Leveraging WebGPU for Real-Time Machine Learning Visualizations in the Browser

PhD Alen Šimec[1], PhD Lidija Tepeš Golubić[2]

[1]University of Applied Sciences Zagreb, Vrbik 8, Croatia
Email: *alen.simec[at]tvz.hr*

[2]University of Applied Sciences Zagreb, Vrbik 8, Croatia
Email: *lidija.tepes[at]tvz.hr*

Abstract: ***This paper investigates the potential of WebGL and WebGPU for real-time scientific visualization in web browsers. By implementing comparable rendering tasks using both APIs, it evaluates their respective strengths in terms of implementation complexity, rendering performance, and scalability. WebGL, while accessible and widely supported, shows limitations in handling high-performance or parallelized tasks. WebGPU, despite its steeper learning curve, offers granular control and native compute-shader support, positioning it as a forward-looking alternative for computation-intensive web applications. Through controlled benchmarks and graphical comparisons, the study provides insights into selecting the appropriate API based on performance needs, development effort, and application complexity.***

Keywords: WebGL, WebGPU, scientific visualization, browser graphics, GPU computing

1. Introduction

Graphics programming in the context of the web has seen tremendous evolution over the past decade, transitioning from basic 2D rendering to complex real-time 3D scenes. This evolution has enabled interactive and immersive content directly within browsers, without the need for additional plugins or software. The demand for rich, responsive, and visually engaging applications has driven the development of new technologies that allow closer integration with modern GPU hardware. This is especially relevant in fields like scientific visualization, where the clarity and accuracy of visual representation are paramount. WebGL emerged as the first widely adopted standard for enabling GPU-accelerated graphics in web browsers. Built on OpenGL ES 2.0, WebGL allows developers to create 2D and 3D visualizations using GLSL shaders, making it a powerful tool for a range of web-based visual applications. One of the strengths of WebGL is its simplicity and compatibility across browsers, which has led to its widespread use in fields like game development, education, and interactive media. It serves as an entry point for many developers exploring graphics programming in web environments.

Despite its strengths, WebGL has limitations that stem from its foundational architecture. It abstracts away much of the GPU's capabilities, which can hinder performance and scalability in complex scenarios involving large datasets or advanced computations. To address these limitations, WebGPU was introduced as a modern graphics API designed for the web. Inspired by native APIs such as Vulkan, Metal, and Direct3D 12, WebGPU provides lower-level access to GPU resources and more granular control over rendering and compute tasks. WebGPU marks a significant shift in the way web graphics are handled, requiring developers to explicitly define GPU pipelines, buffers, and resource layouts. This increased complexity brings with it the potential for better optimization and performance.

The key innovation of WebGPU lies in its native support for compute shaders functions that allow parallel processing directly on the GPU. This capability opens the door to scientific simulations, machine learning visualizations, and large-scale data processing in the browser.

As scientific research and digital communication increasingly rely on interactive visualizations, the tools used to develop these representations must evolve to meet new performance and flexibility demands.

This paper investigates the practical differences between WebGL and WebGPU by implementing identical graphical tasks using both APIs and analyzing their performance, structure, and usability. Through a series of controlled tests, including the rendering of thousands to millions of graphical objects, the study examines how each API handles workload distribution, memory management, and frame rendering under stress. These findings are particularly relevant for developers and researchers working in communication science, where the effectiveness of visual data delivery is essential for knowledge dissemination and audience engagement. Scientific visualization is not merely about aesthetics, it is about conveying complex information in an accessible, accurate, and engaging way. The technologies powering these visualizations must therefore support high fidelity and real-time interaction.

WebGL, while capable of handling moderate levels of graphical complexity, begins to show its age when pushed to the limits of modern hardware potential, especially in scenarios requiring parallel computations. WebGPU's architecture, by contrast, aligns more closely with current GPU hardware and software trends, making it a forward-looking choice for next-generation scientific and educational tools.

