragging[name] ?? parcoords.xScales(name)) * dpr;
      const y = parcoords.yScales[name](d[name]) * dpr;

      if (first) {
        ctx.moveTo(x, y);
        first = false;
      } else {
        ctx.lineTo(x, y);
      }
    });
    ctx.stroke();
  }
}
```

Listing 3.3: Background Layer: Typescript code to render the background lines directly to the background canvas (in case WebGL is not available).

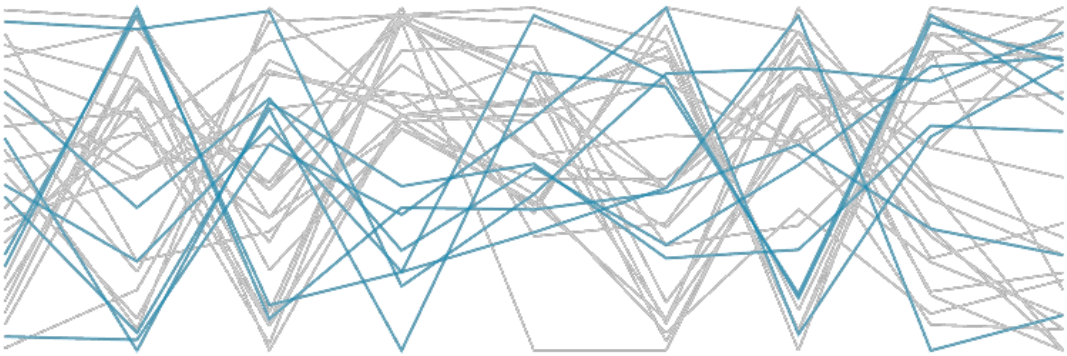


Figure 3.2: Rendering Layer: Active polylines are rendered in the active color (blue) atop the background layer. [Screenshot taken by Michael Anderson.]

3.2 Rendering Layer

BenchLines supports polyline rendering through multiple web graphics technologies, allowing users to select the rendering backend best suited to their performance and compatibility requirements. Supported backends include SVG-DOM, Canvas2D, WebGL, and WebGPU. BenchLines additionally provides implementations built on top of two established rendering libraries, Three.js and Pixi.js, for both WebGL and WebGPU. The specific implementation details and design considerations for each rendering technology are discussed in Chapter 4.

Regardless of the selected rendering backend, BenchLines employs a layered canvas architecture to separate rendering from interaction. Rendered polylines are drawn on top of a static background canvas that contains pre-rasterised visual elements, reducing the need for repeated redraws of inactive content, as illustrated in Figure 3.2.

3.3 Interaction Layer

The interaction layer is an overlay canvas placed above the rendering layer. It contains interactive controls and is responsible for capturing mouse input, including pointer movement and click events. Interaction events detected on this overlay layer drive updates to the rendering layer, enabling responsive operations such as hovering, highlighting, and filtering without unnecessary recomputation of the background. The interaction layer is shown in Figure 3.3, in front of the other two layers. BenchLines offers three main categories of interactivity: dimension-based interactions, polyline hover effects, and polyline selection.

3.3.1 Dimension Interactions

BenchLines provides three primary methods for interacting with dimensions. Users can reorder dimensions by clicking and dragging dimension names horizontally, allowing users to analyze relationships between dimensions. Additionally, clicking the arrow icon beneath a dimension name inverts the sorting order of its values. Finally, dimensions support range-based filtering through draggable yellow arrows positioned at the top and bottom of each axis. By adjusting these arrows, users can define a value range. Any polylines falling outside this range are deactivated and displayed in grey to indicate their filtered status.

It is important to note that all dimension interactions trigger a cascade effect throughout the visualisation. When a user modifies a dimension, all lower layers are automatically notified and updated to maintain consistency.

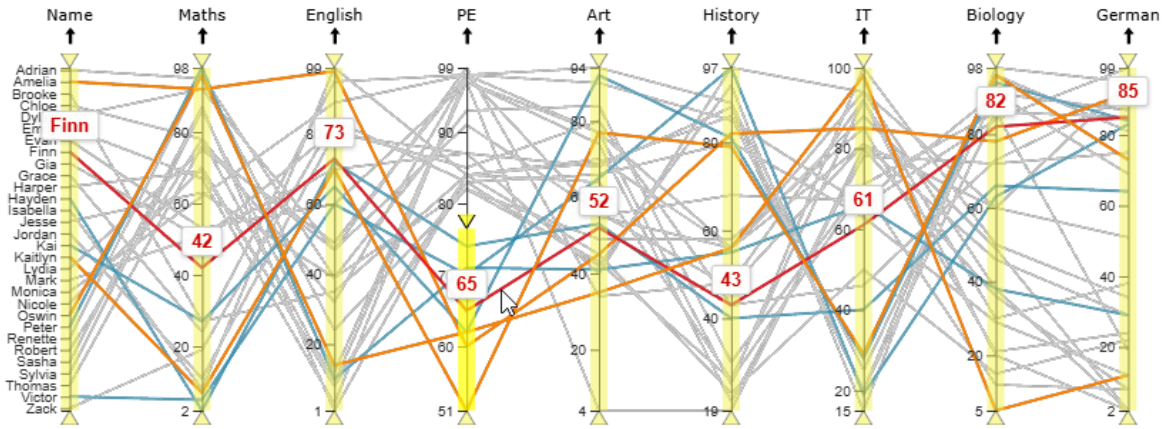


Figure 3.3: Hovered Interactive Lines Atop Render Layer and Background Image. [Screenshot taken by Michael Anderson.]

3.3.2 Polyline Hover and Selection

To implement hovering and selection, a hover layer is placed immediately above the plot area containing the chart. It attaches three event listeners (mousemove, mousedown, and mouseup) directly to the plot area. This enables continuous tracking of mouse position and gestures relative to the rendered polylines. Since the hover layer operates on the same dataset used for rendering, it maintains a consistent spatial reference frame, allowing real-time correlation between cursor position and polyline geometry without requiring additional data transformation or synchronisation overhead. The layer supports three distinct interaction modes: point-based hover detection, line-based selection, and box selection.

The core interaction detection mechanism employs raycasting techniques implemented from scratch, providing functionality equivalent to existing library solutions, while maintaining full control over the implementation. The raycasting logic has been deliberately decoupled from any specific rendering technology, enabling it to operate independently of whether the visualisation uses Canvas2D, WebGL, or other rendering backends. This architectural decision facilitates two parallel implementations: a CPU-based JavaScript solution for broad compatibility and a GPU-accelerated version leveraging compute shaders for enhanced performance.

For the hover layer, polylines are represented internally using a flattened, fixed-size buffer structure. Each polyline is allocated a predetermined segment of a Float32Array buffer capable of storing up to maxPointsPerLine coordinate pairs. Points are encoded sequentially in the form [x1, y1, x2, y2, x3, y3, ...], where consecutive pairs define individual line segments. This representation decomposes each polyline into a series of line segments, enabling efficient segment-by-segment distance calculations during raycasting operations.

3.3.2.1 Hover Detection

The hover detection mechanism determines whether a polyline is within proximity of the cursor by computing the perpendicular distance from the mouse position to each line segment. As illustrated in Figure 3.4, for a given line segment defined by points P1 and P2, and a cursor position, the projection of the cursor onto the segment vector is calculated. This is achieved through vector mathematics: the dot product of the cursor-relative vector with the segment direction vector, normalized by the segment's squared length, yields a clamping parameter that determines the closest point on the segment. The Euclidean distance from the cursor to this projected point represents the minimum distance to the line segment. If this distance falls below a configurable threshold, the polyline is considered hovered and subsequently rendered in red. The configurable distance threshold enables fuzzy selection capabilities,

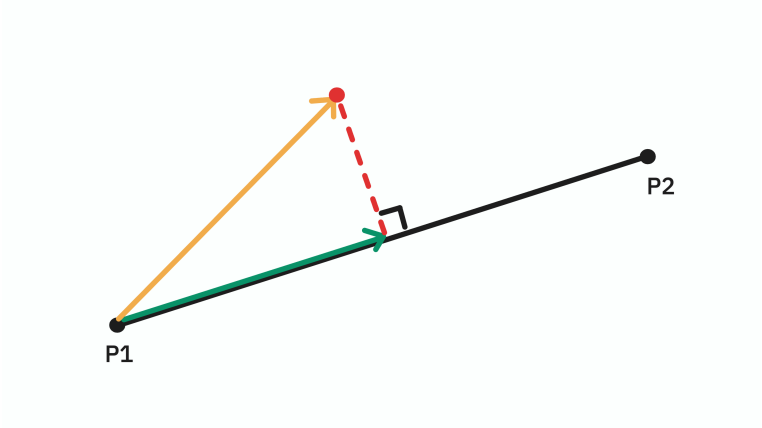


Figure 3.4: Hover Detection: Calculating the distance from a point to a line using the vector product. [Diagram drawn by Filip Ljubotina.]

