# Fast Interactive Web Graphics

Michael Anderson, Jyothish Atheendran, Petra Cukrov, and Filip Ljubotina

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

15 Dec 2025

## Abstract

Interactive web applications often rely on high-performance graphics for visualization, simulation, and sophisticated user interfaces. As the complexity of web-based tools grows, developers face an increasing need to render more and more complex scenes to the user. This survey provides an overview of four basic graphics rendering technologies in the web browser: Canvas2D, SVG-DOM, WebGL, and WebGPU. In addition, Three.js and Pixi.js, two widely-used graphics rendering libraries are described, which build upon the basic graphics rendering technologies

These technologies were evaluated using BenchLines, a benchmarking tool developed to measure rendering times in a polyline drawing scenario. BenchLines allows users to select a rendering method, configure a dataset, and specify the number of trials, after which it reports the average rendering performance. The result aim to help guide developers in choosing appropriate rendering solutions and to inform future design decisions for high-performance, graphics-intensive applications.

# Contents

# List of Figures

# List of Tables

# List of Listings

# Chapter 1

# Introduction

Today, there are essentially four basic ways to draw graphics in the web browser: Canvas2D, SVG-DOM, WebGL, and WebGPU. Each technology has its own benefits and drawbacks.

Canvas2D refers to the HTML `<canvas>` element and its `"2d"` rendering context. It enables raster-based JavaScript API for 2d drawing. The `<canvas>` element was originally introduced by Apple in 2004 and was later standardized in HTML5. SVG-DOM refers to the possibility of using JavaScript to dynamically insert Scalable Vector Graphics (SVG) nodes into the browser's Document Object Model (DOM), causing them to be rendered. SVG is a vector graphics markup language with elements like `<rect>` and `<text>`. Firefox introduced partial support for SVG in 2006, and support was again standardized in HTML5. WebGL is a JavaScript API based on OpenGL ES, which enables hardware-accelerated 3d rendering on a GPU (where available), with a fallback to software rendering. However, this introduces significant extra complexity in terms of coding. Finally, WebGPU is the successor to WebGL, with improved performance, lower overhead, and exposure to more modern GPU capabilities.

Alongside the four native web graphics technologies, higher-level web graphics libraries have emerged to simplify the rendering process and offer alternatives to using the base technologies directly. Three.js abstracts the complexities of doing 3d graphics in the browser, exposing a scene graph system to accomplish the rendering. Pixi.js leverages WebGL to create an easy-to-use library for 2D graphics rendering.

This survey provides an overview of these technologies and libraries. In addition, a benchmarking application, BenchLines, was developed to allows users to evaluate the rendering performance of the technologies and libraries. It repeatedly draw a set of polylines with a given graphics technology or library and chosen number of trials, and reports the average rendering time. By combining an overview of the current state of these rendering technologies with the quantitative results from the benchmarking tool, this survey aims to inform developers about the benefits and limitations of each approach and help them choose an appropriate rendering stack for their own applications.

# Chapter 2

# Web Graphics Technologies

The four basic web graphics technologies are Canvas2D, SVG-DOM, WebGL, and WebGPU. Each technology has its own benefits and drawbacks, which are presented in the following sections, alongside code examples.

## 2.1 Canvas2D

The Canvas 2D 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, Canvas 2D 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 Canvas 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. Listing 2.1 shows some simple JavaScript code for drawing polylines with Canvas2D.

Performance-wise, Canvas 2D is mostly CPU-based, and performance degrades with complex or high-volume drawing operations [PixelFree 2024]. For applications requiring thousands of objects or real-time 3d rendering, WebGL and WebGPU provide significantly better performance by leveraging GPU acceleration.

## 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. Listing 2.2 shows some simple JavaScript code for drawing polylines with SVG-DOM.

