KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Media Technology

Second cycle, 30 credits

Performance Comparison of WebGPU and WebGL for 2D Particle Systems on the Web

An analysis of GPU time in web-based graphics APIs

SAGA PALMÉR

Stockholm, Sweden, 2024

Performance Comparison of WebGPU and WebGL for 2D Particle Systems on the Web

An analysis of GPU time in web-based graphics APIs

SAGA PALMÉR

Degree Programme in Media Technology
Date: November 17, 2024

Supervisors: Björn Thuresson, Edvin Ardö
Examiner: Tino Weinkauf
School of Electrical Engineering and Computer Science
Host company: Bontouch AB
Swedish title: Prestandajämförelse av WebGPU och WebGL för 2D-partikelsystem på webben
Swedish subtitle: En analys av GPU-tid i webbaserade grafik-API:er

© 2024 Saga Palmér

Abstract

This thesis investigates the comparative performance of WebGPU and WebGL in the context of 2D web-based particle systems, which lies in the area of web graphics and GPUs. WebGPU represents the next generation of web graphics API and introduces a more modern design than WebGL, including compute shaders to facilitate general-purpose computations on the GPU. However, the API is still under development and lacks full cross-platform support which makes it a relatively unexplored area of research. This study tests the performance of these APIs using particle systems entirely run on the GPU in a Google Chrome Canary browser, measuring total GPU time, render and compute time per frame, and initialization time over varying numbers of particles, sizes, and types of particles on two different GPUs, a high-end NVIDIA GeForce RTX 3080 and a lower-end GPU, Intel(R) UHD Graphics 620. The findings of the tests show that WebGPU significantly outperforms WebGL, particularly on the high-end GPU, where the updating of particle positions every frame is reduced by approximately 100 times using WebGPU over WebGL. Even on the lower-end GPU, the compute time is improved by 5 to 6 times with WebGPU. The maximum number of particles at 60 fps using WebGPU on the high-end GPU is about 37 and 20 million, depending on the type of particle. For WebGL, it is about 2.7 and 2.3 million. On the lower-end GPU, it is about 2.1 million and 398 000 for WebGPU, whereas, for WebGL, it is around 374 000 and 310 000. The results highlight the significant performance benefits that WebGPU offers and quantify the improvements compared to WebGL, which contributes to a broader understanding and insights into WebGPU as an upcoming cross-platform web graphics API.

Keywords

WebGPU, WebGL, Web Graphics, GPU, Graphics API, Particle Systems

Sammanfattning

Detta examensarbete undersöker den jämförande prestandan hos WebGPU och WebGL i samband med 2D webbaserade partikelsystem, som ligger inom området webbgrafik och GPU:er. WebGPU representerar nästa generations webbgrafik-API och introducerar en modernare design än WebGL, inclusive ”compute shaders” för att underlätta generella beräkningar på GPU:n. API:et är dock fortfarande under utveckling och saknar fullständigt plattformsoberoende stöd vilket gör det till ett relativt outforskat forskningsområde. Den här studien testar prestandan för dessa API:er med hjälp av partikelsystem som helt körs på GPU:n i en Google Chrome Canary webbläsare och mäter total GPU-tid, rendering och beräkningstid per bildruta och initialiseringstid över varierande antal partiklar, storlekar och typer av partiklar på två olika GPU:er, en starkare NVIDIA GeForce RTX 3080 och en mindre stark, Intel(R) UHD Graphics 620. Resultaten av testerna visar att WebGPU avsevärt överträffar WebGL, särskilt på den starkare GPU:n, där uppdateringen av partikelpositioner varje bildruta reduceras cirka 100 gånger med WebGPU över WebGL. Även på den mindre starka GPU:n förbättras beräkningstiden med 5 till 6 gånger med WebGPU. Det maximala antalet partiklar vid 60 fps med WebGPU på den avancerade GPU:n är cirka 37 och 20 miljoner, beroende på typen av partikel. För WebGL handlar det om 2,7 och 2,3 miljoner. På den mindre starka GPU:n är det maximala antalet partiklar vid 60 fps för WebGPU 2,1 miljoner och 398 000, medan det för WebGL är cirka 374 000 och 310 000. Resultaten belyser de betydande prestandafördelar som WebGPU erbjuder och kvantifierar förbättringarna jämfört med WebGL, vilket bidrar till en bredare förståelse och insikter om WebGPU som ett kommande plattformsoberoende webbgrafik-API.

Nyckelord

WebGPU, WebGL, Webbgrafik, GPU, Graphics API, Partikelsystem

Acknowledgments

I want to show my appreciation and thank everyone who helped me throughout this thesis. First, I would like to thank my supervisor at Bontouch, Edvin Ardö, for his continuous encouragement, mentorship, and feedback throughout the journey. I also want to thank everyone else at Bontouch who supported me in every possible way and ensured I had all the necessary equipment. I would also like to thank my supervisor at KTH, Björn Thuresson, for his quick feedback and guidance throughout the period. I also want to thank my examiner, Tino Weinkauf, for his expertise and thoughtful feedback. Lastly, I thank my brother Vidar Palmér, who let me borrow his computer for testing.

Stockholm, November 2024
Saga Palmér

Contents

1 Introduction **1**
1.1 Background 1
1.2 Research question 2
1.2.1 Performance metrics 3
1.3 Purpose 3
1.3.1 Sustainability and social considerations 3
1.4 Goals 4
1.5 Delimitations 4
1.6 Structure of the thesis 5

2 Background **7**
2.1 GPU programming 7
2.1.1 Computer graphics 7
2.1.2 Computer architecture 8
2.1.3 Graphics Processing Unit 8
2.1.4 Graphics APIs 9
2.1.5 Shaders 10
2.1.6 Particle systems 10
2.2 WebGL and WebGPU 11
2.2.1 WebGL 12
2.2.2 WebGPU 13
2.2.3 Shader languages - GLSL and WGSL 15
2.2.3.1 GLSL 15
2.2.3.2 WGSL 16
2.3 Related work 16
2.3.1 WebGPU versus WebGL 16
2.3.2 Related work within particle systems 17
2.4 Summary 18

3 Method **19**
3.1 Research process . . . 19
3.2 Experiment design . . . 19
3.3 Implementation . . . 20
3.3.1 Initialization of particle data . . . 20
3.3.2 Storing the data on the GPU . . . 21
3.3.3 Update positions . . . 22
3.3.4 Rendering points and squares . . . 23
3.3.5 Adding texture to the squares . . . 25
3.4 Benchmarking . . . 25
3.4.1 Initialization time . . . 25
3.4.2 Timing the GPU each frame . . . 26
3.4.2.1 Timing the GPU in WebGPU . . . 26
3.4.2.2 Timing the GPU in WebGL . . . 26
3.4.3 Automated tests . . . 27
3.5 Hardware . . . 27
3.5.1 Issues with different hardware . . . 27
3.5.2 Chosen hardware specifications . . . 27
3.5.3 Motivation for chosen hardware . . . 28
3.6 Data collection . . . 28
3.6.1 Tests . . . 28
3.6.2 Testing process . . . 29
3.7 Data analysis . . . 29

4 Results and Analysis **31**
4.1 Impact of number of particles . . . 31
4.1.1 Maximum number of particles . . . 31
4.1.2 Total GPU time per frame . . . 32
4.1.3 Frames per second . . . 33
4.1.3.1 NVIDIA GeForce RTX 3080 . . . 34
4.1.3.2 Intel(R) UHD Graphics 620 . . . 34
4.1.3.3 Difference between high-end and low-end GPU . . . 34
4.1.4 Breakdown of total time per frame . . . 35
4.1.4.1 WebGL . . . 35
4.1.4.2 WebGPU . . . 36
4.1.4.3 Comparison between WebGL and WebGPU 37
4.2 Impact of size . . . 38
4.2.1 Analysis of quadratic regression models . . . 38

4.2.2 Frames per second 39
4.3 Initialization times 40
4.3.1 Number of particles 40
4.3.2 Varying size 41

5 Discussion **43**
5.1 Frames per seconds as measurement 44
5.2 Benefits of compute shader 45
5.3 Other benefits with WebGPU 45
5.4 Initialization time as measurement 46
5.5 Application in real-world scenarios 47
5.6 Future for WebGPU 47
5.7 Method criticism 47
5.8 Future work 48

6 Conclusions **51**

References **53**

A Additional plots **61**

List of Figures

2.1 The graphics pipeline in OpenGL inspired by [11] 8
2.2 Simplified WebGL Architecture inspired by [30, 38] 11
2.3 The WebGPU Architecture inspired by [7] 15

3.1 Point particle system application 24
3.2 Square particle system application 24
3.3 Textured squares particle system application 25

4.1 Plots for average GPU frame time by number of particles . . . 32
4.2 Comparison compute and render times in WebGL 35
4.3 Comparison compute and render times in WebGPU 36
4.4 Average GPU frame time at fixed particle count by side length of square 39

A.1 Initialization Time Results 61
A.2 Point Particle system Results 62
A.3 Square Particle system Results 63
A.4 Point Particle system Results 64

List of Tables

3.1 Hardware specification strong GPU 27
3.2 Hardware specification weak GPU 28

4.1 Linear coefficients of GPU time with varying number of particles and ratios between WebGL and WebGPU 33
4.2 Maximum number of particles at 60 fps for different tests . . . 33
4.3 Ratios between WebGPU and WebGL of the maximum number of particles remaining at 60 fps on Nvidia GeForce RTX 3080 34
4.4 Ratios between WebGPU and WebGL of the maximum number of particles remaining at 60 fps on Intel(R) UHD Graphics 620 34
4.5 Ratios between the results on NVIDIA GeForce RTX 3080 and Intel(R) UHD Graphics 620 for WebGPU and WebGL of the maximum number of particles remaining at 60 fps 35
4.6 Compute and render coefficients and ratios in WebGL 36
4.7 Compute and render coefficients and ratios in WebGPU 37
4.8 Coefficient ratios between WebGL and WebGPU for rendering and compute time 37
4.9 Quadratic regression models for time (ms) with varying side length of squares $(y = ax^2 + bx + c)$ 39
4.10 Max square sizes for a fixed number of particles at 60 fps . . . 40
4.11 Linear regression models for time (ms) with varying amounts of millions of point particles $(y = kx + m)$ 41

Listings

2.1 Initializing WebGL inspired by [41] 13
2.2 Initializing WebGPU inspired by [22] 13
3.1 Initialization of particle data 21
3.2 Compute shader for update of positions in WebGPU 22
3.3 Vertex and fragment shader for update of positions in WebGL 23

List of acronyms and abbreviations

API	Application Program Interface
CPU	Central Processing Unit
DNN	Deep Neural Network
FPS	frames per second
GLSL	OpenGL Shading Language
GPGPU	General-purpose computing on graphics processing units
GPU	Graphics Processing Unit
JS	Javascript
OS	Operating System
RAM	Random Access Memory
WASM	WebAssembly
WGSL	WebGPU Shader Language

Chapter 1

Introduction

With the digital landscape advancing, web applications are becoming more complex and detailed than ever. However, increased complexity and detail require more computational power. Traditionally, heavy applications require downloads to leverage a computer's full processing capacity, but what if we could bring more of that power to the browser?

WebGL and WebGPU are web graphics Application Programming Interfaces (APIs) that tunnel to the computer's powerful Graphics Processing Unit (GPU) directly in the browser, which is typically inaccessible on the web. These technologies can enhance a web application with advanced 3D graphics and parallel processing, improving immersion and performance. Web applications can run on any device and browser, making them more accessible to everyone than platform-specific applications. By providing access to the GPU directly in the browser, these APIs can open up exciting possibilities for high-performant, cross-platform and accessible web applications.