```
// Hover mode: point-to-line distance
for (var i = 0u; i < maxPts - 1u; i++) {
    let idx = lineIdx * maxPts + i;
    let p1 = lineData[idx];
    let p2 = lineData[idx + 1u];

    let pa = mousePos - p1;
    let ba = p2 - p1;
    let h = clamp(dot(pa, ba) / dot(ba, ba), 0.0, 1.0);
    let d = length(pa - ba * h);

    if (d < hoverDist) {
        hit = 1u;
        break;
    }
}
```

Listing 3.4: Hover Detection: WGSL code for GPU-based point-to-line distance calculation.

allowing adjustments to the hover sensitivity.

The distance calculation follows a standard point-to-line algorithm. Given a cursor point and line segment endpoints, we first compute the vector from the segment start to the cursor, then project this onto the segment direction. The projection is clamped to the [0,1] interval to ensure the closest point lies within the segment bounds rather than on its infinite extension. The final distance is simply the magnitude of the vector from the cursor to this clamped projection point.

Two implementations are available for computing the point-to-line distance: one GPU-based for WebGPU, and the second CPU-based written in standard JavaScript. Both implementations employ identical mathematical logic, yet differ significantly in their execution strategy. The GPU-based implementation is shown in Listing 3.4. It is written in WGSL for WebGPU compute shaders and assigns one thread per polyline. Each thread iterates through all segments of its assigned polyline sequentially, computing the distance to each segment until either a segment within the hover threshold is detected or all segments in the polyline have been examined. This parallelization strategy processes all polylines simultaneously, with the workgroup size of 256 threads enabling efficient utilisation of GPU compute units.

The CPU-based implementation adopts a different optimization strategy suited to sequential processing. Rather than examining all segments of every polyline, it first determines which two adjacent dimensions

```
private findSegmentIndex(x: number): number {
  const positions = this.dimensionXPositions;
  if (positions.length < 2) return -1;

  for (let i = 0; i < positions.length - 1; i++) {
    const leftX = positions[i];
    const rightX = positions[i + 1];

    const minX = Math.min(leftX, rightX);
    const maxX = Math.max(leftX, rightX);

    if (x >= minX && x <= maxX) {
      return i;
    }
  }

  return -1;
}
```

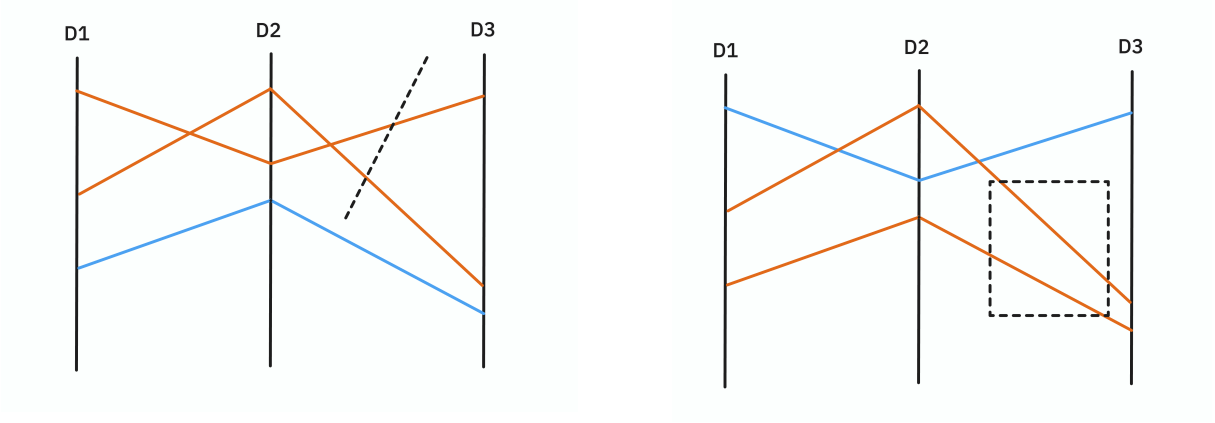
Listing 3.5: Hover Detection: TypeScript code for CPU-based detection of the dimension span.

```
private pointToSegmentDistance(
    px: number, py: number,
    x1: number, y1: number,
    x2: number, y2: number
): number {
    const pax = px - x1;
    const pay = py - y1;
    const bax = x2 - x1;
    const bay = y2 - y1;

    const dotBA = bax * bax + bay * bay;
    if (dotBA === 0) {
        return Math.hypot(pax, pay);
    }

    const h = Math.max(0, Math.min(1, (pax * bax + pay * bay) / dotBA));
    const projX = pax - bax * h;
    const projY = pay - bay * h;