According to MDN [2025a], the SVG standard was in development within the World Wide Web Consortium (W3C) since 1999, emerging from a recognized need for vector graphics on the web. SVG 1.0 became a W3C Recommendation on 4 September 2001, followed by SVG 1.1 which became a W3C Recommendation on 14 Jan 2003. The specification continued to evolve, with SVG 1.1 Second Edition released on 16 Aug 2011. SVG 2 reached the Candidate Recommendation stage on 15 Sep 2016, and revised versions were published on 07 Aug 2018 and 04 Oct 2018. Browser support for SVG 2 is feature-based rather than version-based, and no modern browser fully implements the entire SVG 2 spec.

```
function initCanvas2D(dpr: number) {
  ctx = canvasEl.getContext("2d")!;
  ctx.setTransform(dpr, 0, 0, dpr, 0, 0);
  return ctx;
}

function redrawCanvasLines(dataset, parcoords) {
  for (const d of dataset) {
    const pts = getPolylinePoints(d, parcoords);
    ctx.beginPath();
    ctx.moveTo(pts[0][0], pts[0][1]);
    pts.slice(1).forEach(p => ctx.lineTo(p[0], p[1]));
    ctx.lineWidth = 2;
    ctx.strokeStyle = (lineState[getLineName(d)]?.active ?? true)
    ? "rgba(0,129,175,0.5)" : "rgba(211,211,211,0.4)";
    ctx.stroke();
  }
}
```

**Listing 2.1:** Canvas2D: JavaScript code for drawing polylines.

```
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 2.2:** SVG-DOM: JavaScript code for drawing polylines.

## 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 2025b] 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 2025b], meaning users do not need to enable anything to be able to utilize WebGL.

### 2.3.1 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.

### 2.3.2 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.

```
const vertexShaderSrc = `
attribute vec2 position;
attribute vec4 a_color;
uniform vec2 resolution;
varying vec4 v_color;
void main() {
  gl_Position = vec4((position / resolution * 2.0 - 1.0) * vec2(1,-1), 0, 1);
  v_color = a_color;
}`;
```

**Listing 2.3:** WebGL: An example vertex shader.

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

**Listing 2.4:** WebGL: An example fragment shader.

### 2.3.3 WebGL Usage

WebGL uses vertex and fragment shaders to determine what is displayed on the screen. The vertex shader is responsible for computing the position of each vertex of a shape, such as a triangle, in screen space [WebGLFundamentals 2025]. An example vertex shader is provided in Listing 2.3. The fragment shader then determines the color and other attributes of each fragment (essentially, each pixel) that makes up the final image [Skinner 2021]. An example fragment shader is provided in Listing 2.4.

These shaders operate using buffers, which are blocks of memory passed from the CPU to the GPU. Buffers can contain vertex positions, colors, texture coordinates, and other attributes that the shaders use during rendering. The vertex shader runs first, transforming vertex positions into normalized device coordinates (ranging from -1 to 1 on each axis) and optionally passing additional data to the fragment shader. The fragment shader then uses this information to compute the final color of each pixel.

Once written, the vertex and fragment shaders are compiled into a program and sent to the GPU along with the data buffers. An example initialization process is provided in Listing 2.5. The GPU executes the shader program for each vertex and fragment, efficiently generating the final rendered image. Shaders are a central feature of WebGL because they allow developers to implement custom visuals, manipulate geometry, and take full advantage of the GPU. With the shaders and program created, developers may use it to render content into the scene by passing data to the buffers and using a WebGL function such as `drawArrays()` to render the content to the scene. Listing 2.6 shows some example WebGL code to render polylines.

```
function initCanvasWebGL(){
  gl = canvas.getContext("webgl");
  const vShader = createShader(gl, gl.VERTEX_SHADER, vertexShaderSrc);
  const fShader = createShader(gl, gl.FRAGMENT_SHADER, fragmentShaderSrc);
  program = createProgram(gl, vShader, fShader);

  gl.viewport(0, 0, canvasEl.width, canvasEl.height);

  vertexBuffer = gl.createBuffer();
  colorBuffer = gl.createBuffer();

  return gl;
}
```

**Listing 2.5:** WebGL: Initializing WebGL 1.0 and linking the shaders.