Although WebGPU introduces a steeper learning curve, its adoption is growing, driven by the increasing need for more performant and flexible solutions in web-based graphics programming. In the context of communication science, the ability to deliver complex ideas visually clearly and efficiently is becoming a core requirement, particularly in an era where digital media dominates. This research contributes to the understanding of how emerging technologies like WebGPU can elevate the practice of scientific communication by enabling better, faster, and more interactive visual tools. By comparing WebGL and WebGPU through both qualitative and quantitative analysis, this study provides insights that can inform future development decisions in academic, educational, and applied settings. Ultimately, the paper argues that while WebGL remains a practical choice for many applications, WebGPU represents the future of browser-based graphics particularly in scenarios demanding high performance and scientific accuracy. The study holds significance for developers, educators, and researchers seeking to leverage high-performance web technologies for interactive scientific communication.

2. WebGPU: A modern successor to WebGL

WebGL, despite its success and broad adoption, relies on an older GPU programming model. While it abstracts memory management and simplifies GPU communication, it lacks support for many features of modern GPU architectures. This becomes particularly evident in scenarios that require advanced compute operations or sophisticated memory control.

As web-based applications become more complex and demanding, developers require tools that offer deeper access to GPU capabilities. This has led to the development of WebGPU, a next-generation API that addresses the limitations of WebGL and brings web graphics programming closer to the metal. WebGPU is designed to be a more efficient, flexible, and powerful alternative. Inspired by contemporary APIs such as Vulkan, Metal, and Direct3D 12, it provides low-level access to the GPU while remaining accessible from within the browser. Unlike WebGL, which conceals much of the underlying GPU operations, WebGPU empowers developers to manage memory and resources explicitly. This leads to more optimized and high-performance applications, albeit with increased development complexity.

One of the core differences lies in how rendering pipelines are handled. In WebGPU, the user must explicitly define all aspects of the rendering process, including buffers, shaders, and state configurations. This facilitates precise control over rendering processes. The WebGPU programming model encourages a more structured and predictable workflow. Developers must create and manage command encoders, pipelines, and resource bindings manually, mimicking native GPU programming practices. While this approach increases the learning curve, it unlocks powerful capabilities not available in WebGL. Notably, WebGPU supports compute shaders, enabling direct execution of parallel computations on the GPU. Compute shaders in WebGPU make it possible to perform tasks like matrix multiplication, particle simulation, and real-time physics calculations entirely on the GPU, without falling back to CPU-based JavaScript operations. Such features make WebGPU an ideal tool for scientific computing, machine learning, data visualization, and other domains where high-performance computation is essential. Memory management in WebGPU is more hands-on. Developers allocate and manage GPU buffers explicitly, which reduces hidden overhead and enables fine-tuned optimization strategies. This explicit memory control allows applications to minimize latency and maximize throughput, which is crucial for rendering large datasets or real-time simulations in the browser. Another key innovation is WebGPU's resource binding model. Instead of implicit bindings, developers define bind groups that clearly map GPU resources to shader inputs, making the relationship between code and hardware behavior transparent. WebGPU also introduces enhanced error handling and validation layers that help developers identify issues early in the rendering pipeline setup. This improves debugging and leads to more stable applications. Although still in the early stages of adoption, WebGPU has been integrated into modern browsers like Chrome and Firefox, with experimental support also available in Edge and Safari. Developers working with WebGPU typically use frameworks or helper libraries to simplify boilerplate code. However, for maximum performance, many prefer to work directly with the API to retain full control over rendering logic. WebGPU is not just about graphics. Its support for general-purpose GPU computation (GPGPU) brings high-performance parallelism to web applications, closing the gap between native and browser-based tools. The shift to WebGPU represents a broader trend in web development toward high-performance, low-level APIs that allow the browser to compete with native desktop environments. This has significant implications for fields such as data journalism, scientific visualization, engineering simulation, and real-time 3D rendering, where performance and precision are critical. Despite its benefits, WebGPU also introduces challenges. The API is more verbose, harder to learn, and less forgiving than WebGL. Proper understanding of GPU pipelines, memory allocation, and synchronization is required. To support adoption, communities are developing tutorials, sample projects, and documentation. Over time, the learning curve is expected to flatten, making WebGPU accessible to a broader range of developers. In educational contexts, WebGPU is beginning to replace WebGL in curricula focused on graphics programming, especially in computer science and engineering programs. Long-term, WebGPU is poised to become the standard for web-based graphics and computation. Its alignment with modern hardware trends ensures that applications built with it will be future-proof and highly scalable. While WebGL will continue to serve as a practical solution for simpler projects, WebGPU offers the depth and control necessary for the next generation of web applications.