    return Math.hypot(projX, projY);
}
```

Listing 3.6: Hover Detection: TypeScript code for CPU-based point-to-line distance calculation.

the cursor is positioned between. As shown in Listing 3.5, this is achieved by iterating through the stored x-coordinates of each dimension axis and identifying the dimensional span containing the cursor's horizontal position. The method accounts for potential dimension reordering by comparing against both the minimum and maximum x-values of each span. Once the relevant dimensional span is identified, only the polyline segment within that span undergoes the distance computation for each polyline. The code for distance computation is shown in Listing 3.6. This spatial filtering significantly reduces the computational workload, as each polyline typically contains many segments corresponding to the number of dimensions, yet only one falls within the cursor's horizontal position at any given time.

One might question why the same dimensional filtering optimization strategy was not applied to the GPU-based implementation. While technically feasible, this approach was not pursued, because the



(a) Line selection.

(b) Box selection.

Figure 3.5: Polyline Selection: Line and box selection modes. [Both diagrams created by Filip Ljubotina.]

GPU's parallel architecture diminishes its relative benefit. Since all polylines are processed simultaneously and each thread terminates immediately upon detecting a hover, the additional segments checked represent minimal overhead. Furthermore, computing the line segment index on the CPU and passing it to the shader via a uniform buffer on every mouse movement would introduce synchronization latency that could offset any computational savings.

3.3.2.2 Selection

BenchLines supports two selection modes that enable users to select multiple polylines simultaneously: line selection and box selection. Line selection, illustrated in Figure 3.5a, detects all polylines whose segments intersect a user-drawn line. Box selection, shown in Figure 3.5b, identifies polylines that either pass through or have segments contained within a rectangular region defined by the user. Both selection modes are initiated by clicking and dragging within the plot area, with the selection type determined by a configurable setting.

The line selection algorithm determines whether two line segments intersect using a parametric approach. For a user-drawn line from point P1 to P2 and a polyline segment from P3 to P4, the algorithm computes the denominator of the intersection equation to check for parallel lines. If the segments are not parallel, two parameters (ua and ub) are calculated, representing the relative position of the intersection point along each segment. If both parameters fall within the [0,1] interval, the segments intersect within their defined bounds. Listing 3.7 and Listing 3.8 show the GPU-based and CPU-based implementations, respectively, demonstrating identical mathematical logic.

The box selection algorithm extends the intersection logic to rectangular regions. A polyline is considered selected if any of its segments either has an endpoint within the box boundaries or intersects with any of the four box edges. Listing 3.9 shows the GPU-based implementation, which first performs a quick containment check for segment endpoints before testing against each edge. The CPU-based implementation in Listing 3.10 follows the same two-phase approach: an initial endpoint containment test followed by iteration through all four box edges, utilising the line segment intersection function from Listing 3.8.

The GPU-based implementation, as with hover detection, assigns one thread per polyline and iterates through all polyline segments until an intersection is found. The CPU-based implementation employs an extended version of the dimensional filtering optimization. As shown in Listing 3.11, rather than identifying a single-dimensional span between two adjacent dimensions as in hover detection (Listing 3.5), the algorithm identifies all dimensional spans that fall within the horizontal extent of the drawn selection

```
fn lineSegmentsIntersect(
  p1: vec2<f32>, p2: vec2<f32>,
  p3: vec2<f32>, p4: vec2<f32>
) -> bool {
  let denom = (p4.y - p3.y) * (p2.x - p1.x) -
              (p4.x - p3.x) * (p2.y - p1.y);
  if (abs(denom) < 0.0001) {
    return false;
  }

  let ua = ((p4.x - p3.x) * (p1.y - p3.y) -
            (p4.y - p3.y) * (p1.x - p3.x)) / denom;
  let ub = ((p2.x - p1.x) * (p1.y - p3.y) -
            (p2.y - p1.y) * (p1.x - p3.x)) / denom;

  return ua >= 0.0 && ua <= 1.0 &&
         ub >= 0.0 && ub <= 1.0;
}
```

Listing 3.7: Line Selection: WGSL code for GPU-based line segment intersection.

```
private lineSegmentsIntersect(
  p1x: number, p1y: number, p2x: number, p2y: number,
  p3x: number, p3y: number, p4x: number, p4y: number
): boolean {
  const denom =
      (p4y - p3y) * (p2x - p1x) -
      (p4x - p3x) * (p2y - p1y);
  if (Math.abs(denom) < 0.0001) {
      return false;
  }

  const ua =
      ((p4x - p3x) * (p1y - p3y) -
       (p4y - p3y) * (p1x - p3x)) / denom;
  const ub =
      ((p2x - p1x) * (p1y - p3y) -
       (p2y - p1y) * (p1x - p3x)) / denom;

  return ua >= 0 && ua <= 1 &&
         ub >= 0 && ub <= 1;
}
```

Listing 3.8: Line Selection: TypeScript code for CPU-based line segment intersection.

```
fn lineSegmentIntersectsBox(
  p1: vec2<f32>, p2: vec2<f32>,
  minX: f32, maxX: f32, minY: f32, maxY: f32
) -> bool {
  // Check if either endpoint is inside the box
  if ((p1.x >= minX && p1.x <= maxX &&
     p1.y >= minY && p1.y <= maxY) ||
    (p2.x >= minX && p2.x <= maxX &&
     p2.y >= minY && p2.y <= maxY)) {
    return true;
  }

  // Check intersection with each of the four box edges
  let topLeft     = vec2<f32>(minX, minY);
  let topRight    = vec2<f32>(maxX, minY);
  let bottomLeft  = vec2<f32>(minX, maxY);
  let bottomRight = vec2<f32>(maxX, maxY);

  if (lineSegmentsIntersect(p1, p2, topLeft, topRight) ||
    lineSegmentsIntersect(p1, p2, topRight, bottomRight) ||
    lineSegmentsIntersect(p1, p2, bottomRight, bottomLeft) ||
    lineSegmentsIntersect(p1, p2, bottomLeft, topLeft)) {
    return true;
  }

  return false;
}
```

Listing 3.9: Box Selection: WGSL code for GPU-based box intersection.

```
private lineSegmentIntersectsBox(
  p1x: number, p1y: number, p2x: number, p2y: number,
  minX: number, maxX: number, minY: number, maxY: number
): boolean {
  if ((p1x >= minX && p1x <= maxX &&
     p1y >= minY && p1y <= maxY) ||
    (p2x >= minX && p2x <= maxX &&
     p2y >= minY && p2y <= maxY)) {
    return true;
  }

  const edges: [number, number, number, number][] = [
    [minX, minY, maxX, minY], // top
    [maxX, minY, maxX, maxY], // right
    [maxX, maxY, minX, maxY], // bottom
    [minX, maxY, minX, minY], // left
  ];

  for (const [ex1, ey1, ex2, ey2] of edges) {
    if (this.lineSegmentsIntersect(
      p1x, p1y, p2x, p2y, ex1, ey1, ex2, ey2
    )) {
      return true;
    }
  }

  return false;
}
```

Listing 3.10: Box Selection: TypeScript code for CPU-based box intersection.