```
function redrawWebGLLines(dataset: any[], parcoords: any) {
  gl.useProgram(program);
  gl.clear(gl.COLOR_BUFFER_BIT);

  // Loop through dataset points creating an array of vertices...

  const vertexData = new Float32Array(vertices);

  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.drawArrays(gl.LINES, 0, vertexData.length / 2);
}
```

**Listing 2.6:** WebGL: JavaScript code to draw polylines with WebGL 1.0.

## 2.4 WebGPU

WebGPU is a JavaScript API that enables web pages to perform computations and render graphics using the GPU [W3C 2025a]. The WebGPU project is maintained by the W3C GPU for the Web Community Group, which was launched on February 16, 2017.

After the initial release of WebGL, several new native GPU APIs and designs appeared (eg, Microsoft's Direct3D 12, Apple's Metal, and The Khronos Group's Vulkan) that WebGL failed to adapt to [Ninomiya et al. 2025; MDN 2025c]. The inability of WebGL to render demanding 3d graphics by tapping into the more advanced features of modern GPUs, along with its lack of general-purpose GPU (GPGPU) compute functionalities, eventually led to the creation of WebGPU.

WebGPU brings with it several advantages that makes it the new top contender in web-based graphics rendering. Primarily, it exposes access to more advanced GPU features which helps in taking advantage of modern hardware capabilities. For a developer who has expertise in graphics programming, WebGPU offers tremendous scope for optimizing rendering pipelines and memory access, which ultimately yields

```
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();

context = canvasEl.getContext("webgpu");
const canvasFormat = navigator.gpu.getPreferredCanvasFormat();
context.configure({
device,
format: canvasFormat,
// the colors must have their values already multiplied by the alpha value.
alphaMode: "premultiplied",
});
```

**Listing 2.7:** WebGPU: JavaScript code to initialize WebGPU.

a highly performant graphics rendering. Parallel computations using compute shaders also helps in extracting better performance from the underlying hardware.

### 2.4.1 WebGPU Adoption

As of 15 Sep 2025, WebGPU is supported in all modern browsers [Deveria 2025a]. However, support in Linux is not enabled by default. Han et al. [2025] experimented with a dynamic intermediate translation layer that can convert WebGL invocations from web applications into WebGPU commands. Even after considering several limitations like translation overhead, the absence of refined caching policies, the lack of direct feature counterparts of WebGL in WebGPU, and so on, they were still able to observe around 45% average frame time reduction after running experiments on several devices. This highlights the performance potential that WebGPU holds in comparison to WebGL. Furthermore, libraries like Three.js, Pixi.js, Babylon.js, and Orillusion provide support for WebGPU, making it easier to use.

### 2.4.2 WebGPU Usage

WebGPU uses a <canvas> element with a "webgpu" rendering context, which is set up with the optimal or a specific pixel format. This format defines how colors are stored and displayed. This setup is necessary before issuing any rendering commands. Listing 2.7 shows some JavaSCript code to initialize WebGPU.

Shaders in WebGPU are written using the WebGPU Shading Language (WGSL) [W3C 2025b]. As in the case of WebGL, a vertex shader transforms vertex data (typically in 2d or 3d) into clip space, while a fragment shader decides the final color each pixel on the screen. Listing 2.8 shows an example. A vertex buffer layout is used to instruct the GPU as to how to interpret the vertex data. This layout defines the structure of each vertex including the size of a single vertex and attributes like format, offset and shader location. See Listing 2.9 for an example.

A render pipeline is created to encapsulate the details of the layout and the shaders to be used, as shown in Listing 2.10. A render pass encoder is created to encode commands and also to set the properties that should take effect during the render pass, as shown in Listing 2.11. Finally, the rendering commands stored in the command encoder are finalized into a command buffer and submitted to the GPU queue for execution, as shown in Listing 2.12.

```
const shaderModule = device.createShaderModule({
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 2.8:** WebGPU: Code defining example vertex and fragment shaders.

```
const vertexBufferLayout: GPUVertexBufferLayout = {
  // Size of a single vertex in bytes (2 floats x 4 bytes each)
  arrayStride: 8,
  attributes: [
  {
    // Attribute format: 2 floats per vertex (x, y)
    format: "float32x2" as GPUVertexFormat,
    // Offset within the vertex
    offset: 0,
    // Shader location to link this attribute in the vertex shader
    shaderLocation: 0,
  } as GPUVertexAttribute,
  ],
};
```