1.1 Background

GPU technology has significantly advanced in recent years, revolutionizing fields such as machine learning, scientific computing, and computer graphics. These areas benefit from GPUs because they can quickly perform complex computations through parallel processing. Traditionally, web graphics were limited by the processing power of the CPU. However, the introduction of WebGL marked a significant shift by providing a gateway to the GPU through a web graphics API. This advancement enabled more sophisticated 3D graphics directly on the web, opening up new web possibilities such as more complex games and other visually engaging web applications. WebGL is today

a web standard[19, 45] and is used by web applications such as Figma[13] and Google Maps[29] which both benefits from GPU acceleration. However, WebGL has its limitations, including a lack of support for GPGPU[20] and an inability to fully leverage the functionalities of modern GPUs. Even if WebGL is leveraging GPU acceleration, it is still relatively slow compared to native technology using plug-in methods[23]. WebGPU is an emerging standard graphics API for the web aiming to provide access to these modern capabilities of GPUs on the web. This can open up exciting new possibilities, such as directly bringing machine learning and heavy computer games to the browser. However, as a completely new graphics API, it has not yet been fully investigated[45]. A particle system is a collection of particles in the form of 2D images with defined behaviours and attributes. It can be very computationally intensive, with an increasing number of particles. Therefore, they benefit from running on the GPU. In computer graphics, they are commonly used to represent, for example, fire and smoke [10]. This study examines the performance differences between WebGL and WebGPU in 2D web-based particle systems to provide insight into WebGPU as a new technology.

1.2 Research question

The thesis aims to evaluate the following research question. *How does performance efficiency vary between WebGPU and WebGL in web-based particle systems across different numbers of particles and sizes, and how are these differences affected by the power of the GPU?*

- How do WebGPU and WebGL compare in terms of total GPU time per frame for web-based particle systems with varying numbers of particles and sizes across GPUs of different powers?
- How do render and compute times compare within the total GPU time per frame for WebGPU and WebGL in web-based particle systems, and how do these times vary with changes in the number of particles and GPU powers?
- What are the limits on the number of particles and square size for maintaining 60 fps in web-based particle systems using WebGPU and WebGL across different GPUs?
- How does the number of particles and sizes of particles affect the CPU initialization time?

1.2.1 Performance metrics

The research question involves comparing two web-based particle systems in WebGL and WebGPU respectively and testing on varying numbers of particles and sizes of the particles and looking at the following performance metrics:

- Total GPU time (ms) per frame.
- Compute time (ms) per frame.
- Render time (ms) per frame.
- Frames per second (fps)
- Initialization time of the applications (ms)

With the help of these metrics, the performance of the two applications can be quantified, and help identify the limits, strengths, and weaknesses of both APIs.

1.3 Purpose

The purpose of this project is to contribute valuable data and findings to the field of web graphics, focusing on the emerging WebGPU technology. WebGPU has more limited research available compared to the already established WebGL, and given WebGPU's potential to become a new standard, it is essential to offer data and statistics that show what benefits and drawbacks will come from switching to this new technology. The findings of this project are supposed to show concrete performance comparisons between WebGPU and WebGL, which can guide developers, researchers, and organizations in making informed decisions on which API to use for specific web development needs. Bontouch, the company where this thesis is being conducted, is interested in learning more about cutting-edge technology. The findings can also contribute inspiration for future studies within the area.

1.3.1 Sustainability and social considerations

Since the thesis focuses on performance efficiency and resource utilization, the findings can impact the choice of a specific web graphics API with sustainability in mind. Efficient use of hardware resources can reduce energy consumption and extend the lifespan of devices. If switching to a new technology can achieve these goals, its adoption becomes a more critical consideration. Cross-platform technology is important from a social

perspective since it's something everyone can use, regardless of device, and it increases accessibility.

1.4 Goals

The goal of this project is to create two web-based particle system applications in WebGL and WebGPU, respectively, that can be comparable for performance testing. The applications will also allow for various inputs, including visualized FPS. With the help of these applications, the goal is to conduct a detailed performance analysis under different conditions to understand how they compare in efficiency and resource utilization. An end goal is to use the results of the tests to provide statistics of the data collected from the tests.

The project's deliverables and results will include an open-source GitHub page hosting the two WebGL and WebGPU applications. The platform will enable users to experiment with the various inputs and observe the performance variations between them on their computers. A detailed performance benchmarking report about tests performed on the applications will also be provided to offer comprehensive comparison statistics.

1.5 Delimitations

The thesis is limited to benchmark testing on particle systems in a Google Chrome Canary browser in 2D, excluding 3D rendering. Google Chrome Canary was chosen for its established support for WebGPU since browser compatibility of WebGPU is limited across web browsers[46]. The scope of testing will be limited to desktop browsers, excluding mobile browsers, and particle systems within WebGPU and WebGL will only be compared. The particle systems will exclude lifetime, spawning, and complex physics simulations and focus on numbers of particles and sizes with a constant speed where all particles are independent of each other. Additionally, all particles will be equal to each other for comparative testing. The study will exclude qualitative data, such as user evaluations, and strictly analyze quantitative data. Tests will be based on the current state of WebGPU, which may change since the API is still in development.

Additionally, the thesis will not be tested across different operating systems and will focus solely on Windows. Only two different hardware setups will be compared—one with a powerful GPU and another with a weaker GPU. These delimitations focus on the technical core differences between WebGL

and WebGPU to demonstrate findings applicable to the everyday use of these APIs on the web.

1.6 Structure of the thesis

Chapter 2 presents relevant background information about GPU programming concepts and WebGPU and WebGL. Chapter 3 presents the methodology and implementation used for the research. Chapter 4 presents the results and analysis of the tests. Chapter 5 presents a discussion about the results and future research. Finally, Chapter 6 presents a conclusion for the research.

Chapter 2

Background

This chapter provides basic background information about GPU programming, including computer graphics and general-purpose computing on the graphics processing unit. It also describes WebGL and WebGPU in more detail and related work.

2.1 GPU programming

GPU programming refers to writing code that runs on a Graphics Processing Unit and is often used in areas that include tasks where parallelization is effective. GPU programming involves both computer graphics and GPGPU, which this section will cover.

2.1.1 Computer graphics

Computer graphics involve using computers to create and manipulate images [18]. Graphics can be 2D or 3D and are mostly associated with video games. However, computer graphics are found in many other areas, such as animation, virtual reality (VR), augmented reality (AR), visual effects, simulations, CAD/CAM, medical imaging, and information visualization [31]. To produce and manipulate an image, there is a process called the graphics/rendering pipeline that includes a series of steps that graphic data goes through to output a final image on a screen. Most graphics are drawn using triangles, a common type of primitive in graphics rendering. However, other primitives such as lines and points can also be used [11]. The process of drawing an image with triangles (the rasterization pipeline) includes the following:

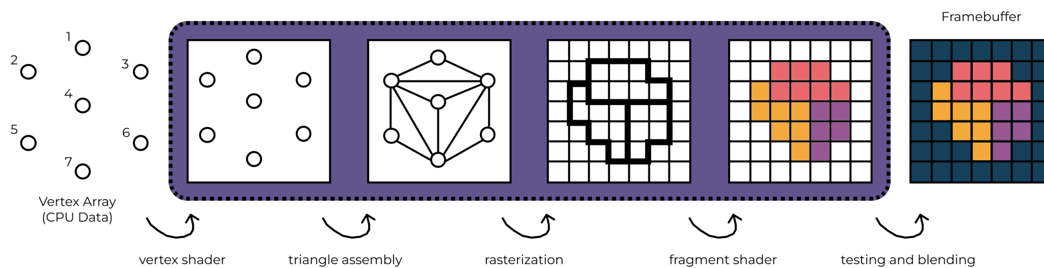


Figure 2.1: The graphics pipeline in OpenGL inspired by [11]

1. Define the coordinates of the geometry to be rendered as an array.
2. Transform vertex data to camera perspective and screen coordinates.
3. Form triangles of the vertices.
4. Rasterization. Determine which pixels will be covered by each triangle in pixel space.
5. Calculate colors and effects for each pixel
6. Determine which pixels to show and blend colors for the final output[11].

Each stage in the graphics pipeline performs specific operations on the data to end up with a final rendered image on the screen. Parts of the stages are programmable, which makes it possible for the developer to manipulate the output by including, for example, lighting, positions, and colors [5].

2.1.2 Computer architecture

CPUs and GPUs are both computational hardware components found in the architecture of a computer. The CPU acts as the brain of any computing device and manages all types of general purpose computing tasks required for the OS and applications to run, such a process input, storing data and output results. It follows instructions serialized, whereas the GPU is designed for parallel instruction processing [14].

2.1.3 Graphics Processing Unit

GPUs were originally designed to accelerate 3D graphics rendering, hence the name [14]. The purpose was to parallel the rendering of pixels on the screen, reducing the computational load on the CPU. The concept of GPUs

dates back to the early 1980s but has developed significantly since [14]. In 1999, the graphics pipeline was fully implemented on the GPU. Later, a pivotal shift was marked, discovering that the GPU parallelization capabilities could extend beyond graphics. The launch of NVIDIA's CUDA in 2006 was a significant milestone for the beginning of GPGPU [6, 26]. CUDA offered a programming model that facilitated general-purpose computations on the GPU and made it possible to do computations in a parallelized manner on an NVIDIA GPU [14]. With this breakthrough, a wide range of applications could leverage accelerated tasks. Today, GPUs are widely used in machine learning and artificial intelligence [9]. They are also a necessary part of most computers [24].

2.1.4 Graphics APIs

A GPU supports various native graphics APIs. An APIs includes a set of functions or tools as a direct way to talk to the GPU on the device and leverage them to create rich graphics applications. The modern graphics APIs are extremely low-level, which enables developers to optimize their applications for better performance and quality. OpenGL is one of the oldest native graphics APIs, first released at the end of 1980s [35], and it provides less direct control over the hardware compared to more recent APIs like Vulkan, Metal, and DirectX 12. Vulkan was developed by Khronos Group and is used primarily on Linux and Android. Metal, developed by Apple, is used on iOS and MacOS. DirectX 12 is for the Windows Platform developed by Microsoft. They all were introduced around 2014 as low-level graphics APIs [1]. Even though OpenGL provides more high-level control, the advantage is that it is widely supported across different platforms and easy to use, as it used to be one of the most popular graphics API [12, 14]. Low-level graphics APIs are essential for developing high-performance graphics applications to leverage the hardware's full capabilities. However, they require much more understanding of memory allocation and programming knowledge than using high-level APIs or libraries that are easier to understand [1].

Graphics APIs for the web, like WebGL and WebGPU, are not native in the same way as they act as a layer on top of a native graphics API to make it better suitable for the web. WebGL generally builds on top of OpenGL [38] whereas WebGPU builds on top of the modern graphics APIs depending on what OS that is in use [7]. The graphics API commands are sent to the driver on the computer, where the calls are translated into low-level commands that the GPU can execute.

2.1.5 Shaders

A shader is a custom program designed to execute on a specific stage of a graphics processor [24]. Shaders give instructions executed simultaneously on the GPU in parallel for programmable phases of the graphics rendering pipeline to modify the output. Still, they can also be used for general-purpose computations outside the rendering pipeline. They are written in specialized shader languages that vary over different graphics APIs [17]. Simply put, shaders are different functions run on the GPU [43]. They enable developers to describe how objects are drawn using vertex transformations, colors, and lights, how they interact with lights and materials in a scene, and how computations are made. Different types of shaders are executed at various parts of the pipeline, and the most essential are vertex shaders, fragment shaders, and compute shaders [17].

A **vertex shader** is executed as the first programmable stage in the rendering pipeline. The vertex shaders process each vertex individually, allowing for manipulation on this vertex, such as positioning, lighting, and texture coordinates [30, 33]. The vertex is a point in 2D or 3D space, and the vertex shader describes this point and its attributes in detail [32].

The **fragment shader** outputs the final color for each pixel (fragment) displayed on the screen. It can be manipulated to handle tasks such as texturing and applying lighting effects before outputting the final color of the pixel being processed. The fragment shader is called once per fragment [32, 42].

A **compute shader** differs from other shaders and is not part of the regular rendering pipeline. A compute shader is designed for GPGPU tasks and does not need to be necessarily related to graphics. Therefore, it is not dependent on any HTML graphical output resources [24]. The purpose of compute shaders is to leverage the parallel processing capabilities of GPUs for computations, essentially running a standard function on the GPU [43].