WebGPU transforms the way developers approach GPU programming in the browser. By exposing more of the hardware and demanding greater precision, it opens the door to high-performance, real-time, and computationally intensive experiences on the web.

3. Objectives and hypotheses

The primary objective of this paper is to compare WebGL and WebGPU within the context of graphics programming in web browsers, using basic practical examples. The focus is on evaluating ease of use, code structure, API flexibility, and overall performance during typical rendering tasks.

This research aims to provide insight into how these two APIs perform under similar conditions and how their design philosophies affect developer experience and output efficiency. By implementing equivalent graphical tasks, the study seeks to illustrate both conceptual and technical differences in their respective approaches to web-based GPU programming.

The following hypotheses are proposed:
- WebGPU offers superior performance and more advanced features compared to WebGL, especially in demanding scenarios involving large-scale rendering or computational workloads.
- Simple graphical tasks, such as rendering a triangle, clearly highlight the implementation complexity differences, providing a useful baseline for comparison.
- Although WebGPU introduces greater initial complexity, it offers significantly higher potential for advanced applications, including parallel computations and large-scale data visualizations.

These hypotheses will be tested through hands-on implementation and performance benchmarking using a unified test environment.

4. Methodology

To compare WebGL and WebGPU, a set of basic rendering examples was developed, focusing on the display of a static triangle. These examples were implemented using HTML and JavaScript, and testing was conducted in a web browser with WebGPU support (Google Chrome).

The implementation process included the setup of an HTML canvas, initialization of the appropriate graphics API, definition of the triangle's geometry and colors, creation of vertex and fragment shaders, and finally, rendering the scene on screen.

For both APIs, the following aspects were analyzed:
- The amount of code required,
- Implementation complexity,
- Clarity of structure,
- And basic performance metrics.

This methodology enables a direct comparison between the two interfaces by maintaining consistent conditions and tasks for both implementations.

4.1 Difference in operational approach

To illustrate the differences in initialization and basic usage between WebGL and WebGPU, a simple rendering example a static triangle is used. Although both APIs produce the same visual result, their coding approaches and structural organization differ significantly.

WebGL employs an older model based on the OpenGL ES 2.0 specification, making it relatively easy to learn and use. Initialization involves defining vertex and fragment shaders, creating a program, setting up a single data buffer, and issuing draw calls. Most steps are straightforward, and the API abstracts much of the underlying hardware complexity. Because of its simplicity, WebGL is often the preferred choice for beginners and rapid prototyping, allowing developers to achieve visual output with a minimal amount of code.