**Listing 2.9:** WebGPU: JavaScript Code to define a vertex buffer layout.

```javascript
const pipeline = device.createRenderPipeline({
layout: device.createPipelineLayout({ bindGroupLayouts: [bindGroupLayout] }),
vertex: { module: shaderModule, entryPoint: "vs_main",
   buffers: [vertexBufferLayout] },
fragment: {
  module: shaderModule,
  entryPoint: "fs_main",
  targets: [{
    format: canvasFormat,
    blend: {
      color: { srcFactor: "src-alpha", dstFactor: "one-minus-src-alpha" },
      alpha: { srcFactor: "one", dstFactor: "zero" }
    }
  }]
},
primitive: { topology: "line-strip" }, });
```

**Listing 2.10:** WebGPU: JavaScript code to define a pipeline.

```javascript
const encoder = device.createCommandEncoder();
const pass = encoder.beginRenderPass({
colorAttachments: [{
  view: context.getCurrentTexture().createView(),
  loadOp: "clear",
  clearValue: { r: 0, g: 0, b: 0, a: 0 },
  storeOp: "store",
}],
});
```

**Listing 2.11:** WebGPU: JavaScript code to define a render pass.

```javascript
pass.setPipeline(pipeline);
pass.setVertexBuffer(0, vertexBuffer);

let offset = 0;
for (const line of allLines) {
  pass.setBindGroup(0, line.active ? activeBindGroup : inactiveBindGroup);
  pass.draw(line.pts.length, 1, offset, 0);
  offset += line.pts.length;
}
pass.end();
device.queue.submit([encoder.finish()]);
```

**Listing 2.12:** WebGPU: JavaScript code to draw polylines.

# Chapter 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. Two such libraries are presented here: Three.js for 3d graphics and Pixi.js for 2d graphics.

## 3.1 Three.js

Three.js, first released in 2010, has grown into one of the most influential JavaScript libraries for web-based 3d rendering [Strasburger 2019]. It is fully open source, licensed under MIT, and actively maintained with a large community on GitHub. Three.js works in all modern browsers, requires no plugins, and offers a very approachable API.

As described by Csipkay [2025], Three.js was built to make WebGL accessible. WebGL is incredibly powerful, but it is also low-level, meaning developers have to manually handle shaders, buffers, and the entire rendering pipeline. Three.js abstracts all of that away, offering an API that allows developers to create 3d scenes without having to write WebGL code.

Today, Three.js supports both WebGL and WebGPU, thanks to its WebGPURenderer. It consists of many components that are used to render different web graphics, with three main components being scene, camera, and renderer. The scene holds everything that will appear on screen, including objects, lights, and cameras. The camera determines how the scene is viewed, and as Vuta [2023] explains, Three.js offers several camera types, including PerspectiveCamera, OrthographicCamera, CubeCamera, ArrayCamera, and StereoCamera. The renderer is responsible for drawing the scene onto the canvas element within the web page.

Other important components include Geometry, which defines the shape and structure of 3d objects, and Materials, which determine how those objects appear by controlling properties such as color, texture, reflectivity, and transparency. Three.js also provides a flexible system of Lights to illuminate the scene, as well as a robust Animation System that manages object movement and transitions over time.

## 3.2 Pixi.js

Pixi.js is an open-source JavaScript library focused on high-performance 2d rendering [Pixi 2025a]. It is actively maintained on GitHub [Pixi 2019]. Similar to Three.js, Pixi.js provides multi-platform support and runs on a wide range of devices without requiring major code adaptations. Its primary advantage is a clean and intuitive API that allows developers to easily create interactive 2D scenes and animations. Pixi.js uses a hierarchical display tree, where objects are organized in parent–child relationships, making scene management flexible and efficient. The library also offers native support for click and touch events, enabling the creation of interactive elements on the web. The foundational visual component in Pixi.js is the Sprite, an image or texture that can be placed, rotated, scaled, and otherwise manipulated on the screen [Pixi 2025b].

One of Pixi.js's key strengths is its compatibility with older devices that do not support WebGL. The library includes an integrated Canvas fallback, which automatically switches the renderer to HTML5 Canvas when WebGL is unavailable, without requiring any changes to the codebase.