2.1.6 Particle systems

A particle system is defined as a collection of a large number of small graphic elements with defined behaviours and attributes. Within the system, particles can be spawned, move around, and disappear [36]. They have been heavily used in many areas of computer graphics to represent complex natural phenomena such as smoke, fire, water, blood splatter, snow, clouds, glow, etc [10] where many small discrete particles make up the complex system. Still, particle systems are also used for scientific visualizations [40]. Every particle in a system has properties such as position, velocity, color, size,

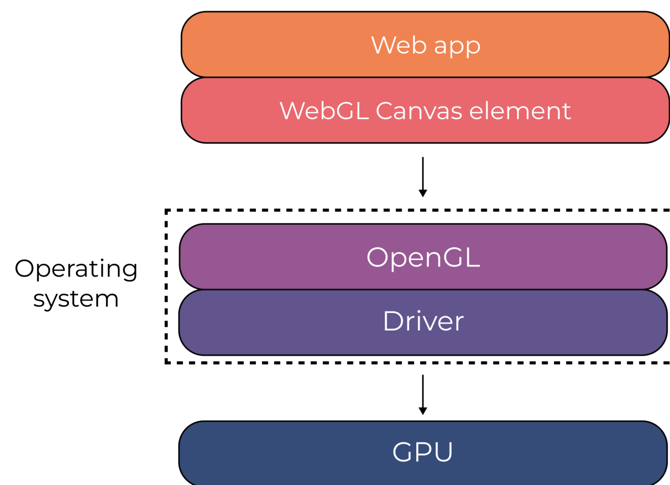


Figure 2.2: Simplified WebGL Architecture inspired by [30, 38]

transparency, and lifespan but can also have other properties that are wished to keep track of [10]. The properties can be changed over time according to predefined rules within the system. In traditional particle engines, the CPU manages physics-based behaviours and movements of particles and sends the data to the GPU to handle the rendering. This setup has evolved with the knowledge and evolution of GPUs, which now feature powerful computational capabilities and efficient memory handling. It is now possible to run a particle system entirely on the GPU [25]. Particle systems often render thousands of tiny particles, which is demanding for the graphics pipeline. The movement of particles involves computations that can be parallelized and can make use of the GPU. Since particle systems can be run entirely on the GPU and are computationally intensive, they are considered a good benchmark on a graphics API to see how efficiently the API utilizes GPU parallelism and how well it can handle a large amount of particles that require constant real-time update. The APIs can be pushed to their limits by stress testing to identify bottlenecks in data transfer, memory bandwidth, and processing power and have been used in other performance evaluations such as [28, 44].

2.2 WebGL and WebGPU

WebGL and WebGPU are graphics APIs for the web, but they differ significantly in how they handle resources, prepare workloads, and submit tasks to the GPU [45].

2.2.1 WebGL

WebGL is a graphics API for the web first released in 2011. Before WebGL, plugins or native applications were needed to get a full graphics 3D experience [35]. WebGL is built as a subset of OpenGL ES (Embedded Systems) 2.0 [12], a native graphics API initially developed in the late 1980s and long considered as a standard in the industry [35]. Like OpenGL, WebGL is operated and maintained by Khronos Group. The API is free, has support in every modern web browser on both desktop and mobile, and no plugin is needed [12, 35]. WebGL accelerates 2D and 3D computer graphics on the web by enabling the power of the GPU hardware on the computer and has for long been the lowest-level graphics API for the web [15]. Numerous libraries have been built on WebGL, including the well-known Three.js for high-level graphics programming [12, 35].

Both OpenGL and WebGL use GLSL as shader language, and shaders are required in WebGL for graphics to show on the screen [32, 35]. Even if OpenGL is written in C/C++, WebGL is written in JavaScript to be designed for the web. Vertex and fragment shaders are used in WebGL and are both needed for a program to execute [32]. The browsers handle the calls made in WebGL by converting them into native graphics commands so the underlying hardware driver can understand them. If the OS supports OpenGL drivers as a native graphics API, the translation is very similar as WebGL is based on it [8]. The ideal way of a WebGL application is to make calls directly to the OpenGL driver, but this is usually not the case depending on the OS [38]. ANGLE is a layer some operating systems use to seamlessly translate the calls to one of the native graphics APIs supported with that platform [16]. Figure 2.2 shows a simplified WebGL architecture without ANGLE.

WebGL was, for a long time, the most low-level graphics API on the web but is much more high-level than WebGPU. This means that WebGL offers less control than WebGPU over the GPU to the developer. The strength of this is that WebGL is easier to use and does not need many configurations to be able to use it. However, less control presents more CPU performance overhead. One example of driver overhead in WebGL is in resource management, where the driver needs to keep track of every resource in the rendering pipeline, which comes with unnecessary performance drawbacks that can be avoided by giving this control to the developers [24]. Additionally, WebGL inherits uncontrollable performance overheads from OpenGL, which, as a rather old graphics API, also has a few common issues.

Access to WebGL is provided by using the <canvas> element in HTML and

```
<canvas id="webglCanvas" width="640" height="480"></canvas>

<script>
const canvas = document.getElementById("webglCanvas");
const gl = canvas.getContext("webgl2");
if (!gl) {
  throw new Error("Can't initialize WebGL");
}
</script>
```

Listing 2.1: Initializing WebGL inspired by [41]

```
<canvas id="webGPUCanvas" width="640" height="480"></canvas>

<script>
const canvas = document.getElementById("webGPUCanvas");
const context = canvas.getContext("webgpu");

if (!navigator.gpu) {
  throw new Error("WebGPU not supported on this browser.");
}
const adapter = await navigator.gpu.requestAdapter();
if (!adapter) {
  throw new Error("No appropriate GPUAdapter found.");
}

const device = await adapter.requestDevice();
const canvasFormat = navigator.gpu.getPreferredCanvasFormat();

context.configure({
  device: device,
  format: canvasFormat,
});
</script>
```

Listing 2.2: Initializing WebGPU inspired by [22]

obtaining a drawing context for it [35]. The initialization process of WebGL is shown in Listing 2.1.

WebGL is designed for graphics processing and has no support for compute shaders, making it less suitable for general-purpose computing on the GPU (GPGPU). GPGPU in WebGL comes with overhead costs [20]. There are two versions of WebGL: WebGL 1.0, released in 2011 and based on OpenGL ES 2.0, and WebGL 2.0, released in 2017 and based on OpenGL ES 3.0. WebGL 2.0 will be used in this research, which, in addition to version 1.0, features transform feedback, allowing the output of vertex shaders to be saved in a buffer [34].

2.2.2 WebGPU

WebGPU is an upcoming modern graphics API for the web designed as

a successor of WebGL rather than as a replacement [37]. It represents the most significant architectural change of GPU solutions for the web so far [24]. Its first specification draft was released in 2021 by W3C's GPU for the Web Community Group, with engineers from Apple, Mozilla, Microsoft, Google, and others agreeing on a potentially new web standard graphics API. WebGPU is open source; anybody can join and help with the development [37, 47]. In April 2023, WebGPU was first shipped by default in the Chrome 113 browser [3]. However, it is still under development and not yet available on all platforms in all browsers, but it strives to be in the near future. In some browsers, it is not available by default but can be used by enabling a flag [46]. The development of WebGPU was driven by the industry demand for a more modern graphics API for the web, more similar to recent native APIs that provide developers with more control over the hardware. WebGPU is designed to meet these needs and give more control of the GPU to the developers [24]. However, this higher control makes the API more complex and challenging to learn than with WebGL.

Except for more control, WebGPU also offers tools for general computing on the GPU, including a compute shader. With a compute shader, general-purpose computing can be done on the GPU with little overhead [20, 37]. Better GPU acceleration can be used to optimize many areas on the web, such as video, digital image, audio signal processing, statistical physics, scientific computing, medical imaging, computer vision, neural networks, deep learning, cryptography, among many others [24].

WebGPU is just like WebGL, written in JavaScript, and uses an HTML <canvas> object to draw. Additionally, the API introduces a novel shading language called WGSL. However, WebGPU is a bit more complex to initialize than WebGL, as the increase of control over the hardware requires more configuration and management by the developer [24]. WebGPU divides functionalities into distinct contexts, unlike WebGL, which relies on a single context object. This architecture makes WebGPU more parallel and simplifies state management compared to WebGL [45]. When initializing WebGPU, an adapter is requested with $navigator.gpu.requestAdapter()$, allowing for a specific GPU to be selected. This is particularly beneficial for modern machines equipped with multiple GPUs [7, 45]. The initialization code for WebGPU is seen in Listing 2.2.

WebGPU is not a subset of another API as with WebGL. However, it acts as an abstraction layer over modern native graphics APIs such as Vulkan, Metal, or DirectX 12. Depending on the OS that is in use, the commands will be translated to fit the underlying system [7]. An overview of the WebGPU

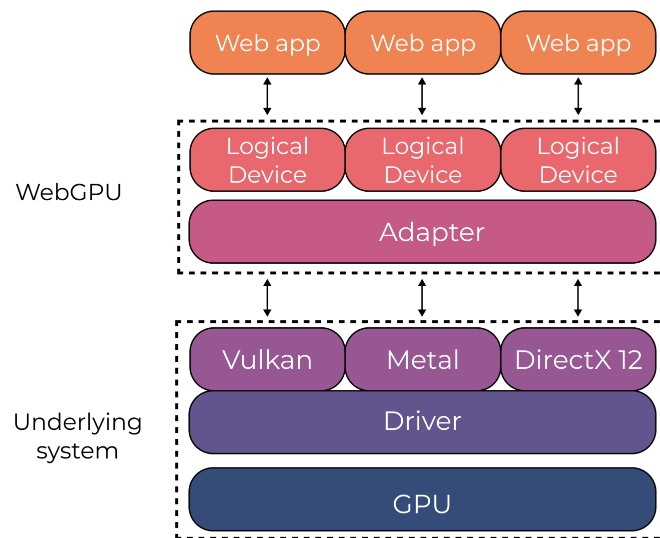


Figure 2.3: The WebGPU Architecture inspired by [7]

Architecture is shown in 2.3.

2.2.3 Shader languages - GLSL and WGSL

2.2.3.1 GLSL

WebGL utilizes GLSL ES as shader language, the same language that OpenGL uses and was developed from the OpenGL Shading Language (GLSL) and is written in a C language syntax [32, 35]. WebGL has vertex and fragment shaders but has no support for compute shaders, making it more complicated to do GPGPU using this API and introducing more overheads. WebGL is designed to create graphics and understand problems through graphic primitives. With a good understanding of this, it is still possible to create GPGPU solutions using vertex and fragment shaders. The principle is storing the data in textures, conducting the calculations, and then reading the result back [24] or using transform feedback that was introduced in WebGL 2.0 [34]. A texture is not an image; it is a 2D array of colors that are values. Similarly, a buffer not only contains positions but can also contain any data. Transform feedback means allowing the output of a vertex shader to be captured in GPU memory and reused as input. GPGPU can be creatively done in WebGL without using compute shaders. This includes using one pair of vertex and fragment shaders for updating positions and one pair for rendering the particles and separating them with different programs [42].

2.2.3.2 WGSL

WGSL is a completely new shading language developed parallel to WebGPU [24]. It shares similarities with other shading languages but is designed to be specifically tailored for WebGPU. WGSL introduces a compute shader for WebGPU for general-purpose computing that includes functionalities such as atomic operations and other tools to easier compute on the GPU [2, 24]. WebGPU consists of a draw command to execute a render pipeline and a dispatch command to execute a compute pipeline for the compute shader [2].

2.3 Related work

2.3.1 WebGPU versus WebGL

There hasn't been much research made yet regarding rendering comparisons between WebGL and WebGPU. A Master Thesis made a performance comparison of WebGL and WebGPU in the Godot game engine as a backend[15]. The study's findings suggest that WebGPU generally outperforms WebGL in synthetic tests and games, with a faster mean GPU and CPU time.