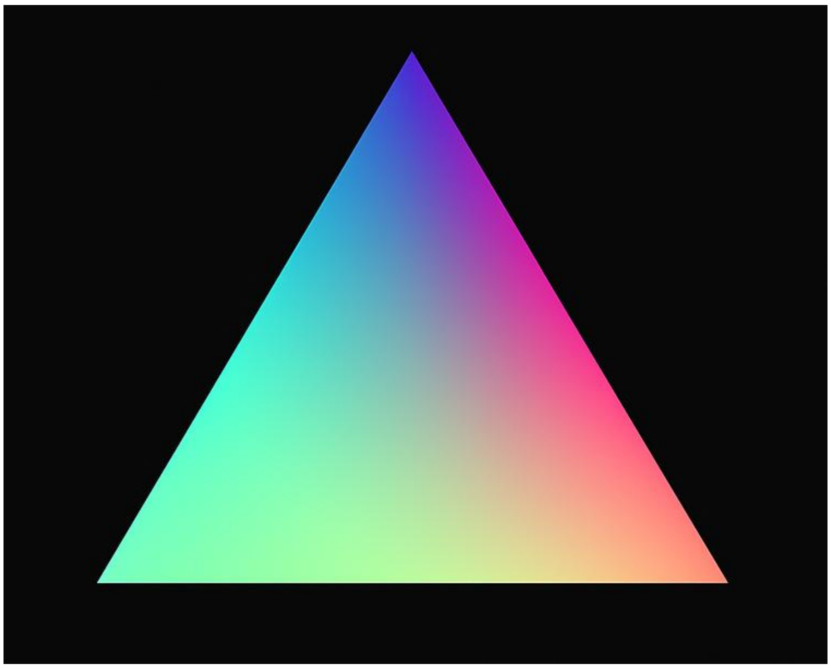
WebGPU, in contrast, introduces a more modern and explicit architecture that aligns closely with contemporary GPU design. Its initialization process requires manual definition of several key components: the device, the canvas context, the rendering pipeline (including the configuration of all draw stages), and the GPU data buffer. Additionally, rendering commands must be constructed and submitted explicitly using a command encoder. While this results in more lines of code, it also offers greater flexibility and efficiency especially for complex or graphics-intensive applications.

This distinction in complexity becomes evident even in a simple triangle rendering task. WebGL allows for a quicker and more accessible entry point, whereas WebGPU lays the groundwork for deeper control and optimization, at the cost of higher initial complexity.

```
const gl = document.getElementById('webgl').getContext('webgl');

const vs = `
attribute vec2 pos;
attribute vec2 uv;
varying vec2 vUv;
void main() {
   gl_Position = vec4(pos, 0.0, 1.0);
   vUv = uv;
}
`;

const fs = `
precision mediump float;
varying vec2 vUv;
void main() {
   gl_FragColor = vec4(vUv, 0.5, 1.0);
}
`;
```

```
function shader(type, src) {
   const s = gl.createShader(type);
   gl.shaderSource(s, src);
   gl.compileShader(s);
   return s;
}

const prog = gl.createProgram();
gl.attachShader(prog, shader(gl.VERTEX_SHADER, vs));
gl.attachShader(prog, shader(gl.FRAGMENT_SHADER, fs));
gl.linkProgram(prog);
gl.useProgram(prog);

const buf = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buf);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([
    0.0,  0.5, 0, 1,
   -0.5, -0.5, 0, 0,
    0.5, -0.5, 1, 0,
]), gl.STATIC_DRAW);

const pos = gl.getAttribLocation(prog, 'pos');
const uv  = gl.getAttribLocation(prog, 'uv');
gl.enableVertexAttribArray(pos);
gl.enableVertexAttribArray(uv);
gl.vertexAttribPointer(pos, 2, gl.FLOAT, false, 16, 0);
gl.vertexAttribPointer(uv, 2, gl.FLOAT, false, 16, 8);

gl.clearColor(0.1, 0.1, 0.1, 1);
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

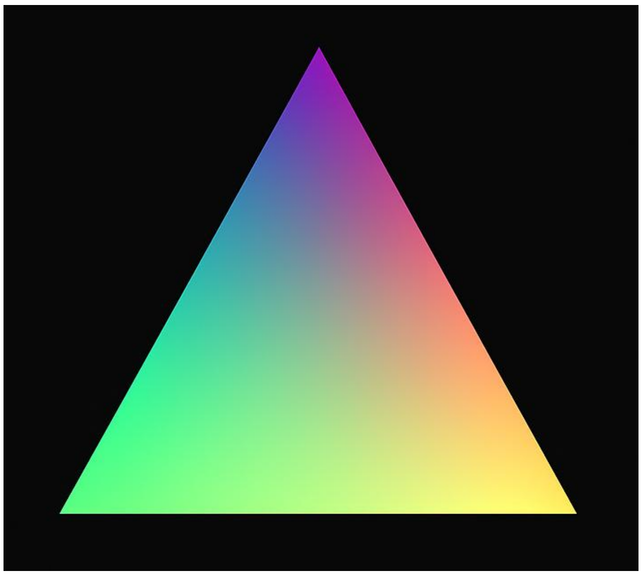
Code 1: UV triangle using the WebGL API



Picture 1: UV triangle using the WebGL API