```
private findSegmentRange(
  minX: number, maxX: number
): [number, number] {
  const positions = this.dimensionXPositions;
  if (positions.length < 2) {
    return [-1, -1];
  }
  let startIdx = -1;
  let endIdx = -1;
  for (let i = 0; i < positions.length - 1; i++) {
    const segMinX = Math.min(positions[i], positions[i + 1]);
    const segMaxX = Math.max(positions[i], positions[i + 1]);
    if (segMaxX >= minX && segMinX <= maxX) {
      if (startIdx === -1) {
        startIdx = i;
      }
      endIdx = i;
    }
  }
  return [startIdx, endIdx];
}
```

Listing 3.11: Line and Box Selection: TypeScript code for CPU-based multi-dimensional span identification. It returns the start and end indices for dimensional spans overlapped by the selection object.

object. For line selection, this encompasses all spans from the leftmost to the rightmost x-coordinate of the drawn line; for box selection, it includes all spans overlapped by the rectangle's width. Only polyline segments crossing these identified dimensional spans are then tested using the intersection functions.

Chapter 4

Rendering Parallel Coordinates

BenchLines supports multiple rendering methodologies for parallel coordinates, each with differing rendering models and performance characteristics.

4.1 Core Web Graphics Technologies

Implementations are provided for each of the four core web graphics technologies: SVG-DOM, Canvas2D, WebGL, and WebGPU.

4.1.1 SVG-DOM

BenchLines builds upon the SPCD3 project [Andrews et al. 2026], which used SVG-DOM as its primary rendering method. Consequently, the original SVG-DOM rendering pipeline was preserved during the development of BenchLines and adopted as a visual baseline against which other rendering approaches were compared. Visual properties such as color, line thickness, and interaction behavior were defined to mirror the SVG-DOM implementation and subsequently replicated across the remaining rendering technologies to ensure visual and functional consistency. Listing 4.1 shows an example of SVG-DOM–based rendering. This code is not used within the BenchLines project and is included solely as a reference to illustrate how such an approach may be implemented by interested developers in future work.

4.1.2 Canvas2D

The Canvas2D implementation uses the HTML5 `<canvas>` element and its `"2d"` rendering context, providing a CPU-based approach to drawing graphics. While not as performant as GPU-accelerated methods for large datasets, Canvas2D offers simplicity and broad browser compatibility. The 2d context is obtained from the canvas element, with DPR scaling applied via `setTransform()`. For the background canvas containing inactive lines, BenchLines leverages WebGL rendering, which is then rasterised onto a 2D canvas, combining simple compositing with GPU-accelerated performance. Active polylines are rendered by iterating through the dataset and constructing paths using `beginPath()`, `moveTo()`, and `lineTo()` calls, then stroking with the appropriate color and width. Listing 4.2 demonstrates this approach.

4.1.3 WebGL

The WebGL implementation in BenchLines is designed to leverage GPU acceleration for rendering large parallel coordinate datasets efficiently. The process begins with the initialisation of the WebGL context. As part of this initialisation, BenchLines first calculates the device pixel ratio (DPR), which ensures that the visualisation scales appropriately across a variety of screen resolutions and high-DPI displays.

Next, BenchLines creates the core GPU programs required for rendering. A vertex shader, shown in Listing 4.3, is used to process the position of each vertex in the dataset. A fragment shader, shown in

```
const svg = document.createElementNS("http://www.w3.org/2000/svg", "svg");
svg.setAttribute("width", canvas.width);
svg.setAttribute("height", canvas.height);
document.body.appendChild(svg);

for (const d of dataset) {
  const polyline =
    document.createElementNS("http://www.w3.org/2000/svg", "polyline");
  polyline.setAttribute("points", getPolylinePoints(d));
  polyline.setAttribute("stroke", "steelblue");
  polyline.setAttribute("stroke-width", "2");
  polyline.setAttribute("fill", "none");
  svg.appendChild(polyline);
}
```

Listing 4.1: SVG-DOM: JavaScript code for drawing polylines.

```
export function redrawCanvasLines(newDataset: any, parcoords: any) {
  if (!ctx || !canvasEl || !newDataset) return;
  dataset = newDataset;
  ctx.clearRect(0, 0, canvasEl.width, canvasEl.height);

  for (const d of newDataset) {
    const id = getLineNameCanvas(d);
    const active = lineState[id]?.active ?? true;
    if (!active) continue;

    const pts = getPolylinePoints(d, parcoords);
    if (!pts.length) continue;

    ctx.beginPath();
    ctx.moveTo(pts[0][0], pts[0][1]);
    for (let i = 1; i < pts.length; i++) {
      ctx.lineTo(pts[i][0], pts[i][1]);
    }
    ctx.lineWidth = 2;
    ctx.strokeStyle = "rgba(0,129,175,0.5)";
    ctx.stroke();
  }
  redrawHoverOverlay();
}
```

Listing 4.2: Canvas2D: TypeScript code for Canvas2D rendering of active polylines.

```
const vertexShaderSrc = `
attribute vec2 position;
attribute vec4 a_color;
uniform vec2 resolution;
varying vec4 v_color;

void main() {
  vec2 zeroToOne = position / resolution;
  vec2 clipSpace = zeroToOne * 2.0 - 1.0;
  gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
  v_color = a_color;
}
`;
```

Listing 4.3: [WebGL: TypeScript code for a vertex shader.

```
const fragmentShaderSrc = `
precision mediump float;
varying vec4 v_color;
void main() {
  gl_FragColor = v_color;
}
`;
```

Listing 4.4: WebGL: TypeScript code for a fragment shader.

Listing 4.4, handles the coloring of pixels. These shaders are compiled and then linked into a WebGL program, shown in Listing 4.5, which enables the GPU to perform the rendering operations efficiently.

Once the program is set up, BenchLines creates buffers to store the vertex positions, colors, and other attributes required for rendering each line. At this stage, BenchLines also initializes its three-layer architecture, consisting of the background canvas, the rendering canvas, and the interaction overlay canvas. Each layer serves a specific purpose: the background canvas stores rasterized inactive polylines, the rendering canvas handles displaying the active polylines, and the interaction canvas is used for hovering, selection, and other interactive controls.

Rendering the active polylines involves iterating through the dataset of polylines. Each polyline is split into individual line segments, and each segment is represented as two triangles. This conversion allows precise control over the line thickness, at a slight cost in terms of rendering performance compared to rendering as WebGL lines. The three vertices for each triangle are loaded into the vertex buffer and drawn using the WebGL program (Listing 4.5). By performing these operations on the GPU, BenchLines can efficiently render thousands of lines while maintaining interactivity, even for high-dimensional datasets. The code for drawing polylines with WebGL is shown in Listing 4.6.

```
function createProgram(
  gl: WebGLRenderingContext,
  vShader: WebGLShader,
  fShader: WebGLShader
) {
  const program = gl.createProgram();
  if (!program) throw new Error("createProgram failed");
  gl.attachShader(program, vShader);
  gl.attachShader(program, fShader);
  gl.linkProgram(program);
  if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
    console.error(gl.getProgramInfoLog(program));
    throw new Error("Program link failed");
  }
  return program;
}
```

Listing 4.5: WebGL: TypeScript code for a program.

State	Color	RGBA Values
Clear	Transparent	(0,0,0,0)
Active	Light Blue	(0.502, 0.749, 0.839, 1.0)
Inactive	Light Gray	(0.922, 0.922, 0.922, 1.0)
Hover	Red	(1.0, 0.2, 0.2, 1.0)
Selected	Orange	(1.0, 0.502, 0.0, 1.0)

Table 4.1: WebGPU: Color usage.

4.1.4 WebGPU

Similar to the WebGL implementation, WebGPU also takes advantage of the GPU to render graphics. The program initially declares the global state variables that help maintain the rendering context and interaction state. Afterwards, constants that specify the active and inactive line widths are declared.

Once a call to render the polylines is made, the navigator interface is initially used to check if a GPU is available. If yes, the process continues; else, the rendering attempt stops after throwing an error indicating the lack of a GPU. Continuing the process, a WebGPU context is acquired from the HTML canvas element. Afterwards, the context is configured to use the WebGPU device acquired before. Also, the preferred canvas format is set up, and alpha mode is declared. In this instance, alpha mode is set as premultiplied, which means that the colors put in the canvas must have their color values already multiplied by the alpha value. Furthermore, a secondary canvas positioned absolutely over the main canvas is dynamically created for rendering the hovered and selected lines. Also, a third canvas is inserted behind the main canvas to display inactive (filtered) polylines.

Listing 4.7 shows the WGSL (WebGPU Shading Language) code for the vertex and fragment shaders. The vertex shader takes 2d vertex positions as input and converts them to 4d clip-space coordinates. Meanwhile, the fragment shader simply returns the uniform color for all pixels.

Listing 4.8 defines how vertices are processed and rendered. And, the Listing 4.9 shows the creation of a vertex buffer that holds the vertex data in memory. Furthermore, the Listing 4.10 shows how the colors are managed for the rendering. Each separate color state is defined with its own uniform buffer and bind group. The Table 4.1 shows the RGBA Values of the different colors used for various line states.