Brandon Jones is one of the key spec editors of WebGPU and has also contributed significantly to the development of WebGL 1.0 and 2.0. He is also listed as one of the five official editors on the WebGPU Wikipedia page[48]. His Github page has several examples of WebGPU being used. One of the projects includes a comparison of WebGPU and WebGL for implementing clustered shading on a Sponza model, a popular computer graphics model. Users can explore various parameters and switch between WebGPU and WebGL to observe the differences visually and in FPS. In this implementation, WebGPU is performing significantly better. The web application developed in this thesis will be based on this performance comparison model [21].

Usta has made a performance comparison research between WebGL and WebGPU in real-world use cases for WebGIS applications. The research published in March 2024 showed that WebGPU performed significantly better.[45]

However, there has been some research about comparisons of WebGL and WebGPU when it comes to GPGPU. With the addition of compute shaders in WebGPU, using the GPU for parallel computing on the web is more accessible. According to Hidaka and colleagues, a WebGPU implementation of a DNN is

approximately 36 times faster than the exact implementation with WebGL[20].

2.3.2 Related work within particle systems

A JavaScript/TypeScript developer did a blog post about performance testing in WebGPU compared to WASM and JS using particle systems. The results show that WebGPU could handle around 10 times more particles over JS and WASM. WebGPU was shown to render 23k particles while remaining at 60 FPS. Even at 30k particles, the FPS are still more than 30[28].

A bachelor's thesis studied particle systems in WebGPU and investigated the performance of different ways of setting up particle systems using instancing and vertex pulling. The results show that using a compute shader makes it possible to simulate and render ten million particles around 63 FPS on a GTX 1060. The results also indicate that using vertex pulling leads to better performance than using instancing. [4]

Research from 2004 [27] demonstrates how particle systems benefit from a full GPU implementation. The study is done before the implementation of compute shaders in graphics APIs. Instead, it uses the fragment shader to fully utilize the GPU to simulate and render the particle systems. This implementation allows for a rendering of up to one million particles in real-time compared to earlier implementations that could handle 10 0000, where they did not follow the approach of using fragment shaders for the simulation. Their implementation also handles collision detection and reaction of particles with objects of arbitrary shape. The research demonstrates the benefits of GPU for higher performance in rendering large particle systems. WebGL does not have compute shaders, and therefore, the implementation should still be modified to do the update on the GPU.

A performance evaluation of WebGL and WebVR was conducted in 2019, and their performance was tested in web applications. The study aims to determine which graphics API optimizes hardware resources more effectively when working with Virtual Reality (VR) applications. The performance was tested on FPS and their consumption of system resources like CPU, GPU, RAM, and battery. The findings indicate they perform similarly on PCs, but WebXR outperforms WebGL on mobile devices. They used particle systems as one of the key elements for their performance evaluation as they allowed stress testing of the hardware. They simulate scenarios that require high computational resources and can test the limits of the two APIs[44].

2.4 Summary

GPU programming is the term of programming calls for the GPU on the device to execute in parallel. This is used for rendering graphics and general-purpose computing to accelerate computations. The original purpose of GPUs was to enhance 3D graphics but are today a powerful tool in machine learning and artificial intelligence (AI), due to its parallelization capabilities. Graphics APIs are used to talk to the GPU to render graphics, and modern APIs provide more control to the developer over the hardware. Shaders are used in graphics APIs as the programs that run on the GPU. WebGL is a graphics API for the web built as a subset of an older API, which comes with less low-level and no support for general-purpose computing. WebGPU is a new graphics API for the web and aims to be more designed like modern graphics APIs and introduces a compute shader for general-purpose computing. A particle system is a technique in computer graphics that renders thousands of particles in a system to represent a fuzzy phenomenon like fire or smoke. It can be very computationally intensive to render, making it a good technique for performance comparisons between graphics APIs.

Chapter 3

Method

The purpose of this chapter is to provide an overview of the research method used in this thesis. Section 3.1 describes the research process. Section 3.2 describes the overall design of the applications. Section 3.3 focuses on the implementation techniques of the applications for this research. Section 3.4 describes the benchmarking methods. Section 3.5 explains the hardware being used for the tests. Section 3.6 describes the method used for the collection of performance data. Finally, Section 3.7 describes how the collected data will be analyzed.

3.1 Research process

The overall research process is

1. Develop a particle system web application in WebGL
2. Develop a particle system web application in WebGPU
3. Implement benchmarks to be able to time the GPU and initialization time
4. Conduct tests on different sizes, number of particles, and hardware
5. Gather quantitative data from tests
6. Analyze results

3.2 Experiment design

To gather performance data on WebGL and WebGPU, web-based particle system applications dedicated to benchmarking were developed in each

technology. In the applications, parameters, including the number of particles, speed, and size of particles, can be modified, and those remain the same for all particles for control. Each particle in the system is always initialized with randomized data for position and velocity, with the magnitude determined by the speed parameter. The movement of the particles has been created in a simple way, where each particle moves in one direction and then bounces when reaching an edge, completely independent of other particles. This was the chosen approach because it is consistent and predictable, which makes it efficient for testing. Furthermore, the benchmark can then focus more on measuring the performance impacts of the update and rendering process isolated from complex logic regarding particle behaviour. The parameter lifetime usually included in a particle system has been excluded in the applications to ensure precise control over the number of particles for the tests.

Three particle system variants have been developed for WebGPU and WebGL to create diverse testing scenarios for a more comprehensive evaluation. The first system consists of particles that are simple points, each uniformly colored with a fixed size of one pixel. The second system includes particles that are color-filled re-sizeable squares. The final particle system utilizes identical square particles but textured. This approach is significant to test as it mirrors a typical particle system usage where a texture is usually applied to the particles to achieve the illusion of realistic visual effects such as smoke or fire.

3.3 Implementation

3.3.1 Initialization of particle data

In both technologies and each particle system, the initialization of positions and velocities are implemented using JavaScript-typed arrays of 32-bit floats to store the values. This format is the smallest native data array for floating-point numbers available in JavaScript. Even though libraries in WebGL can enable smaller-bit float arrays, such support is not yet available in WebGPU. Therefore, 32-bit float arrays are consistently used throughout all applications and in both APIs. Using 16-bit float arrays would have been sufficient for storing positions and velocities, potentially saving memory and improving performance.

The positions and velocities are stored in separate data arrays of size two times the number of particles. This is because they can be allocated in separated buffers on the GPU. This separation creates more convenient storage

```
const rand = (min, max) => min + Math.random() * (max - min);
const positionData = new Float32Array(this.particleCount * 2);
const velocityData = new Float32Array(this.particleCount * 2);

for (let i = 0; i < this.particleCount; i++) {
    positionData[2 * i + 0] = rand(-1, 1); // posx
    positionData[2 * i + 1] = rand(-1, 1); // posy

    const theta = Math.random() * Math.PI * 2;
    velocityData[2 * i + 0] = Math.cos(theta) * this.particleSpeed; // velx
    velocityData[2 * i + 1] = Math.sin(theta) * this.particleSpeed; // vely
}
```

Listing 3.1: Initialization of particle data

since the rendering part of the applications only uses the positions while only the update part uses both. The positions and velocities are randomized using the built-in JavaScript function `Math.random()`. This randomization for initial data values does not affect the outcome of performance testing because the performance metrics are influenced primarily by the quantity of data and complexity of computations for every frame rather than by the specific initial values of the data. The initialization of particle data is shown in Listing 3.1.

3.3.2 Storing the data on the GPU

The data was initialized on the CPU. To store this data on the GPU, the approach varies depending on the API. In WebGPU, when requesting a device from a computer - a step necessary to interact with the GPU on that device - there are some `GPUSupportedLimits` that come with that device. Each computer may have different limits, which is essential to consider when developing an application in WebGPU. One crucial limit encountered in this project was `maxStorageBufferBindingSize`, which was reached when attempting to bind large buffers created from a mass number of particles to the compute bind group. To get around this problem when working with a large number of particles, the particles were created in batches to ensure the buffers did not become too big. For every batch, two buffers for positions and two for velocities were designed as vertex and storage buffers. They were then bound to compute bind groups to enable ping-pong buffering, which is a method for switching between the buffers at every frame to not read and write from the same buffer.

In WebGL, these limits were not encountered, and the particle data did not have to be separated into batches. However, two buffers were created for each containing the same data. The reason for this approach is to enable ping-pong

```
@binding(0) @group(0) var<storage, read> positionsA : array<vec2<f32>>;
@binding(1) @group(0) var<storage, read> velocitiesA : array<vec2<f32>>;

@binding(2) @group(0) var<storage, read_write> positionsB : array<vec2<f32>>;
@binding(3) @group(0) var<storage, read_write> velocitiesB : array<vec2<f32>>;

@compute @workgroup_size(256)
fn main(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>)
{
  var index : u32 = GlobalInvocationID.x;
  var p = positionsA[index];
  var v = velocitiesA[index];

  if (p.x <= -1.0 || p.x >= 1.0) {
    v.x = -v.x;
  }
  if (p.y <= -1.0 || p.y >= 1.0) {
    v.y = -v.y;
  }

  positionsB[index] = p + v;
  velocitiesB[index] = v;
}
```

Listing 3.2: Compute shader for update of positions in WebGPU

buffering with transform feedback. This technique uses one buffer to read from and another to write to, alternating every frame to which the buffer is read and which is written. This technique is a good practice in both APIs to ensure no updating on the buffers is done simultaneously as reading for rendering to avoid race conditions.

3.3.3 Update positions

For comparable results, the positions are updated on the GPU in both WebGPU and WebGL. However, the implementation varies significantly.

WebGPU uses compute shaders, ideal for updating buffers directly on the GPU. A compute pipeline was configured with the compute shader shown in Listing 3.2. Compute bind groups were also created, each containing batches of particle data buffers. This setup allowed the particle data to be updated in every frame before drawing them to the canvas with the render pipeline.

In WebGL, compute shaders do not exist. Instead, transform feedback has been used, an efficient way to capture the output of vertex shaders into buffers. A WebGL program was created for this with a vertex shader that updates the position and an empty fragment shader. Additionally, `transformFeedbackVaryings` were added to the program to specify what variables would store the captured data. Then, the transform feedback was set up and bound to the buffers containing the data. Ping-pong buffers

```
#version 300 es
in vec2 positionA;
in vec2 velocityA;

out vec2 positionB;
out vec2 velocityB;

void main() {
  vec2 p = positionA;
  vec2 v = velocityA;

  if (p.x >= 1.0 || p.x <= -1.0) {
    v.x = -v.x;
  }

  if (p.y >= 1.0 || p.y <= -1.0) {
    v.y = -v.y;
  }

  positionB = p + v;
  velocityB = v;
}
```

```
#version 300 es
precision highp float;
void main() {
}
```

Listing 3.3: Vertex and fragment shader for update of positions in WebGL

were set up so the transform feedback would alternate between each frame. The vertex and the fragment shader are shown in Listing 3.3

3.3.4 Rendering points and squares

Rendering of the particles was done using a vertex and a fragment shader and is executed entirely on the GPU. Both WebGL and WebGPU have the functionality to render points. In WebGPU, the topology is set to ‘point-list‘ in the render pipeline, and the number of points is specified in the draw command. In WebGL, points and the number of points are specified directly in the draw command.