Pixi.js is lightweight, highly optimized for performance, and has great support for interactivity such as zooming. However, Pixi.js sometimes has issues rendering lines and text, which can appear fuzzy. It is widely used in 2d games, visualizations, interactive interfaces, and applications requiring reliable, high-speed 2d rendering.

# Chapter 4

# BenchLines

BenchLines is a web application designed for benchmarking various web graphics technologies in a polyline drawing scenario. In particular, BenchLines simulates the drawing of polylines in a parallel coordinates visualization. This chapter describes the design and structure of the BenchLines application, outlining how its components work together to benchmark different web-based graphics technologies.

## 4.1 Design

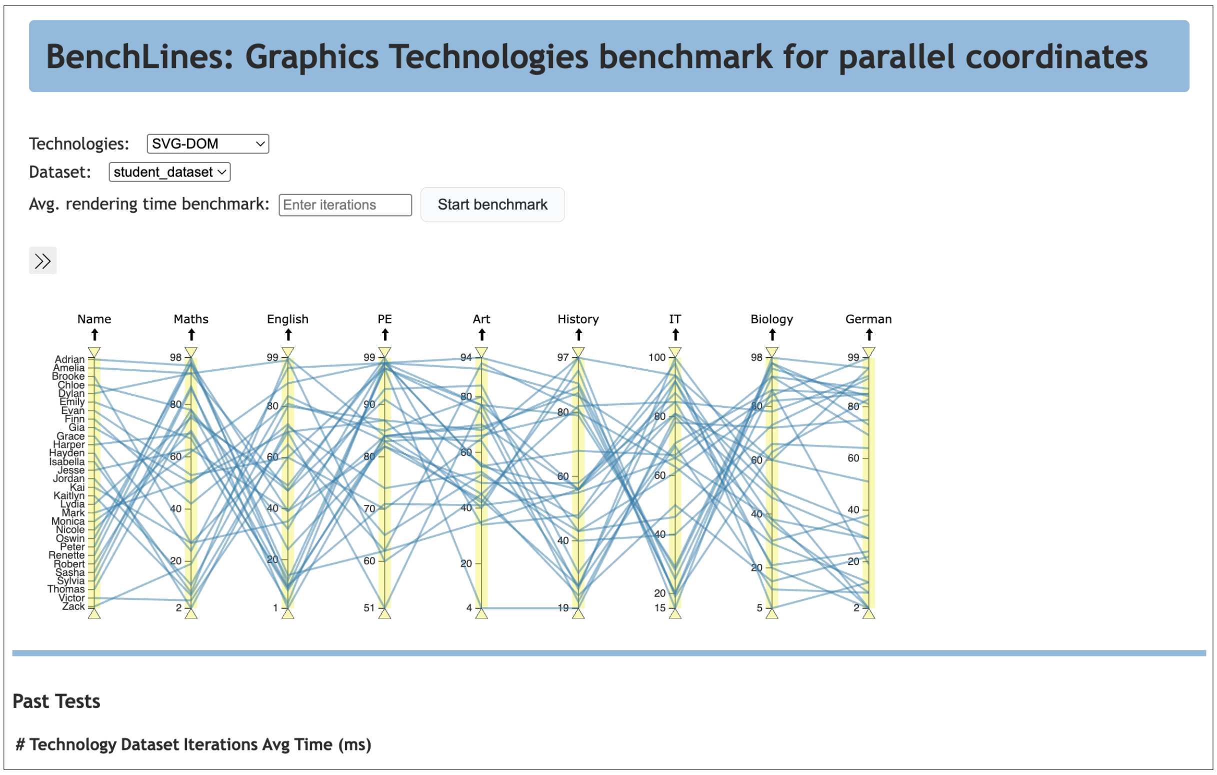
The BenchLines application has three main sections: an Input panel, an Interactive Chart, and a Results Table that displays benchmark test outcomes. These can be seen in Figure 4.1. In the Input panel at the top, the user first select a specific web graphics technology, including library implementations. Next, the user selects a dataset. Multiple datasets are available for testing, including a real-world student dataset and six synthetic datasets comprising 10 and 20 dimensions and each of 100, 1000, and 10,000 records. Finally, the user enters the desired number of iterations and starts the trial.