```
export function redrawWebGLLines(newDataset: any[], parcoords: any) {
  if (!gl || !vertexBuffer || !colorBuffer) return;
  dataset = newDataset;

  gl.useProgram(program);
  gl.uniform2f(resolutionLoc, canvasEl.width, canvasEl.height);
  gl.clearColor(0, 0, 0, 0);
  gl.clear(gl.COLOR_BUFFER_BIT); // clears active lines canvas

  const dpr = window.devicePixelRatio || 1;
  const vertices: number[] = [];
  const colors: number[] = [];

  for (const d of newDataset) {
    const id = getLineNameCanvas(d);
    const active = lineState[id]?.active ?? true;
    if (!active) continue; // skip inactive lines
    const pts = getPolylinePoints(d, parcoords, dpr);
    if (pts.length < 2) continue;

    const color = [128 / 255, 191 / 255, 214 / 255, 1];
    const width = ACTIVE_LINE_WIDTH;

    for (let i = 0; i < pts.length - 1; i++) {
      const x1 = pts[i][0], y1 = pts[i][1];
      const x2 = pts[i + 1][0], y2 = pts[i + 1][1];
      const dx = x2 - x1;
      const dy = y2 - y1;
      const length = Math.sqrt(dx * dx + dy * dy);
      if (length === 0) continue;
      const perpX = -dy / length * (width / 2);
      const perpY = dx / length * (width / 2);

      vertices.push(x1 + perpX, y1 + perpY);
      vertices.push(x1 - perpX, y1 - perpY);
      vertices.push(x2 + perpX, y2 + perpY);
      vertices.push(x1 - perpX, y1 - perpY);
      vertices.push(x2 - perpX, y2 - perpY);
      vertices.push(x2 + perpX, y2 + perpY);

      for (let j = 0; j < 6; j++) {
        colors.push(...color);
      }
    }
  }

  const vertexData = new Float32Array(vertices);
  const colorData = new Float32Array(colors);

  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, vertexData, gl.DYNAMIC_DRAW);
  gl.vertexAttribPointer(posLoc, 2, gl.FLOAT, false, 0, 0);

  gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, colorData, gl.DYNAMIC_DRAW);
  gl.vertexAttribPointer(colorLoc, 4, gl.FLOAT, false, 0, 0);

  gl.drawArrays(gl.TRIANGLES, 0, vertexData.length / 2);

  redrawHoverOverlay();
}
```

Listing 4.6: WebGL: TypeScript code to render the active polylines.

```
// Create a new shader module on the GPU device
const cellShaderModule = device.createShaderModule({
// The label is used for debugging purposes.
label: "Shaders",
code: `
  @group(0) @binding(0) var<uniform> color: vec4<f32>;
  struct VSOut {
    @builtin(position) position : vec4<f32>,
  };
  @vertex
  fn vs_main(@location(0) pos: vec2<f32>) -> VSOut {
    var out: VSOut;
    out.position = vec4<f32>(pos, 0.0, 1.0);
    return out;
  }
  @fragment
  fn fs_main() -> @location(0) vec4<f32> {
    return color;
  }
`,
});
```

Listing 4.7: WebGPU: WGSL code for vertex and fragment shaders.

WebGPU does not natively support rendering lines with custom thickness. To overcome this, the vertices are extrapolated to form rectangles out of two triangles joined together. The Listing 4.11 shows a sample code showcasing this.

Finally, Listing 4.12 shows how polylines are rendered with WebGPU. The function begins with the Command Encoder, which is a recording interface that builds a list of GPU commands to be executed later. Then the render pass begins. This clears the canvas to transparent, prepares the GPU to receive the drawing commands and sets up the rendering pipeline state. The canvas is then set up using the dpr value obtained from the current window, so that the render is also presented well on high-resolution displays without blurriness. Afterwards, the vertices are processed based on their active or inactive state, and the data values are transformed to pixel positions using scale functions.

To display the lines in their correct thickness, 6 vertices are required for each line segment, which represents the edges of the two triangles, essentially forming one thick line. The `addVertex()` function converts the pixel coordinates to the WebGPU clip space. Then, a vertex buffer is created to allocate GPU memory for the drawing. The buffer can store vertex data and can receive data from the CPU. Next, the pipeline is configured to activate the shader programs and rendering settings configured during initialisation. The vertex buffer is then bound to slot 0, making it available to the vertex shader. From here, each polyline is given the appropriate color based on its active or inactive state using the bind groups defined before. The draw command accepts the number of vertices to draw, the number of instances, the starting position in the vertex buffer and the first instance index. Finally, the end of the render pass is marked, and the command encoder converted into an immutable command buffer is sent to the GPU for execution. The GPU processes the commands independently while the CPU continues with its other tasks.

```
pipeline = device.createRenderPipeline({
  layout: device.createPipelineLayout({
    bindGroupLayouts: [bindGroupLayout],
  }),
  vertex: {
    // GPUShaderModule containing vertex shader
    module: cellShaderModule,
    // name of function in shader code called for every vertex invocation
    entryPoint: "vs_main",
    // array of GPUVertexBufferLayout objects describing how data is packed
    buffers: [vertexBufferLayout],
  },
  fragment: {
    module: cellShaderModule,
    entryPoint: "fs_main",
    targets: [
      {
        format: canvasFormat,
        blend: {
          color: {
            // factor for source color (color being drawn)
            srcFactor: "src-alpha",
            // factor for destination color (color already in framebuffer)
            dstFactor: "one-minus-src-alpha",
            // blending operation to apply
            operation: "add",
          },
          alpha: {
            // factor for source alpha (alpha being drawn)
            srcFactor: "one",
            // factor for destination alpha (alpha already in framebuffer)
            dstFactor: "one-minus-src-alpha",
            // blending operation to apply
            operation: "add",
          },
        },
      },
    ],
  },
  primitive: {
    // we are drawing triangles
    topology: "triangle-list",
    // For pipelines with strip topologies ("line-strip" or "triangle-strip"),
    // this determines the index buffer format and primitive restart value
    stripIndexFormat: undefined,
  },
});
```

Listing 4.8: WebGPU: TypeScript code for the render pipeline.