A significant difference between the two APIs is that WebGL includes the `gl_PointSize` function, which allows the size of each point to be specified in pixels within the vertex shader. The size is defined such that each point is rendered as a square, with an area equal to $\texttt{gl_PointSize}^2$. WebGPU is limited to rendering points at a fixed size of one pixel. This restriction aligns with the design principles of modern graphics APIs, to have more direct control over the graphics pipeline and less reliance on fixed-function states. However, rendering a square in WebGPU is a slightly more complex process than WebGL. A square is constructed by drawing two triangles requiring six vertices, with two vertices overlapping. To optimize this process in WebGPU applications, a vertex buffer and an index buffer were initialized to store the

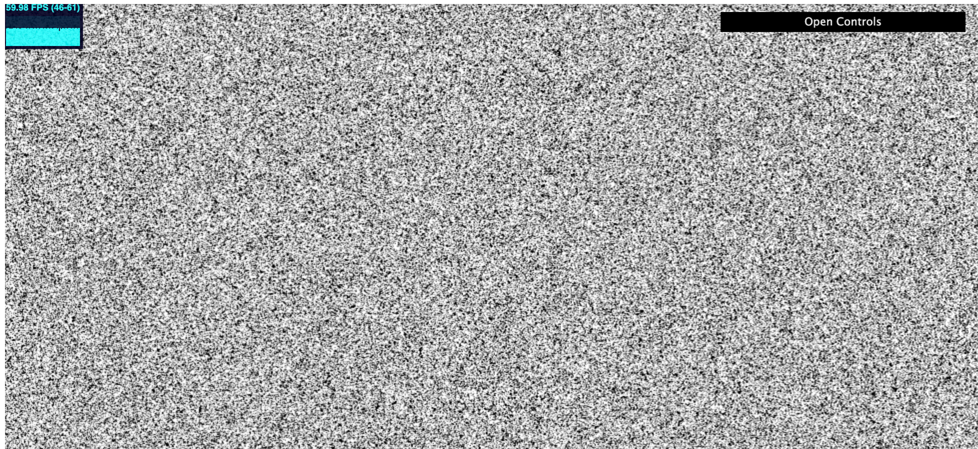


Figure 3.1: Point particle system application

Figure 3.2: Square particle system application

square's data, allowing for vertex reuse during rendering. The topology of the render pipeline was set to 'triangle-list.' These buffers were activated in each frame, and the rendering was executed using a drawIndexed command. Additionally, a resolution parameter stored in a uniform buffer was updated and recalculated in the vertex shader whenever the canvas size changed. This adjustment ensures that the shapes maintain their square proportion and do not become distorted into rectangles. In both WebGPU and WebGL, the squares were filled with just a plain fixed color. This is to focus only on the performance of rendering squares as particles.

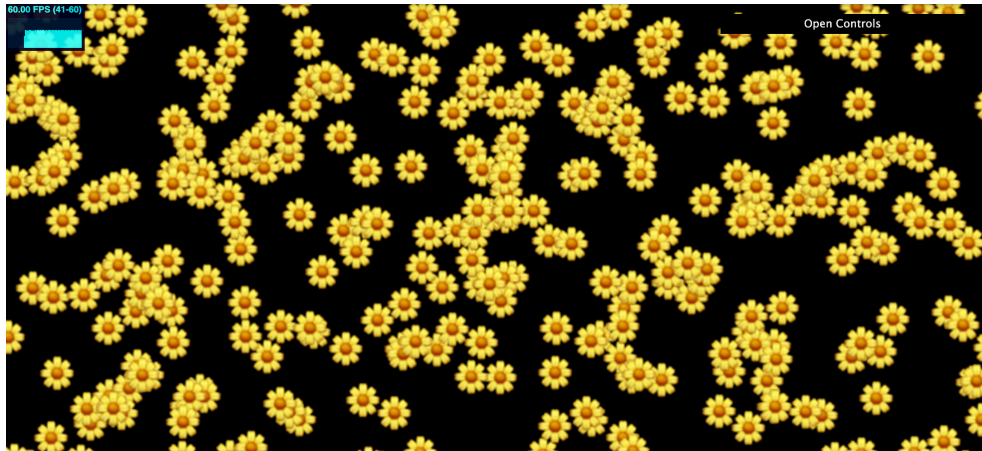


Figure 3.3: Textured squares particle system application

3.3.5 Adding texture to the squares

For both APIs, a texture was created using an OffscreenCanvas filled with an emoji, which was then used as a texture. This method provides a simple way to generate unique textures without uploading images from external files. The reason for this is to not look at the type of texture and instead focus on the configurations of the texture, such as its resolution, format, usage, and blend modes. In each API, the texture was configured in the same way to provide consistency between them for accurate comparisons, and the same emoji texture is used for all applications.

3.4 Benchmarking

3.4.1 Initialization time

In each application, the initialization time was implemented to capture the CPU time required for setup before the rendering process began. This metric was captured using simple CPU timers, specifically through the `performance.now()` function, which returns a high-resolution timestamp in milliseconds. This duration is logged in the console for copying. It requires a web page refresh to be recaptured.

3.4.2 Timing the GPU each frame

To time the GPU for each frame was a bit more complex. Extensions and features needed to be added.

3.4.2.1 Timing the GPU in WebGPU

In WebGPU, enabling timer features for the GPU requires turning on the "WebGPU Developer Features" flag at $chrome : //flags/$ in the Google Chrome Canary browser. On some machines, this feature exists in regular Google Chrome but not consistently between all hardware. Therefore, Google Chrome Canary was used for all tests. With this flag enabled, "$timestamp - query$" can be requested from the device, and timestamps in nanoseconds can be achieved and stored in a buffer on the GPU. The process involves the following steps: Create a query set sized to hold the desired amount of timestamps. Record timestamps during GPU operations using a command encoder. After the operations, resolve the query to a buffer. Finally, read the results back to the CPU. Four timestamps were added to time every frame before, after, and after rendering to time both the rendering and compute part every frame. The four timestamps are to make sure that the timestamps are in pairs, even if there would have been enough to have three timestamps.

3.4.2.2 Timing the GPU in WebGL

In WebGL, timer queries could also be used with the extension `EXT_disjoint_timer_query_webgl2`. This extension does not require turning on any flags in the browser. However, it is not supported in all browsers and does not work on all hardware. With the help of this extension, GPU times in nanoseconds can be achieved. The process differs slightly from timer queries in WebGPU and involves the following steps: First, enable the extension in the WebGL context. Next, create a query object on the context. Begin the query before the GPU operation you wish to measure, and end the query after the operation. The results will not be immediately available. After the operation, you need to check for their availability. Once available, the timing data can be retrieved from the query object, providing the duration of the GPU operation measured in nanoseconds. Two queries must be created to measure both compute time and render time for each frame, and their availability is checked for each frame. WebGL cannot handle too many queries simultaneously, and with operations that take long, the time to retrieve query results may increase significantly. Therefore, the GPU time cannot be retrieved every frame when

timing GPU operations take more time. The timer queries implementation was inspired by [39].

3.4.3 Automated tests

Fully automated tests have not been implemented. Instead, a button was added to initiate a benchmark with a duration of choice. This benchmark records the timestamps for every frame from the timer queries and writes them into a CSV file. At the end of the benchmark, the file is automatically downloaded. This idea was implemented with inspiration from [4]. However, for the WebGL tests, not all frames are captured since the query results can take more than one frame to be available. Instead, longer durations are necessary for the WebGL tests to gather enough data.

3.5 Hardware

3.5.1 Issues with different hardware

It wasn't easy to find hardware where WebGL and WebGPU and their timer queries work. A lot of problems were discovered throughout the process of finding hardware to test on. Two Linux computers were tested, but none were compatible with WebGPU. The applications were developed on a Macbook Pro 2019, where everything worked except for the timer extension for WebGL, showing inaccurate results. Discussions online found that Apple, in general, could show this inaccuracy. However, the applications have not been tested on other Apple devices. Additionally, browsers work differently on different hardware. Google Chrome Canary turned out to work best on all hardware and included all features that are needed for the tests.

3.5.2 Chosen hardware specifications

Table 3.1: Hardware specification strong GPU

Computer	Desktop Gaming Computer
OS	Windows 10
CPU	Intel(R) Core(TM) i7-10700 CPU @ 3.80HHz
GPU	Discrete NVIDIA GeForce RTX 3080

Table 3.2: Hardware specification weak GPU

Computer	ASUS ZenBook 14 UX433F
OS	Windows 10
CPU	Intel(R) Core(TM) i7-8565U @ @ 1.80HHz
GPU	Integrated Intel(R) UHD Graphics 620

3.5.3 Motivation for chosen hardware

Hardware options were limited since the tests were incompatible with Apple or Linux operation systems. The tests were initially intended to be conducted on a standard hardware configuration to represent an average computer. However, during the evaluation, significant performance differences between various hardware setups were observed between WebGL and WebGPU. As a result, two distinct hardware configurations were chosen for the tests. One features a high-performance dedicated GPU (considered a high-end GPU), and the other has a less powerful, integrated GPU (considered a low-end GPU). This is to better evaluate the differences between WebGL and WebGPU in performance between stronger and weaker GPUs.

3.6 Data collection

3.6.1 Tests

Four different tests will be conducted for both WebGL and WebGPU: two to test how the number of particles affects performance and two to test how size affects performance.

1. Point particle system with varying numbers of particles
2. Square particle system of fixed size 2x2 pixel particles with varying numbers of particles
3. Square particle system with a fixed amount of particles and varying sizes
4. Textured square particle system with a fixed amount of particles and varying sizes

All tests will be conducted on a 944x944 pixels HTML canvas. The first test aims to evaluate the efficiency of both APIs in managing point rendering across varying particle counts. However, due to WebGPU lacking a feature for adjusting point size, an additional test is necessary to measure the impact

of the number of particles. This other test represents a more real-life scenario where WebGPU renders resizable squares. Here, WebGPU must render two triangles per point. For this test, a fixed size of 2x2 pixels per square has been chosen to minimize the influence of the size variable and focus on the number of particles. For these tests, each setup will start with 100 particles and then increase the number of particles until the applications cannot handle any more.

The third and last test will evaluate the influence of size by choosing a fixed amount of particles. This fixed number of particles will be adjusted based on the hardware, ensuring a target frame rate of 60 FPS to fall around the midpoint of the size range of 1 to 300 pixels. One million was chosen as the fixed number of particles for the hardware with a powerful GPU. For the hardware with a weaker GPU, 100,000 particles were chosen. The first test aims to evaluate simple color-filled squares, while the second test evaluates those squares with added textures instead. The latter represents a more realistic scenario simulating the usage of particle systems in real-world applications.

3.6.2 Testing process

Eight different setups will be tested on each hardware - four for WebGL and four for WebGPU. The goal is to collect at least 30 data points for each configuration, covering various metrics. For collecting every data point, a benchmark duration of at least 10 seconds will be utilized to record the duration of every frame during this time and store the results in a CSV file. Each row in the file will represent one frame, capturing render time, compute time, total time, and FPS. Total time is the sum of compute and render time, and frame rate is calculated as $1/\text{Total Time in Seconds}$. Each data point will generate a separate file with all frame times and later be averaged as a single row to a new file with one data point per row for one setup. The initialization time for every data point will be collected by refreshing the web page five times and then computing the average from these values.

3.7 Data analysis

The data will be analyzed in Python. Python scripts will read, average, and summarize the CSV files containing data points for each test. A new file will also be generated for every test, including columns for size or number of particles, render time, compute time, total time, frame rate, and initialization time. Eight files will be created for each hardware configuration, covering all tests conducted. These files can be visualized through plotting and fitted to

graphs with regression with calculated R^2. For every test and each hardware configuration, WebGL and WebGPU results will be visualized in the same plot and compared across total time, render time, compute time, and initialization time.

Chapter 4

Results and Analysis

In this chapter, the results of the tests will be presented and analyzed.

4.1 Impact of number of particles

A total of eight different tests were conducted focused on assessing the impact of the number of particles. The tests included two distinct types of particles - point particles and 2x2 pixel particles, both uniformly colored - for both WebGL and WebGPU. Additionally, the tests were conducted on two separate hardware configurations with different GPU powers, NVIDIA GeForce RTX 3080 and Intel(R) UHD Graphics 620. For all tests, the WebGL and WebGPU canvas were 920x920 pixels.

4.1.1 Maximum number of particles

In WebGL, the total amount of particles before becoming too slow or crashing the application was around 100 million for point particles and 50 million for 2x2 pixel particles on the NVIDIA GPU. For WebGPU on the same GPU, the number of particles could reach an impressive 250 million for both particle types. The maximum number of particles on the Intel GPU before becoming too slow or crashing was much less. For point particles, the maximum number of particles could reach 15 million for both WebGL and WebGPU. For 2x2 pixel particles, the max number of particles reached 9 million for WebGL and 7 million for WebGPU.