The time required to re-render all polylines over the specified number of iterations is measured and the avaerage render time per iteration is calculated. The results of each trial are appended to the Results Table at the bottom for easy reference and copying. As an alternative, it is possible to manually adjust the filter control on one of the dimensions by sliding the yellow triangle up or down. This provides an intuitive sense of rendering smoothness during polyline updates.

## 4.2 Results

The average render time for three datasets of 10 dimensions and each of 100, 1000, and 10,000 records can be seen in Table 4.1. WebGPU consistently delivers the fastest rendering overall. Its advantage comes from its modern, low-level design. A strong pattern emerges when comparing native implementations to library-based ones. Native versions are reliably faster. Abstraction layers used in frameworks like Pixi.js or Three.js add convenience, but introduce processing and management overhead, which becomes more noticeable as data complexity increases. At the lower end of performance, SVG-DOM consistently ranks among the slowest options. This is due to the heavy cost of maintaining a DOM node for each graphical element, which scales poorly as the number of rendered shapes grows.

Surprisingly, the results show Canvas2D outperforming the WebGL 1.0 implementation for both small and large datasets. This is surprising, as WebGL utilizes the GPU and was expected to be the second most performant technology. Even though WebGL is generally capable of higher performance for large, complex scenes, modern browsers have extremely optimized 2d pipelines. This allows for Canvas2D to sometimes outperform WebGL in 2d contexts [Autar 2023]. For small datasets, Canvas2D avoids GPU setup overhead entirely and runs through highly streamlined CPU-side rendering paths, making it unexpectedly but consistently faster than WebGL 1.0.



BenchLines: Graphics Technologies benchmark for parallel coordinates

Technologies: SVG-DOM

Dataset: student_dataset

Avg. rendering time benchmark: Enter iterations Start benchmark

Past Tests

# Technology Dataset Iterations Avg Time (ms)

**Figure 4.1:** BenchLines: The user interface of the benchmarking web application. It simulates drawing polyline for a parallel coordinates visualization. [Screenshot taken by Filip Ljubotina.]

| # | 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 | WebGPU-Pixi | 10D_100 | 10 | 2.350 |
| 5 | Canvas2DPixi | 10D_100 | 10 | 2.470 |
| 6 | SVG-DOM | 10D_100 | 10 | 3.140 |
| 7 | WebGPU-Three | 10D_100 | 10 | 3.590 |
| 8 | WebGLThree | 10D_100 | 10 | 3.800 |
| 9 | WebGLPixi | 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 | WebGPU-Three | 10D_1000 | 10 | 3.630 |
| 5 | WebGLThree | 10D_1000 | 10 | 5.410 |
| 6 | WebGPU-Pixi | 10D_1000 | 10 | 6.100 |
| 7 | Canvas2DPixi | 10D_1000 | 10 | 7.650 |
| 8 | SVG-DOM | 10D_1000 | 10 | 11.360 |
| 9 | WebGLPixi | 10D_1000 | 10 | 12.650 |
| 1 | WebGPU | 10D_10000 | 10 | 11.040 |
| 2 | Canvas2D | 10D_10000 | 10 | 14.270 |
| 3 | WebGLThree | 10D_10000 | 10 | 21.930 |
| 4 | WebGPU-Three | 10D_10000 | 10 | 22.360 |
| 5 | WebGL | 10D_10000 | 10 | 26.730 |
| 6 | Canvas2DPixi | 10D_10000 | 10 | 39.790 |
| 7 | WebGPU-Pixi | 10D_10000 | 10 | 64.140 |
| 8 | SVG-DOM | 10D_10000 | 10 | 67.740 |
| 9 | WebGLPixi | 10D_10000 | 10 | 82.010 |

**Table 4.1:** BenchLines: Benchmark results for three different 10-dimensional datasets, with 100, 1000, and 10,000 records. The results for each dataset are sorted in increasing order of average rendering time for all the polylines in the dataset.

# Chapter 5

# Concluding Remarks

This survey paper presented an overview of four basic web graphics technologies (Canvas2D, SVG-DOM, WebGL, and WebGPU) and two higher-level web graphics libraries (Three.js and Pixi.js). The BenchLines benchmarking tool was developed to evaluate the performance of the various graphics technologies in a controlled and repeatable way, in the context of drawing polylines for a parallel coordinates visualization.