```
(async () => {
   const adapter = await navigator.gpu?.requestAdapter();
   if (!adapter) return;

   const device = await adapter.requestDevice();
   const canvas = document.getElementById('webgpu');
   const ctx = canvas.getContext('webgpu');
   const fmt = navigator.gpu.getPreferredCanvasFormat();
   ctx.configure({ device, format: fmt });

   const code = `
   struct VertexOutput {
     @builtin(position) pos: vec4f,
     @location(0) uv: vec2f,
   };

   @vertex
   fn vs(@location(0) pos: vec2f, @location(1) uv: vec2f) -> VertexOutput {
     var output: VertexOutput;
     output.pos = vec4f(pos, 0, 1);
     output.uv = uv;
     return output;
   }

   @fragment
   fn fs(@input: VertexOutput) -> @location(0) vec4f {
     return vec4f(@input.uv, 0.5, 1);
   }
   `;

   const module = device.createShaderModule({ code });
   const pipeline = device.createRenderPipeline({
     layout: 'auto',
     vertex: {
       module,
       entryPoint: 'vs',
       buffers: [{
         arrayStride: 16,
         attributes: [
           { shaderLocation: 0, offset: 0, format: 'float32x2' },
           { shaderLocation: 1, offset: 8, format: 'float32x2' },
         ],
       }],
     },
     fragment: {
       module,
       entryPoint: 'fs',
       targets: [{ format: fmt }],
     },
     primitive: { topology: 'triangle-list' },
   });

   const buf = device.createBuffer({
     size: 48,
     usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST
   });

   device.queue.writeBuffer(buf, 0, new Float32Array([
      0.0,  0.5, 0, 1,
     -0.5, -0.5, 0, 0,
      0.5, -0.5, 1, 0,
   ]));

   const encoder = device.createCommandEncoder();
   const pass = encoder.beginRenderPass({
     colorAttachments: [{
       view: ctx.getCurrentTexture().createView(),
       clearValue: { r: 0.1, g: 0.1, b: 0.1, a: 1 },
       loadOp: 'clear',
       storeOp: 'store',
     }],
   });

   pass.setPipeline(pipeline);
   pass.setVertexBuffer(0, buf);
   pass.draw(3);
   pass.end();