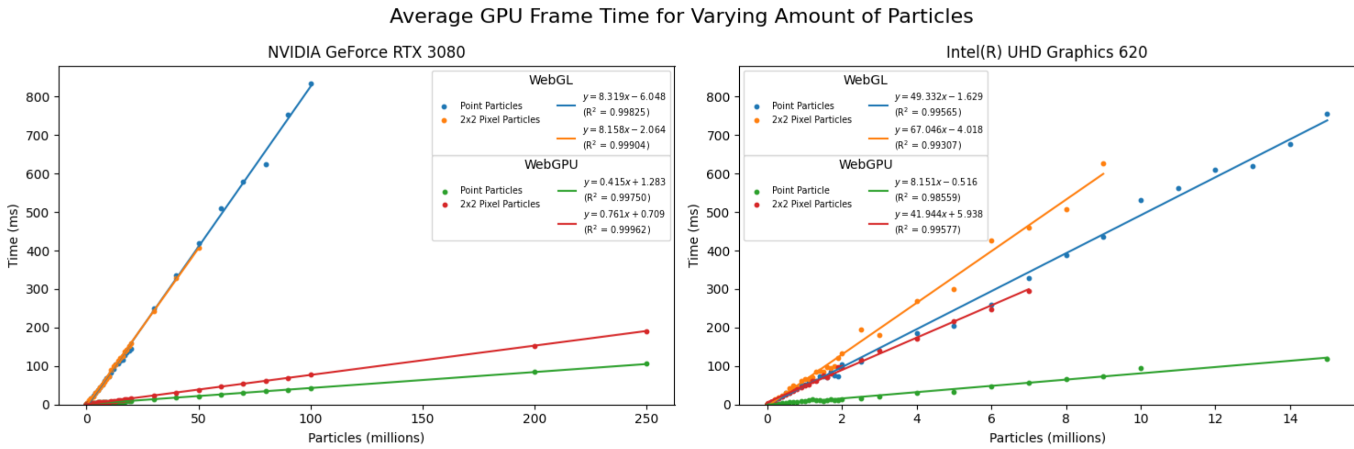


Figure 4.1: Plots for average GPU frame time by number of particles

4.1.2 Total GPU time per frame

Plotting the data points showed a possible linear relationship between the number of particles and time. Linear regressions show a $R2 > 0.997$ for all tests on the NVIDIA GPU and $R2 > 0.985$ for all tests on the Intel GPU, which enhances the fact that the relationship is linear. However, some constants are negative in the models, which indicates that they are not fully accurate since time at zero cannot be negative in this scenario.

Analyzing the fitted models to the data points seen in Figure 4.1, the results are very different between the two different GPUs. The models closely align with the NVIDIA GPU data points, but the fit is less seamless on the Intel GPU. The left plot, which presents results from the NVIDIA tests, reveals a significant, similar increase for both particle types in WebGL, while as for WebGPU, the ascent is not as strong for both particle types. However, the 2x2 pixel particles have a greater ascent than point particles.

In contrast, the right plot, which details the results of the Intel tests, indicates excellent performance for WebGPU point particles. The performance of 2x2 pixel particles in WebGPU is comparable to that of the WebGL point particles. The WebGL 2x2 pixel particles perform the worst, with the most significant increase.

The calculations of the ratios between the coefficients of the linear regression models quantify the performance differences between WebGL and WebGPU across the different tests. As seen in Table 4.11, on the NVIDIA GPU, WebGPU performs approximately 20 times better than WebGL for point particles and about 11 times better for 2x2 pixel particles. For point particles

Table 4.1: Linear coefficients of GPU time with varying number of particles and ratios between WebGL and WebGPU

GPU	Particle Type	k_{WebGL}	k_{WebGPU}	$\frac{k_{WebGL}}{k_{WebGPU}}$
NVIDIA GeForce RTX 3080	Points	8.31925	0.41529	20.03246
	2x2 Squares	8.15823	0.76115	10.71829
Intel(R) UHD Graphics 620	Points	49.33238	8.15082	6.05245
	2x2 Squares	67.04608	41.94413	1.59846

on the Intel GPU, WebGPU performs around 6 times better than WebGL. However, for 2x2 pixel particles on the Intel GPU, WebGPU's performance is only about 60% better than WebGL, indicating a less pronounced improvement in performance.

4.1.3 Frames per second

The framerate is calculated as 1 / Total GPU Time (s). The max framerate on a webpage for WebGL and WebGPU is 60 fps. Knowing this, we can calculate the max number of particles before dropping below 60 fps with the help of our fitted models in Figure 4.1. The maximum number of particles while maintaining 60 fps for the different tests is shown in table 4.2. The results show a more substantial performance difference between WebGL and WebGPU on the more powerful GPU comparing the maximum number of particles at 60 fps. However, WebGPU still handles more particles at 60 fps in all tests on both hardware. The graphs showing the FPS by number of particles can be found in the Appendix.

Table 4.2: Maximum number of particles at 60 fps for different tests

GPU	Particle Type	API	Particles
NVIDIA GeForce RTX 3080	Points	WebGL	2 795 043
		WebGPU	37 047 598
	2x2 Squares	WebGL	2 311 798
		WebGPU	20 966 624
Intel(R) UHD Graphics 620	Points	WebGL	374 104
		WebGPU	2 108 690
	2x2 Squares	WebGL	309 791
		WebGPU	397 560

4.1.3.1 NVIDIA GeForce RTX 3080

Analysis from Table 4.2 indicates that on the NVIDIA GeForce RTX 3080 GPU, WebGPU significantly outperforms WebGL. From the calculated ratios in Table 4.3, it is observed that WebGPU can handle approximately 13 times as many point particles at 60 fps. For 2x2 pixel particles, WebGPU continues to surpass WebGL in performance, managing about 9 times as many 2x2 pixel particles at 60 fps.

Table 4.3: Ratios between WebGPU and WebGL of the maximum number of particles remaining at 60 fps on Nvidia GeForce RTX 3080

Particle Type	**WebGPU**	**WebGL**	$\frac{WebGPU}{WebGL}$
Points	37 047 598	2 795 043	13.255
2x2 Squares	20 966 624	2 311 798	9.069

4.1.3.2 Intel(R) UHD Graphics 620

Continuing analyzing the table 4.2 and looking at the tests conducted on the Intel GPU, it is noticeable that the differences are not as different between WebGPU and WebGL. Table 4.4 calculates the ratios, and there, we can see that WebGPU can handle roughly 6 times more point particles and 1.283 times more 2x2 pixel particles. This indicates that while WebGPU has a significant superiority in rendering point particles, the performance difference for larger particles is less pronounced, yet WebGPU still has a slight advantage of 28%.

Table 4.4: Ratios between WebGPU and WebGL of the maximum number of particles remaining at 60 fps on Intel(R) UHD Graphics 620

Particle Type	**WebGPU**	**WebGL**	$\frac{WebGPU}{WebGL}$
Points	2 108 690	374 104	5.637
2x2 Squares	397 560	309 791	1.283

4.1.3.3 Difference between high-end and low-end GPU

Examining the results of the tests on the less powerful GPU - Intel(R) UHD Graphics 620, compared to the more powerful NVIDIA GeForce RTX 3080, a significant reduction in performance is observed. As seen in Table 4.5, the stronger GPU handles about 7 times more point particles at 60 fps in WebGL and almost 18 times more point particles in WebGPU at 60 fps compared to the

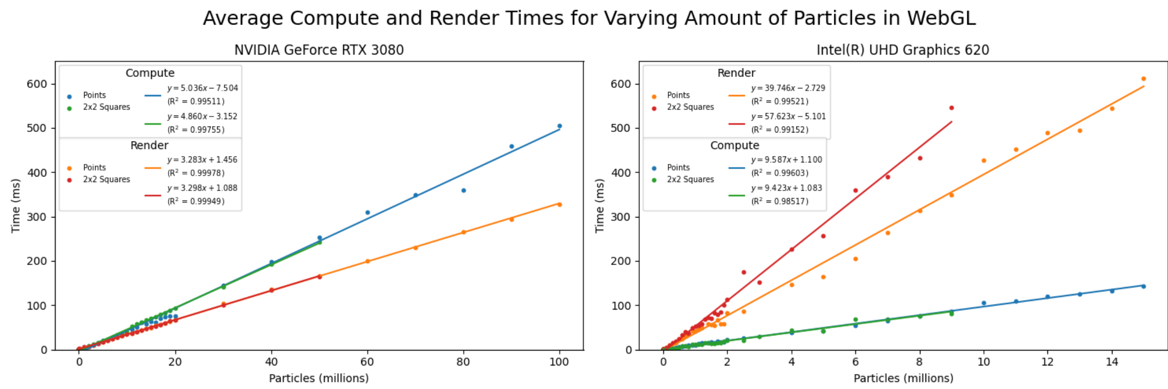


Figure 4.2: Comparison compute and render times in WebGL

weaker GPU. It also supports about 7 times more 2x2 pixel particles in WebGL calculated and an impressive nearly 53 times more 2x2 square particles in WebGPU compared to the weaker GPU.

Table 4.5: Ratios between the results on NVIDIA GeForce RTX 3080 and Intel(R) UHD Graphics 620 for WebGPU and WebGL of the maximum number of particles remaining at 60 fps

Particle Type	**API**	**NVIDIA GPU**	**Intel GPU**	$\frac{NVIDIAGPU}{IntelGPU}$
Points	WebGL	2 795 043	374 104	7.471
	WebGPU	37 047 598	2 108 690	17.569
2x2 Squares	WebGL	2 311 798	309 791	7.462
	WebGPU	20 966 624	397 560	52.738

4.1.4 Breakdown of total time per frame

The total time per frame is the sum of compute and rendering time. These data points can also be fitted to linear regression models.

4.1.4.1 WebGL

Analysis from the plots in Figure 4.2 shows that the compute time in both tests on the NVIDIA GPU is longer than the render time in WebGL. However, both tests' render time is longer on the Intel GPU. On the Intel GPU, it is revealed that render time is the critical factor that results in longer total times for 2x2 particles compared to point particles. Table 4.6 shows the ratio between render time and calculation time for all tests.

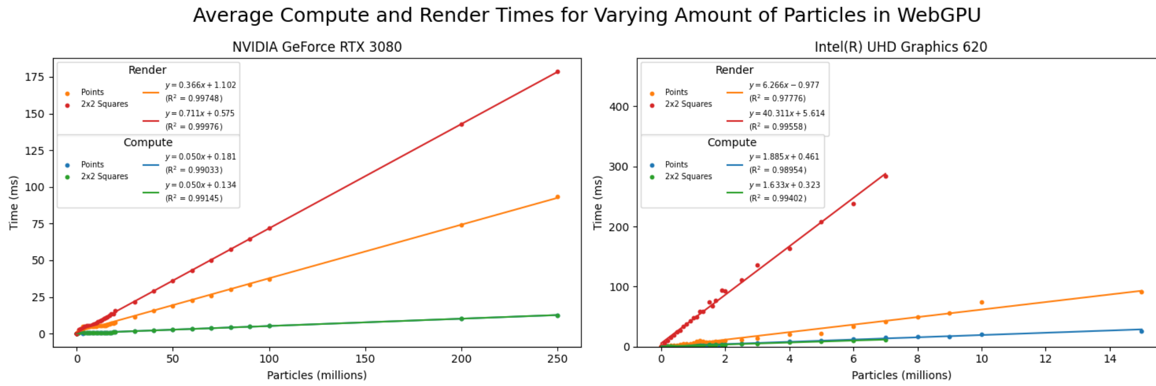


Figure 4.3: Comparison compute and render times in WebGPU

Table 4.6: Compute and render coefficients and ratios in WebGL

GPU	Particle Type	k_{R_WebGL}	k_{C_WebGL}	$\frac{k_{R_WebGL}}{k_{C_WebGL}}$
NVIDIA GeForce RTX 3080	Points	3.283	5.036	0.652
	2x2 Squares	3.298	4.860	0.679
Intel(R) UHD Graphics 620	Points	39.746	9.587	4.146
	2x2 Squares	57.623	9.423	6.115

On the Intel GPU, the render time is about 4 times more than the compute time for point particles and 6 times more for 2x2 pixel particles. On the NVIDIA GPU, the render time was about 35% less than the compute time for point particles and 32% for 2x2 pixel particles.