```
const vertexBufferLayout: GPUVertexBufferLayout = {
// number of bytes tGPU needs to skip forward for next vertex
arrayStride: 8,
attributes: [
  {
    format: "float32x2" as GPUVertexFormat,

    // offset into vertex for this attribute
    offset: 0,

    // arbitrary number between 0 and 15, unique for every attribute
    shaderLocation: 0,
  } as GPUVertexAttribute,
],};
```

Listing 4.9: WebGPU: TypeScript code for vertex buffer layout.

```
// Create hover color buffer (red)
const hoverColorBuffer = device.createBuffer({
size: 16,
usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,
});
device.queue.writeBuffer(
hoverColorBuffer,
0,
new Float32Array([1.0, 51.0 / 255.0, 51.0 / 255.0, 1.0]) // Red with alpha 1.0
);
hoverBindGroup = device.createBindGroup({
layout: bindGroupLayout,
entries: [{ binding: 0, resource: { buffer: hoverColorBuffer } }],
});
```

Listing 4.10: WebGPU: TypeScript code for bind groups and uniform buffers.

```
for (const line of allLines) {
const width = line.active ? ACTIVE_LINE_WIDTH : INACTIVE_LINE_WIDTH;
for (let i = 0; i < line.pts.length - 1; i++) {
  const p1 = line.pts[i];
  const p2 = line.pts[i + 1];
  const x1 = p1[0], y1 = p1[1];
  const x2 = p2[0], y2 = p2[1];
  const dx = x2 - x1;
  const dy = y2 - y1;
  const length = Math.sqrt(dx * dx + dy * dy);
  if (length === 0) continue;
  const perpX = -dy / length * (width / 2);
  const perpY = dx / length * (width / 2);

  // Triangle 1
  addVertex(x1 + perpX, y1 + perpY);
  addVertex(x1 - perpX, y1 - perpY);
  addVertex(x2 + perpX, y2 + perpY);

  // Triangle 2
  addVertex(x1 - perpX, y1 - perpY);
  addVertex(x2 - perpX, y2 - perpY);
  addVertex(x2 + perpX, y2 + perpY);
}
}
```

Listing 4.11: WebGPU: TypeScript code for rendering lines with custom thickness.

```
export function redrawWebGPULines(newDataset: any[], parcoords: any) {
  dataset = newDataset;
  currentParcoords = parcoords;
  // Create command encoder to encode GPU commands
  encoder = device.createCommandEncoder();
  pass = encoder.beginRenderPass({
    colorAttachments: [
      {
        view: context.getCurrentTexture().createView(),
        loadOp: "clear",
        clearValue: { r: 0, g: 0, b: 0, a: 0 },
        storeOp: "store",},],});
  const dpr = window.devicePixelRatio || 1;
  const canvasWidth = canvasEl.width;
  const canvasHeight = canvasEl.height;
  const allLines: { pts: [number, number][]; active: boolean }[] = [];
  let totalVertexCount = 0;
  for (const d of newDataset) {
    const id = getLineNameCanvas(d);
    const active = lineState[id]?.active ?? true;
    if (!active) continue;
    const pts = getPolylinePoints(d, parcoords, dpr);
    if (pts.length >= 2) {
      allLines.push({ pts, active });
      // Each segment has 6 vertices (2 triangles)
      totalVertexCount += (pts.length - 1) * 6;
    }
  }
  if (totalVertexCount === 0) {
    // If no lines to draw, finish the pass and submit empty command buffer
    pass.end();
    device.queue.submit([encoder.finish()]);
    return;
  }
  const totalBufferSize = totalVertexCount * 2 * 4;
  const allVerts = new Float32Array(totalVertexCount * 2);
  let currentOffset = 0;
  function addVertex(x: number, y: number) {
    const xClip = (x / canvasWidth) * 2 - 1;
    const yClip = 1 - (y / canvasHeight) * 2;
    allVerts[currentOffset++] = xClip;
    allVerts[currentOffset++] = yClip;}
  const vertexBuffer = device.createBuffer({
    label: "all-polyline-vertices",
    size: totalBufferSize,
    usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,});
  device.queue.writeBuffer(vertexBuffer, 0, allVerts);
  pass.setPipeline(pipeline);
  pass.setVertexBuffer(0, vertexBuffer);
  let vertexOffset = 0;
  for (const line of allLines) {
    const lineVertexCount = (line.pts.length - 1) * 6;
    pass.setBindGroup(0, line.active ? activeBindGroup : inactiveBindGroup);
    pass.draw(lineVertexCount, 1, vertexOffset, 0);
    vertexOffset += lineVertexCount;}
  pass.end();
  device.queue.submit([encoder.finish()]);
}
```

Listing 4.12: WebGPU: TypeScript code for rendering polylines.

4.2 Using Web Graphics Libraries

Implementations are provided for two web graphics libraries: Three.js and Pixi.js.

4.2.1 Three.js

Three.js was used to implement both WebGL (using Three.js's WebGLRenderer) and WebGPU (using Three.js's experimental WebGPURenderer) versions of polyline rendering. Listing 4.13 shows some sample Three.js code to render a line. Depending on the underlying rendering technology needed, the renderer can be swapped. In Three.js, the scene allows the user to define what is to be rendered and where. It acts as a container for all visual elements (objects, lights, cameras) in the 3d or 2d world. A scene is only created once, and objects are added dynamically to it later. The camera defines the viewpoint and projection type. An `OrthographicCamera` is used in the example provided since, in this case, the size of the object rendered stays constant regardless of its distance from the camera. This prevents any perspective distortion. The camera decides how the scene gets projected onto the canvas. The points in the example are a flat array of coordinates [x1, y1, z1, x2, y2, z2, ...] defining the line's vertices. Finally, the `LineGeometry` specifies the vertices that form a polyline and the `Line2` is used in the given example, so that the width or thickness of the line can also be configured.

4.2.2 Pixi.js

The Listing 4.14 demonstrates example line rendering using Pixi.js. The example code uses the WebGPU renderer if it is set up (check using the navigator object), and if not, it falls back to the WebGL renderer. After the renderer is set up, the code creates a stage and a `linesContainer`, which is added to the stage. The graphics object helps to draw shapes such as rectangles, circles, or any polygons. In this case, it is used to render a polyline. The `pts` array defines the vertices of the polyline. Once all the segments are defined, line properties such as width, color and alpha are configured. Finally, the renderer renders the stage to the canvas.