   device.queue.submit([encoder.finish()]);
 })();
```

Code 2: UV triangle using the WebGPU API



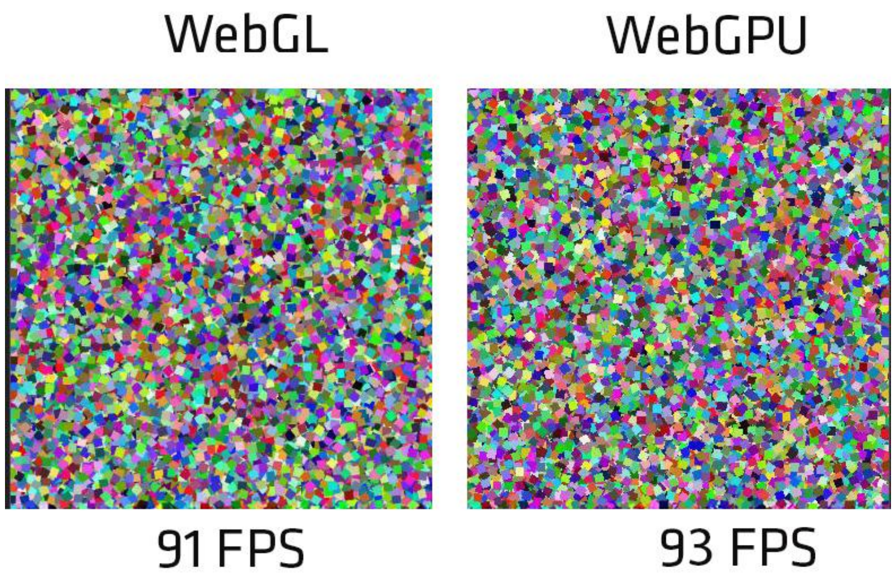
Picture 2: UV triangle using the WebGPU API

4.2 Performance differences

WebGL and WebGPU differ not only in terms of usage and API design but also in their performance, particularly when it comes to certain types of computational tasks. WebGPU enables more direct access to modern GPUs and introduces substantial advantages in areas that require intensive data processing.

One of the most significant benefits of WebGPU is its native support for compute shaders. While WebGL must rely on indirect methods, such as rendering to textures, to perform complex operations, WebGPU can execute parallel algorithms directly on the GPU. This capability is particularly valuable for data processing, physical simulations, and applications in machine learning.
In traditional graphics tasks, as demonstrated in the performance examples later in this paper, the difference between WebGL and WebGPU is often minor. Depending on the level of optimization and specific implementation, either API may perform slightly better in simple scenarios.

However, the real strength of WebGPU becomes evident in high-demand use cases, such as image processing, complex particle systems with interactivity, or large-scale matrix operations. In such contexts, the ability to run compute-intensive workloads natively on the GPU provides a major advantage over WebGL, which lacks this functionality altogether.



Picture 3: Performance difference when instancing and moving a large number of objects

4.3 Performance difference when instancing and moving a large number of objects

The benchmark was executed in Google Chrome with WebGPU support on a system equipped with an NVIDIA GeForce RTX 4070S and an AMD Ryzen 7 7800X3D. The canvas resolution was 400×400 px, runs lasted 10 seconds, and the metric reported was average Frames Per Second (FPS). These conditions apply to both WebGL and WebGPU, ensuring a like-for-like comparison.

The scene instanced and animated a large number of small quads (“squares”), scaling the object count across trials. Results were recorded for 10,000; 20,000; 500,000; 1,000,000; 2,000,000; and 5,000,000 objects.

At lower instance counts, WebGL showed a small advantage (e.g., 10k: 4851.5 FPS vs. 4723.8; 20k: 4680.2 vs. 4315.8). At mid range the APIs were near parity (500k: 430.7 vs. 427.8; 1M: 220.9 vs. 215.7). At higher loads, WebGPU pulled ahead (2M: 108.2 vs. 102.5; 5M: 42.4 vs. 38.7).

The crossover is consistent with each API’s design. WebGL’s older, higher-level model can be slightly quicker to get pixels on screen when the workload is light, but WebGPU’s explicit control over resources and pipeline state scales better as instance counts grow and CPU–GPU coordination becomes the bottleneck.

The paper also illustrates a side-by-side visual of the instanced scene where WebGPU edges WebGL by a small FPS margin reinforcing that differences in classic raster workloads can be modest and implementation-dependent. Larger gains are expected in compute-heavy scenarios thanks to WebGPU’s native compute shaders, which WebGL lacks.

For large-scale instancing and motion of many objects, both APIs can deliver high throughput; WebGPU tends to win as the problem size grows or when more granular control over memory and scheduling matters. The strongest advantages of WebGPU emerge in workloads that offload parallel computation to the GPU (image processing, particle interactions, matrix ops), beyond what WebGL can natively support.

4.4 Measurements

Test system specifications and measurement environment:
- Graphics card: NVIDIA GeForce RTX 4070S
- Processor: AMD Ryzen 7 7800X3D
- Browser: Google Chrome (with WebGPU support)
- Canvas size: 400 × 400 px
- Measurement duration: 10 seconds
- Measurement type: Average FPS (Frames Per Second)

Results of the instancing benchmark with moving squares

Table 1: Results of measurements for instancing and moving a large number of objects.

#	Object Count	WebGL FPS (AVG)	WebGPU FPS (AVG)
1.	10,000	4851.5	4723.8
2.	20,000	4680.2	4315.8
3.	500,000	430.7	427.8
4.	1,000,000	220.9	215.7
5.	2,000,000	102.5	108.2