4.1.4.2 WebGPU

The plots for render and compute times in WebGPU are shown in Figure 4.3 and reveal that compute time overall in WebGPU is significantly low compared to render time for all tests. However, it seems like the render time increases greatly for particles larger than a point.

Table 4.7: Compute and render coefficients and ratios in WebGPU

GPU	Particle Type	k_{R_WebGPU}	k_{C_WebGPU}	$\frac{k_{R_WebGPU}}{k_{C_WebGPU}}$
NVIDIA GeForce RTX 3080	Points	0.36556	0.04973	7.35096
	2x2 Squares	0.71111	0.05004	14.20997
Intel(R) UHD Graphics 620	Points	6.26573	1.88508	3.32385
	2x2 Squares	40.31145	1.63269	24.69023

Table 4.7 shows the coefficient ratio between compute and rendering time in WebGPU. It reveals a significant difference between rendering and computing time for 2x2 pixel particles. On the NVIDIA GPU, the ratio is about 14, and on the Intel GPU, the ratio is almost 25.

4.1.4.3 Comparison between WebGL and WebGPU

Table 4.8 presents the coefficient ratios between WebGL and WebGPU for compute and render time for all tests. The analysis reveals a significant benefit for WebGPU in computing time compared to WebGL on the high-performance NVIDIA GPU. The compute time ratio between WebGL and WebGPU is approximately 100 for both particle types, suggesting that WebGPU outperforms WebGL on a more powerful GPU with a compute shader.

Table 4.8: Coefficient ratios between WebGL and WebGPU for rendering and compute time

GPU	Particle Type	$\frac{k_{R_WebGL}}{k_{R_WebGPU}}$	$\frac{k_{C_WebGL}}{k_{C_WebGPU}}$
NVIDIA GeForce RTX 3080	Points	8.980	101.278
	2x2 Squares	4.638	97.123
Intel(R) UHD Graphics 620	Points	6.343	5.086
	2x2 Squares	1.429	5.772

Even though GPU usage was not explicitly tested, it was observed during the tests. On the Intel GPU, both APIs reached 100% usage, whereas on the NVIDIA GPU, only WebGL reached 100% usage. WebGPU achieved barely 30% GPU usage.

4.2 Impact of size

A total of eight tests were done to assess the impact of size. Two particle types were tested with a fixed amount of particles with varying sizes and on two different types of GPUs, NVIDIA GeForce RTX 3080 and Intel(R) UHD Graphics 620. The two types of particles were uniformly colored squares and textured flower emoji squares. For the more powerful NVIDIA GPU, 1 million particles were chosen as the fixed amount. On the less powerful Intel GPU, 100,000 particles were selected as the fixed amount.

When plotting the data points, seen in Figure 4.4, the data demonstrated a quadratic relationship on the form $ax^2 + bx + c$. This is likely related to the fact that the x-axis represents the length of a side of the squares. As size increases, the area of the squares increases quadratically. However, the relationship of time and area is not linear. For the NVIDIA GPU, the quadratic trend is strongly supported with an $R^2 > 0.998$ across all tests. In contrast, the data from the Intel GPU is imprecise and does not align perfectly with any model, possibly due to timing issues on this GPU. For the test in WebGL involving colored squares, the R^2 is as low as around 0.96. Yet, the quadratic solid relationship on the NVIDIA GPU suggests a similar relationship should exist for the Intel GPU. However, this assumption will be approached with caution.

4.2.1 Analysis of quadratic regression models

From the plots in Figure 4.4, it is evident that colored squares in WebGL take the most time on the GPU, and the rest of the tests align closely with each other on both GPUs. Unexpectedly, textured squares perform better than colored squares in all tests. Table 4.9 presents the quadratic regression models.

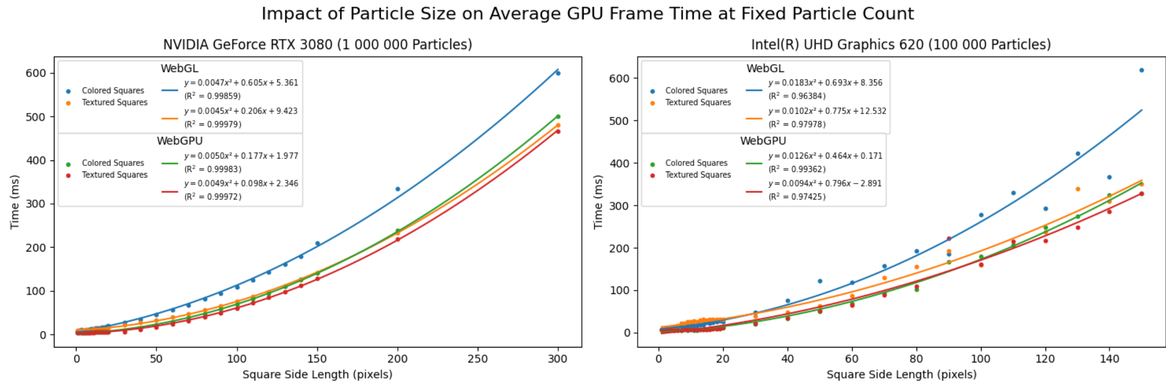


Figure 4.4: Average GPU frame time at fixed particle count by side length of square

Table 4.9: Quadratic regression models for time (ms) with varying side length of squares ($y = ax^2 + bx + c$)

GPU	Particle Type	API	a	b	c
NVIDIA GeForce RTX 3080	Colored Squares	WebGL	0.00467	0.60510	5.361
		WebGPU	0.00496	0.17676	1.977
	Textured Squares	WebGL	0.00455	0.20567	9.423
		WebGPU	0.00485	0.09792	2.346
Intel(R) UHD Graphics 620	Colored Squares	WebGL	0.01832	0.69268	8.356
		WebGPU	0.01258	0.46387	0.171
	Textured Squares	WebGL	0.01023	0.77477	12.532
		WebGPU	0.00940	0.79571	-2.891

4.2.2 Frames per second

The framerate is the inverse of total GPU time in seconds. Even if the quadratic models did not reveal much difference between WebGL and WebGPU, the calculations of maximum sizes of the squares at 60 fps reveal more considerable differences, giving WebGPU a significant advantage in all tests. The results are seen in Table 4.10.

Table 4.10: Max square sizes for a fixed number of particles at 60 fps

GPU	Particle type	API	Max side length $(pixels)$	Max area $(pixels^2)$
NVIDIA GeForce RTX 3080 $(1000000\ particles)$	Colored Squares	WebGL	17	289
		WebGPU	40	1600
	Textured Squares	WebGL	23	529
		WebGPU	45	2025
Intel(R) UHD Graphics 620 $(100000\ particles)$	Colored Squares	WebGL	10	100
		WebGPU	22	484
	Textured Squares	WebGL	5	25
		WebGPU	20	400

4.3 Initialization times

The initialization times are how long it takes for the initial CPU calculations to finish before the rendering begins doing the GPU work. The initialization time did not change across particle types. Therefore, only point particles of varying amounts of particles and a constant number of colored squares with varying sizes are being presented. The plots for initialization time can be found in the Appendix.

4.3.1 Number of particles

The initialization time for varying numbers of point particles exhibits a linear relationship, with linear regression models having $R^2 > 0.99$ on the NVIDIA GPU and $R^2 > 0.988$ for the Intel GPU tests. Considering that each data point is averaged from five tests where the initialization time could be very different, these R^2 are impressively high. The fitted models are shown in Table 4.11. However, for the 2x2 pixel particles on the Intel GPU, the initialization times did not produce accurate results on either API. The initialization time could jump hundreds of milliseconds every refresh.

Table 4.11: Linear regression models for time (ms) with varying amounts of millions of point particles ($y = kx + m$)

GPU	API	k	m	R^2
NVIDIA GeForce RTX 3080	WebGL	72.74	119.26	0.99224
	WebGPU	67.66	85.83	0.99997
Intel(R) UHD Graphics 620	WebGL	139.44	79.42	0.99589
	WebGPU	148.54	175.98	0.98811

The linear regression models indicate that WebGL takes the most time on the NVIDIA GPU to initialize, and WebGPU takes the most time on the Intel GPU, looking at both the k and m values.

4.3.2 Varying size

In the experiments of measuring the initialization time for a fixed amount of squares of varying sizes, both WebGL and WebGPU showed constant initialization times initially on both GPUs for varying sizes.

However, WebGL's initialization time started to increase notably for squares larger than 100x100 on the NVIDIA GPU and 80x80 on the Intel GPU. The GPU was observed to reach 100% usage at these larger sizes using WebGL. Before the increase, WebGL shows a significantly shorter initialization time. These graphs can be found in the Appendix but cannot be fitted to any regression model.

Chapter 5

Discussion

WebGPU consistently outperforms WebGL in terms of total GPU time, render and compute time per frame, and FPS across different numbers of particles and sizes on both hardware. Especially on the more powerful GPU, NVIDIA GeForce RTX 3080, WebGPU performs significantly better than WebGL with around 20 times faster GPU time for point particles and a 10 times faster GPU time for 2x2 pixel particles. This indicates that WebGPU has many advantages when running applications using high-end hardware. At the same time, the difference between WebGL and WebGPU might be less noticeable on lower-end hardware. On the Intel(R) UHD Graphics 620, WebGPU performs with around 6 times faster GPU time for point particles and about 60% faster GPU time for 2x2 pixel particles, which is still an improvement but not as remarkable. WebGPU has excellent potential to improve performance on web applications, including advanced graphics and computational tasks that were previously challenging or impossible in a web environment without plugins or downloads. These improvements will be most significant when using a high-end GPU.

However, something to consider is that these APIs are running in a browser where the developer can not account for the type of GPU being used. The variability of underlying hardware between users of web applications can significantly impact the performance of web applications, as the performance difference can vary greatly for both WebGL and WebGPU across hardware. During the development of a web application, it is therefore essential to account for this variability and design a solution that meets the needs of a broad range of users. It is also necessary to consider the variability of performance consistency across different operating systems and browsers.

5.1 Frames per seconds as measurement

GPU time provides precise performance measurement and clearly illustrates the relationship between the number of particles, sizes, and computational load. However, framerate (FPS) is a more familiar and practical performance metric for end-users. FPS intuitively indicates how smoothly an application runs, making it a more understandable and relatable performance measure. The framerate is the inverse of the GPU time in seconds, and higher framerates are associated with better user experiences. The maximum framerate in a browser for WebGL and WebGPU is usually 60, which developers should strive to meet for a smooth web application.

For the tests conducted in this study, WebGPU demonstrated an impressive capability of handling up to approximately 37 million point particles and 21 million 2x2 pixel particles on the NVIDIA GPU while maintaining 60 fps. In contrast, WebGL managed up to around 2.8 million point particles and 2.3 million 2x2 pixel particles in 60 fps. On the Intel GPU, WebGPU could handle up to around 2.1 million point particles and 398,000 2x2 pixel particles, whereas WebGL supported up to about 374,000 point particles and 310,000 2x2 pixel particles.

For the tests involving different particle sizes, WebGPU demonstrated the ability to handle 1 million square particles, with each particle having an area of nearly 1557 pixels^2 for colored squares and 2041 pixels^2 for textured squares on the NVIDIA GPU, maintaining 60 fps. Under the same conditions, WebGL could manage an area up to nearly 275 pixels^2 for colored squares and 541 pixels^2 for textured squares while maintaining 60 fps. On the Intel GPU, WebGPU could handle 100,000 square particles with an area of up to 493 pixels^2 for colored squares and 396 pixels^2 for textured squares at 60 fps. In comparison, WebGL managed the same amount of square particles with a maximum area of around 92 pixels^2 for colored squares and 25 pixels^2 for textured squares at 60 fps.