```
// Shared imports and setup
import * as THREE from "three"; // For WebGL
import { Line2 } from "three/examples/jsm/lines/Line2.js"; // Shared
import { LineGeometry } from "three/examples/jsm/lines/LineGeometry.js"; // Shared
import { LineMaterial } from "three/examples/jsm/lines/LineMaterial.js";

// Renderer setup (swap based on backend)
let renderer;
if (backend === "webgl") {
  renderer = new THREE.WebGLRenderer({ canvas: canvasEl, alpha: true });
} else if (backend === "webgpu") {
  renderer = new THREE.WebGPURenderer({ canvas: canvasEl, alpha: true });
  await renderer.init();
}
renderer.setSize(width, height);
renderer.setPixelRatio(window.devicePixelRatio || 1);

// Material (swap based on backend)
let lineMaterial;
if (backend === "webgl") {
  let lineMaterial: LineMaterial | null = null;
  lineMaterial = new LineMaterial({
    color: 0x80bfd6,
    linewidth: 3,
  });
  lineMaterial.resolution.set(width, height);
} else if (backend === "webgpu") {
  let lineMaterial: THREE.Line2NodeMaterial | null = null;
  lineMaterial = new Line2NodeMaterial({
    color: 0x80bfd6,
    linewidth: 3,
    worldUnits: false,
    alphaToCoverage: true,
  });
}

// Scene and camera (shared)
const scene = new THREE.Scene();
const camera = new THREE.OrthographicCamera(0, width, height, 0, -1, 1);

// Draw a simple thick line
const geometry = new LineGeometry();
// Horizontal line across canvas
const points = [0, height / 2, 0, width, height / 2, 0];
geometry.setPositions(points);
const line = new Line2(geometry, lineMaterial);
line.computeLineDistances();
scene.add(line);

// Render
renderer.render(scene, camera);
```

Listing 4.13: Three.js: TypeScript code for rendering a line.