6.	5,000,000	38.7	42.4

At lower instance counts, WebGL shows a slight advantage, while at higher instance counts the trend reverses in favor of WebGPU. The difference depends on the specific implementation and optimization level of both approaches.

5. Conclusion

This study demonstrates that WebGL and WebGPU occupy complementary positions in browser-based graphics. WebGL despite its legacy origins remains a pragmatic choice for rapid prototyping and modest visual workloads due to its lower entry barrier and broad, stable support. In contrast, WebGPU’s explicit control over pipelines, buffers, and resource binding, together with native compute-shader capability, enables more efficient utilization of modern GPU architectures. The empirical pattern is consistent: differences on classic raster tasks are modest, but advantages for WebGPU grow with workload scale and computational intensity.

Several limitations qualify these findings. Measured performance reflects a specific hardware–browser configuration and a narrow set of workloads; implementation details and optimization strategies can shift the balance in either direction. This work did not examine energy efficiency, memory pressure under extreme data sizes, or cross-browser parity over time factors that matter in production environments and on mobile devices.

For scientific visualization and communication, the implications are clear. When interactive exploration of large datasets, image processing, particle interactions, or matrix operations is required, WebGPU’s compute path and granular memory control provide a materially stronger foundation for responsiveness and fidelity. For instructional demonstrations, simplified dashboards, or cases requiring broad compatibility, WebGL continues to suffice.

Future research should
- Extend benchmarks to real application scenarios and mixed graphics compute pipelines,
- Analyze portability and performance variability across vendors and browsers,
- Assess energy/performance trade-offs, and
- Evaluate how higher-level libraries (e.g., Three.js, Babylon.js, wgpu-based tooling) can make something less severe, harmful, or difficult for WebGPU’s initial complexity while preserving its performance benefits.

As the ecosystem matures, WebGPU is well positioned to become the dominant standard for advanced, computation-aware web graphics.

Disclosure of interest
The author declares no competing interests related to the content of this article.

Declaration of Funding
No funding was received. This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

References

[1] Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M., & Hillaire, S. (2018). Real-Time Rendering (4th ed.). CRC Press.
[2] Cozzi, P. (Ed.). (2015). WebGL Insights. CRC Press. (Open PDF available.) Taylor & Francis
[3] Hansen, C. D., & Johnson, C. R. (Eds.). (2005). The Visualization Handbook. Academic Press.
[4] Wilke, C. O. (2019). Fundamentals of Data Visualization. O'Reilly Media. (Author's web edition available.); O'Reilly Media
[5] Chickerur, S., Balannavar, S., Hongekar, P., Prerna, A., & Jituri, S. (2024). WebGL vs. WebGPU: A performance analysis for Web 3.0. Procedia Computer Science, 233, 919–928. https://doi.org/10.1016/j.procs.2024.03.281
[6] Yu, J., Qin, R., & Xu, Z. (2025). The implementation of a WebGPU-based volume rendering framework for interactive visualization of ocean scalar data. Applied Sciences, 15(5), 2782. https://doi.org/10.3390/app15052782
[7] Marrinan, T., Moeller, M., Kanayinkal, A., Mateevitsi, V. A., & Papka, M. E. (2024). VisAnywhere: Developing multi-platform scientific visualization applications. arXiv:2404.17619.
[8] Sung, N.-J., Ma, J., Kim, T., Choi, Y.-j., & Hong, M. (2025). Real-time cloth simulation using WebGPU: Evaluating limits of high-resolution. arXiv:2507.11794.
[9] University of Innsbruck. (2024/2025). WebGPU evaluation: A comprehensive analysis of WebGPU as a compute platform (Master's thesis).
[10] Umeå University. (2022). Evaluation of the performance of WebGPU in a cluster of web-browsers (Master's thesis).
[11] Blekinge Institute of Technology. (2024). A comparison of performance on WebGPU and WebGL (Godot rendering backend) (Bachelor's thesis).