WebGPU was the most performant core technology, consistently outperforming the other approaches. Among the frameworks, Three.js delivered the best results for larger datasets, and Pixi.js performed the best for small datasets. Canvas2D performed better than expected, particularly for larger datasets. In the future, BenchLines could be extended to support additional frameworks and core technologies, as well as to implement a native WebGL 2.0 version. This would allow for a more direct comparison between WebGL 1.0, WebGL 2.0, and the frameworks built on top of the technology.

# Bibliography

Aircada [2024]. *Picking Sides in WebGL vs Canvas*. 05 Apr 2024. https://aircada.com/blog/webgl-vs-canvas (cited on page 3).

Autar, Avishkar [2023]. *A look at 2D vs WebGL canvas performance*. 16 Jan 2023. https://semisignal.com/a-look-at-2d-vs-webgl-canvas-performance/ (cited on page 13).

Csipkay, Peter [2025]. *Three.js Facts*. 2025. https://threejsresources.com/facts (cited on page 11).

Deveria, Alexis [2025a]. *Can I use: WebGPU*. 11 Oct 2025. https://caniuse.com/webgpu (cited on page 8).

Deveria, Alexis [2025b]. *WebGL– 3D Canvas graphics*. 11 Oct 2025. https://caniuse.com/webgl (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 5).

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 8).

Khronos [2011]. *Khronos Releases Final WebGL 1.0 Specification*. 03 Mar 2011. https://khr.io/ba (cited on page 5).

MDN [2025a]. *Introducing SVG from Scratch*. Mozilla Developer Network, 07 Dec 2025. https://developer.mozilla.org/docs/Web/SVG/Tutorials/SVG_from_scratch/Introduction (cited on page 3).

MDN [2025b]. *WebGL: 2D and 3D Graphics for the Web*. Mozilla Developer Network, 15 Jul 2025. https://developer.mozilla.org/docs/Web/API/WebGL_API (cited on page 5).

MDN [2025c]. *WebGPU API*. Mozilla Developer Network, 30 Nov 2025. https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API (cited on page 7).

Mo, Zhenyao [2017]. *WebGL 2.0 Arrives*. 27 Feb 2017. https://khronos.org/blog/webgl-2.0-arrives (cited on page 5).

Ninomiya, Kai, Corentin Wallez, and Dzmitry Malyshau, editors [2025]. *WebGPU Explainer*. GPU for the Web Community Group, 09 Dec 2025. https://gpuweb.github.io/gpuweb/explainer/ (cited on page 7).

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 [2019]. *PixiJS Repository*. 2019. https://github.com/pixijs/pixijs (cited on page 11).

Pixi [2025a]. *Pixi.js*. 2025. https://pixijs.com/8.x/guides/getting-started/intro (cited on page 11).

Pixi [2025b]. *Sprite*. 2025. https://pixijs.com/8.x/guides/components/scene-objects/sprite (cited on page 12).

Skinner, Jesse [2021]. *Introduction to WebGL and Shaders*. 15 Sep 2021. https://codingwithjesse.com/blog/introduction-to-webgl-and-shaders/ (cited on page 6).

Strasburger, Oskar [2019]. *Getting started with WebGL using Three.js*. 2019. https://merixstudio.com/blog/getting-started-webgl-using-threejs (cited on page 11).

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).

Vuta, Gopisaikrishna [2023]. *Exploring Cameras in Three.js*. 2023. https://medium.com/@gopisaikrishna.vuta/exploring-cameras-in-three-js-32e268a6bebd (cited on page 11).

W3C [2025a]. *WebGPU API and WebGPU Shading Language (WGSL) specifications*. W3C GPU for the Web Working Group, 09 Dec 2025. https://github.com/gpuweb/gpuweb (cited on page 7).

W3C [2025b]. *WebGPU Shading Language*. 07 Dec 2025. https://w3.org/TR/WGSL/ (cited on page 8).

WebGLFundamentals [2025]. *WebGL Shaders and GLSL*. 26 Feb 2025. https://webglfundamentals.org/webgl/lessons/webgl-shaders-and-glsl.html (cited on page 6).