```
import * as PIXI from "pixi.js";
async function initPixiLine(canvas: HTMLCanvasElement) {
  const dpr = window.devicePixelRatio || 1;
  const width = canvas.clientWidth;
  const height = canvas.clientHeight;
  // Choose WebGPU if available, otherwise use WebGL
  let renderer: PIXI.Renderer | PIXI.WebGPURenderer;
  if ("gpu" in navigator) {
    const webgpuRenderer = new PIXI.WebGPURenderer();
    await webgpuRenderer.init({
      canvas,
      width,
      height,
      resolution: dpr,
      backgroundAlpha: 0,
      antialias: true,
      autoDensity: true,
      clearBeforeRender: true,
    });
    renderer = webgpuRenderer;
  } else {
    renderer = new PIXI.Renderer({
      view: canvas,
      width: width / dpr,
      height: height / dpr,
      resolution: dpr,
      backgroundAlpha: 0,
      autoDensity: true,
      clearBeforeRender: true,
    });
  }
  const stage = new PIXI.Container();
  const linesContainer = new PIXI.Container();
  stage.addChild(linesContainer);
  const graphics = new PIXI.Graphics();
  linesContainer.addChild(graphics);
  const pts: [number, number][] = [
    [50, 50],
    [150, 80],
    [250, 40],
    [350, 120],
  ];
  graphics.clear();
  if (pts.length > 0) {
    graphics.moveTo(pts[0][0], pts[0][1]);
    for (let i = 1; i < pts.length; i++) {
      graphics.lineTo(pts[i][0], pts[i][1]);}
    // Use WebGPU-style stroke API if available, else fall back to lineStyle API.
    if ("stroke" in graphics) {
      graphics.stroke({width: 3,color: 0x0081af,alpha: 1,});}
      else {
      (graphics as any).lineStyle(3, 0x0081af, 1);
    }}
  renderer.render(stage);
}
```

Listing 4.14: Pixi.js: TypeScript code for rendering polylines.

Chapter 5

Benchmarking Results

Using BenchLines, rendering time was analysed across three datasets of 10 dimensions and each of 100, 1000, and 10,000 records for 10 iterations. BenchLines subsequently reports the average rendering time. The tests were performed on a MacBook Pro with an Apple M3 Pro and 18 GB of RAM. The results can be seen in Table 5.1.

WebGPU consistently delivers the fastest overall rendering performance. This advantage stems from its modern, low-level design, which enables more efficient and direct use of GPU resources. By leveraging contemporary GPU rendering techniques, WebGPU achieves superior performance compared to the other approaches. This performance advantage comes at the cost of increased implementation complexity, as fully leveraging WebGPU requires explicit GPU programming and a strong understanding of modern graphics pipelines when compared to approaches such as SVG-DOM. Despite this trade-off, WebGPU proved to be the most efficient rendering technology for parallel coordinates visualisations.

Canvas2D and WebGL are both competitive across varying dataset sizes. This result was somewhat surprising, as WebGL was initially expected to outperform Canvas2D. However, modern browsers are highly optimized for Canvas2D in these scenarios and can effectively leverage GPU acceleration. WebGL also remains competitive in terms of performance, although the BenchLines WebGL implementation does not scale as efficiently as the library-based implementation provided by Three.js. The reduced overhead from avoiding an external library is offset at larger scales by enabling more direct and efficient WebGL rendering.

SVG-DOM, while conceptually straightforward and easy to implement, demonstrated limited scalability. It performs adequately for small visualisations, such as those containing fewer than approximately 100 polylines, but does not scale well to larger datasets. Consequently, SVG-DOM is best suited for small-scale visualisations, where simplicity and direct DOM-based interaction are prioritised over rendering performance.

For developers seeking to avoid low-level GPU programming, high-level libraries such as Pixi.js and Three.js provide abstraction layers over WebGL and WebGPU. While these libraries reduce implementation complexity, they still require familiarity with library-specific features to achieve optimal performance. Additionally, the abstraction introduces a small performance overhead compared to custom WebGPU or WebGL implementations. Overall, these libraries offer a practical compromise by combining GPU acceleration with improved developer accessibility.

#	Technology	Dataset	Iterations	Avg Time (ms)
1	WebGPU	10D_100	10	0.460
2	Canvas2D	10D_100	10	0.480
3	WebGL	10D_100	10	1.840
4	Pixi-WebGPU	10D_100	10	2.350
5	Pixi-Canvas2D	10D_100	10	2.470
6	SVG-DOM	10D_100	10	3.140
7	Three-WebGPU	10D_100	10	3.590
8	Three-WebGL	10D_100	10	3.800
9	Pixi-WebGL	10D_100	10	4.760
1	WebGPU	10D_1000	10	1.170
2	WebGL	10D_1000	10	3.240
3	Canvas2D	10D_1000	10	3.300
4	Three-WebGPU	10D_1000	10	3.630
5	Three-WebGL	10D_1000	10	5.410
6	Pixi-WebGPU	10D_1000	10	6.100
7	Pixi-Canvas2D	10D_1000	10	7.650
8	SVG-DOM	10D_1000	10	11.360
9	Pixi-WebGL	10D_1000	10	12.650
1	WebGPU	10D_10000	10	11.040
2	Canvas2D	10D_10000	10	14.270
3	Three-WebGL	10D_10000	10	21.930
4	Three-WebGPU	10D_10000	10	22.360
5	WebGL	10D_10000	10	26.730
6	Pixi-Canvas2D	10D_10000	10	39.790
7	Pixi-WebGPU	10D_10000	10	64.140
8	SVG-DOM	10D_10000	10	67.740
9	Pixi-WebGL	10D_10000	10	82.010

Table 5.1: BenchLines: Benchmark results for three different 10-dimensional datasets, with 100, 1000, and 10,000 records. Tested using a MackBook Pro, with Apple M3 Pro and 18 GB RAM. The results for each dataset are sorted in increasing order of average rendering time for all the polylines in the dataset.

Chapter 6

Future Work

BenchLines has significant potential for extension and improvement. Additional rendering libraries, such as Orillusion [Orillusion 2023], could be incorporated into the system. By taking advantage of the existing three-layer canvas architecture, additional features can be integrated with minimal system-level modifications. Benchmarking capabilities could also be expanded beyond rendering performance to include interactive features, such as hover behavior, using the existing JavaScript and WebGPU-based hover algorithms. Further implementation of customizable interaction features, including rendering color, line thickness, and other visual properties, would enhance both usability and flexibility.

Accessibility improvements represent another avenue for future work, including the implementation of color blind friendly palettes and alternative color options. As well as the ability to upload or generate custom-sized datasets for further comparison capabilities. Together, these enhancements would broaden the utility of BenchLines as both a benchmarking tool and a framework for developing interactive web-based visualisations.

Chapter 7

Concluding Remarks

BenchLines was developed to evaluate and compare web-based rendering technologies within a parallel coordinates visualisation. The system enables the comparison of common rendering methods as well as widely used graphics libraries, providing both quantitative performance assessment and qualitative evaluation of interaction features. By separating background, rendering, and interaction layers, BenchLines allows consistent benchmarking across different technologies while maintaining the interactive features expected of a parallel coordinate system.

Based on the evaluation, WebGPU offers the highest performance, making it the preferred choice when implementation complexity is not a consideration and the target audience supports modern GPU-enabled browsers. For scenarios where low-level GPU programming is not desired, libraries such as Three.js and Pixi.js provide abstraction over WebGL or WebGPU, offering a balance between performance and ease of use, albeit with a minor overhead and the need for library-specific knowledge. Canvas2D remains a practical alternative, delivering competitive performance while providing a simpler and more accessible programming model.

Bibliography

Aircada [2024]. *Picking Sides in WebGL vs Canvas*. 2024. https://aircada.com/blog/webgl-vs-canvas (cited on page 3).

Anderson, Michael, Jyothish Atheendran, and Filip Ljubotina [2026]. *BenchLines: Benchmarking Interactive Web Graphics for Parallel Coordinates*. Graz University of Technology, 03 Feb 2026. https://github.com/filip-ljubotina/BenchLines (cited on page 1).

Andrews, Keith, Romana Gruber, Philipp Drescher, Jeremias Kleinschuster, Sebastian Schreiner, and Burim Vrella [2026]. *Steerable Parallel Coordinates in D3*. Graz University of Technology, 03 Feb 2026. https://github.com/tugraz-isds/spcd3 (cited on pages 1, 19).

Apple [2026]. *Metal*. Apple Developer Documentation, 2026. https://developer.apple.com/documentation/metal (cited on page 5).

Cabello, Ricardo [2026]. *three.js: JavaScript 3D Library*. 03 Feb 2026. https://github.com/mrdoob/three.js (cited on page 5).

Danchilla, Brian [2012]. *Three.js Framework*. In: *Beginning WebGL for HTML5*. Berkeley, CA: Apress, 2012, pages 173–203. ISBN 1430239972. doi:10.1007/978-1-4302-3997-0_7 (cited on page 5).

Deveria, Alexis [2025]. *WebGL– 3D Canvas graphics*. 11 Oct 2025. https://caniuse.com/webgl (cited on page 4).

Deveria, Alexis [2026]. *Can I use WebGPU?* 2026. https://caniuse.com/webgpu (cited on page 5).

Gilbert, Kelsey [2025]. *WebGL 2.0 Specification*. 07 Feb 2025. https://registry.khronos.org/webgl/specs/latest/2.0/ (cited on page 4).

Gruber, Romana and Keith Andrews [2026]. *Parallel Coordinates: An Interactive Tutorial*. 2026. https://tugraz-isds.github.io/pcee/ (cited on page 3).

Han, Yudong, Weichen Bi, Ruibo An, Deyu Tian, Qi Yang, and Yun Ma [2025]. *GL2GPU: Accelerating WebGL Applications via Dynamic API Translation to WebGPU*. Proc. The ACM Web Conference (WWW 2025) (Sydney, Australia). 28 Apr 2025, pages 751–762. doi:10.1145/3696410.3714785 (cited on page 5).

Inselberg, Alfred [2009]. *Parallel Coordinates: Visual Multidimensional Geometry and Its Applications*. Springer, 08 Oct 2009. 554 pages. ISBN 0387215077 (cited on page 3).

Khronos [2011]. *Khronos Releases Final WebGL 1.0 Specification*. 03 Mar 2011. https://khr.io/ba (cited on page 4).

Larabel, Michael [2023]. *Firefox 110 Released With Better WebGL Performance, GPU-Accelerated 2D Canvas*. 14 Feb 2023. https://www.phoronix.com/news/Firefox-110-Released (cited on page 3).

Lecamus, Jocelyn [2026]. *What's New in Three.js (2026): WebGPU, New Workflows & Beyond*. Utsubo, 10 Jan 2026. https://utsubo.com/blog/threejs-2026-what-changed (cited on page 5).

Malyshau, Dzmitry and Kai Ninomiya [2021]. *WebGPU*. W3C Working Draft. First Public Working Draft. W3C, May 2021. https://w3.org/TR/2021/WD-webgpu-20210518/ (cited on page 5).

MDN [2025]. *WebGL: 2D and 3D Graphics for the Web*. Mozilla Developer Network, 15 Jul 2025. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (cited on page 4).

Microsoft [2026]. *Direct3D 12 programming guide*. Microsoft Learn, 2026. https://learn.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide (cited on page 5).

Mo, Zhenyao [2017]. *WebGL 2.0 Arrives*. 27 Feb 2017. https://khronos.org/blog/webgl-2.0-arrives (cited on page 4).

Orillusion [2023]. *Orillusion*. 2023. https://github.com/Orillusion/orillusion (cited on page 35).

PixelFree [2024]. *WebGL vs. Canvas: Which is Better for 3D Web Development?* 2024. https://blog.pixelfreestudio.com/webgl-vs-canvas-which-is-better-for-3d-web-development/ (cited on page 3).

Pixi [2026]. *PixiJS Repository*. 03 Feb 2026. https://github.com/pixijs/pixijs (cited on page 5).

Taivalsaari, Antero, Tommi Mikkonen, Cesare Pautasso, and Kari Systä [2017]. *Comparing the Built-In Application Architecture Models in the Web Browser*. Proc. International Conference on Software Architecture (ICSA 2017) (Gothenburg, Sweden). IEEE, 03 Apr 2017, pages 51–54. doi:10.1109/ICSA.2017.23. https://si.usi.ch/assets/publications/conf/icsa/icsa2017/TaivalsaariMPS17.pdf (cited on page 3).

Vulkan [2026]. *Vulkan Documentation*. 03 Feb 2026. https://docs.vulkan.org/spec/latest/ (cited on page 5).

Wallez, Corentin [2016]. *Explicit Web Graphics API*. Presentation to the WebGL Working Group. 08 Jun 2016. https://docs.google.com/presentation/d/1FkfvYgwlFmdrx8GhqomFgL0B0ppb6LNXBPT9DyQE0qY/ (cited on page 5).

Weitz, Dario [2020]. *Parallel Coordinates Plots*. 27 Jul 2020. https://medium.com/data-science/parallel-coordinates-plots-6fcfa066dcb3 (cited on page 3).

Wikipedia [2025]. *SVG*. 07 Dec 2025. https://wikipedia.org/wiki/SVG (cited on page 4).