These results highlight the significant performance differences between WebGPU and WebGL across different hardware configurations, presented more easily since it can be directly translated to real-world usability. It also shows that the textured square particles in the tests performed better than the colored particles. This was surprising since there are many more setups and configurations for adding texture to the squares. This could be because the texture came from an emoji, including many transparent areas that are ignored in the fragment shader. Therefore, the actual area being filled in the square is smaller than the square. It might have been more comparable to

use a texture without transparent areas for a better comparison of colored and textured squares.

5.2 Benefits of compute shader

The introduction of compute shaders is one of the main advantages that WebGPU has over WebGL and is known for reducing overhead. A compute shader simplifies general-purpose GPU operations in the browser and was used in the tests to update the positions of the particles in WebGPU. The results of the tests show that compute shaders in WebGPU significantly improve the performance of updating particle systems compared to using transform feedback in WebGL. Compute shaders are indicated to be a significant factor in the superior performance of WebGPU in the tests, including varying numbers of particles, particularly on a high-end GPU like the NVIDIA GeForce RTX 3080. On this GPU, the compute shader in WebGPU improves the compute time by 100 times compared to using transform feedback in WebGL for point particles and 2x2 pixel particles. This dramatic difference highlights the efficiency of WebGPU in handling computationally intensive tasks on a high-end GPU.

On lower-end GPUs, such as Intel(R) UHD Graphics 620, the performance improvement of compute shaders is still notable but less pronounced. For the same tests, the compute time is around 5 to 6 times faster in WebGPU compared to WebGL. While this difference is smaller than the one seen on the high-end GPU, it still represents a significant performance improvement. This efficiency of compute shaders contributes to faster computations and frees up resources for the rendering part of the particle systems. The compute part was constant for the tests of varying sizes since no additional computations were made.

5.3 Other benefits with WebGPU

It is no surprise that WebGPU outperforms WebGL, considering it is designed to do so. It is inspired by the newer, modern graphics APIs like Vulkan, Metal, and DirectX 12, which are more low-level APIs, giving the developer more control over the hardware, allowing for reduced overhead and optimization. WebGL is based on the older OpenGL API, which already has limits, along with its shader language, GLSL. Apart from introducing compute shaders in WebGPU, the pipeline is more flexible and is not fixed as it is in WebGL.

WebGPU also supports more advanced GPU features, including asynchronous operations, which are better for parallelization and performance. It is also designed to work well with WASM and can also be developed for use outside the browser. However, it is important to consider that all these modern features require a deeper understanding of GPUs and resource management when developing a web graphics application.

5.4 Initialization time as measurement

The initialization time was quite similar for both WebGL and WebGPU. The initialization time for the point particle tests showed a linear relationship with the number of point particles. The results were incredibly inconsistent on the Intel GPU, particularly for the 2x2 pixel particles. This could be because the initialization time varied significantly with every refresh. This inconsistency makes it challenging to determine which API performs better overall. Both WebGL and WebGPU showed comparable performance, with shifting advantages between the two depending on the hardware and specific test. For the user, a few milliseconds difference is generally not noticeable at initialization. However, something to consider here is as the number of particles gets very large, the initialization time can take several seconds for both WebGL and WebGPU. Even if the GPU can handle a large number of particles at 60 fps, the time to initialize the application should still be acknowledged when developing a heavy application.

The initialization time for a decided number of particles with varying sizes is somewhat constant for both APIs but gives a slight advantage to WebGL for smaller sizes. The fact that the initialization is constant is expected as the square size typically should not influence initialization time, given that it does not add additional computations and setups. However, WebGL's initialization time rose when the particle size got huge. This rise could be linked to the GPU reaching 100% usage at these larger sizes for WebGL, which was observed during the tests. Even if initialization time is mainly CPU load, the initialization phase involves configurations of the GPU, such as setting up buffers, which could increase the load on the GPU. To confirm this hypothesis, further testing would be necessary.

5.5 Application in real-world scenarios

This thesis tested point particles and 2x2 pixel particles with varying numbers of particles. However, using 1-pixel point particles is limited and impractical in real-world applications. Particle systems do not typically contain such small particles. Instead, they are usually larger and more complex to simulate phenomena like smoke, fire, and water. Therefore, the results from 2x2 pixel particles provide a more realistic insight into the performance between WebGL and WebGPU, even if the point particles present remarkable performance results for WebGPU. Two triangles need to be rendered to draw bigger squares than points in WebGPU, which is more computationally intensive than rendering single-point particles. Furthermore, the textured particles in the tests represent a more realistic particle system since they can represent smoke, fire, water, and other complex effects.

5.6 Future for WebGPU

Powerful hardware is becoming more frequent among users, making it possible for WebGPU to thrive. Having WebGPU available for everyone in a browser makes GPU programming more accessible, enabling both GPGPU and computer graphics programming for a broader audience. This accessibility can promote innovation and learning of GPUs for the public and further encourage the development of GPUs.

Industries can benefit from WebGPU by developing advanced web applications that were previously impossible without downloads. Making these applications available directly in the browser can significantly increase their user base. Imagine having sophisticated video and photo editing programs, programming interfaces, computer games, and various other apps running seamlessly in the browser. This not only enhances accessibility for users and across hardware but also simplifies the sharing and collaboration of projects online.

5.7 Method criticism

The study only evaluates two hardware setups, one representing a high-end GPU and one a low-end GPU. However, this is not an entirely accurate representation since the performance may vary significantly across hardware configurations and GPU vendors. Mid-range GPUs are more commonly found

among consumers, which would be a more relevant configuration to test. However, such hardware was not available for these tests. The tests were also conducted on a Windows operating system using the Chrome Canary Browser. This also limits the generalizability of the results. The study does not account for mobile devices.

Furthermore, the tests were not fully automated, which makes room for human errors and makes them harder to replicate. Every test on one data point included refreshing the web page five times to copy and paste the initialization time and start the benchmark with a duration of at least ten seconds to capture GPU times every frame in a CSV file. There are several moments during this process where things can go wrong. Every data point needed to be processed cautiously, and there were around 300 data points in total across all tests, where every point created a separate file that needed to be handled correctly. The tests should be fully automated for more accurate testing to avoid human mistakes and permit more efficient testing across different hardware and platforms. Automated tests could also allow for more data points, longer benchmark durations, and more initialization times to average within one data point. The initialization time in this study was only captured from an average of over five web page refreshes. It varied a lot with every refresh, which did not give consistent results. More data points averaged over many times would lead to more accurate and reliable results.

Transform feedback was used to update the positions of the particles on the GPU using WebGL. This feature is available in WebGL 2.0 but not in WebGL 1.0. This thesis did not explore other methods for updating the positions on the GPU using WebGL. More methods could have been studied for a potentially more accurate comparison with compute shaders in WebGPU.

5.8 Future work

More hardware configurations should be tested and evaluated for future work to provide more generalized performance results between WebGL and WebGPU. Additionally, since this thesis encountered limits with different operating systems and browsers for WebGL and WebGPU, only Windows with a Chrome Canary browser was used for the tests. This might not be the case in the future, and other operating systems and browsers could be analyzed.

Different kinds of particle systems, including 3D particle systems with different textures, shapes, attributes, and more complex behaviour, should be further investigated to better represent a real-world particle system.

Other performance measurements, such as the number of calls to the GPU,

would be interesting to investigate in more depth to quantify overhead and GPU and CPU usage. Additionally, this study focused on using the gl_PointSize function in WebGL to create different sizes of squares. However, future work would be to evaluate squares consisting of triangles in WebGL as were done in WebGPU to investigate how that would influence the performance to have more consistent setups between WebGL and WebGPU. Other ways to update the positions on the GPU in WebGL could also be further investigated along with optimizations in all applications.

Chapter 6

Conclusions

This thesis has explored the performance differences between WebGL and WebGPU in web-based 2D particle systems. The tests revealed significant performance differences between WebGL and WebGPU in terms of total GPU time and render and compute time per frame across various numbers of particles and sizes, as well as on two different GPU powers, one high-end GPU and a low-end GPU.

The findings indicate that WebGPU offers significant performance improvements over WebGL on the high-end GPU, NVIDIA GeForce RTX 3080. On this GPU, with tests over different numbers of particles, the compute time was improved an impressive 100 times over WebGL. The render time was improved around 4 to 9 times, and the total GPU time was improved 10 to 20 times over WebGL.

On the low-end GPU, Intel(R) UHD Graphics 620, the performance differences between WebGL and WebGPU were not as extensive but still significant. The total GPU time in the tests of varying numbers of particles with WebGPU was improved by 60% to 6 times over WebGL, the render time by 40% - 6 times, and the compute time by 5 - 6 times.

The thesis also highlights the efficiency of using compute shaders, a new feature in WebGPU that does not exist in WebGL. Compute shaders were used in the project to update the positions of the particles every frame and improved the compute time significantly compared to transform feedback in WebGL.

The most significant tests were those of varying numbers of particles that showed a linear relationship. Even if the tests of varying sizes resulted in a great advantage to WebGPU, they were influenced by the number of particles. They were also fitted to a quadratic model, which complicated quantifying the relationship between WebGPU and WebGL. The results of initialization times

were quite inconsistent, but they did show a linear relationship with the number of particles overall and were quite similar between WebGPU and WebGL.

For future work, expanding the scope to more hardware configurations, different operating systems, and browsers would provide a more comprehensive understanding of WebGPU's capabilities. Furthermore, investigating 3D particle systems, various textures, shapes, and more complex behaviours of the particles could contribute more valuable insights.

In conclusion, WebGPU is a promising next-generation web graphics API that offers multiple modern features and improved performance capabilities. This research's findings can help developers, researchers, and organizations make informed decisions about adopting this new technology.

The GitHub repo of the study can be accessed here:
github.com/sagapalmer/webgpu-webgl-particles

References

[1] Andi, *I want to talk about WebGPU*, May 2023. [Online]. Available: https://cohost.org/mcc/post/1406157-i-want-to-talk-about (visited on 02/10/2024).

[2] A. Baker, M. Oguz Derin, and D. Neto, *WebGPU Shading Language*, May 2024. [Online]. Available: https://www.w3.org/TR/2024/WD-WGSL-20240514/.

[3] F. Beaufort and C. Wallez, *Chrome ships WebGPU*, en, Apr. 2023. [Online]. Available: https://developer.chrome.com/blog/webgpu-release (visited on 05/19/2024).

[4] P. Benedikt, "Particle System in WebGPU," Bachelor Thesis, TU Wien, 2023. [Online]. Available: https://www.cg.tuwien.ac.at/research/publications/2023/PETER-2023-PSW/ (visited on 01/29/2024).

[5] A. Boreskov and E. Shikin, *Computer Graphics*, en. Oct. 2013, ISBN: 978-1-4822-1557-1. [Online]. Available: https://learning.oreilly.com/library/view/computer-graphics/9781482215571/ (visited on 03/18/2024).

[6] R. Brooks, *Understanding NVIDIA CUDA: Know The Basics of GPU Parallel Computing*, en, Dec. 2023. [Online]. Available: https://medium.com/@rowanbrooks.cloudies/understanding-nvidia-cuda-know-the-basics-of-gpu-parallel-computing-9ec59115f2da (visited on 05/14/2024).

[7] M. contributors, *WebGPU API*, en-US, Jul. 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API (visited on 02/14/2024).

[8] P. Cozzi, *WebGL Insights*, en. Aug. 2015, ISBN: 978-1-4987-1607-9. [Online]. Available: https://learning.oreilly.com/library/view/webgl-insights/9781498716079/ (visited on 03/17/2024).

[9] T. S. Crow, "Evolution of the Graphical Processing Unit," 2004. [Online]. Available: https://www.semanticscholar.org/paper/Evolution-of-the-Graphical-Processing-Unit-Crow/8f68c39e3925275e1d5a881619fb37944b0c4e31 (visited on 03/18/2024).

[10] B. Danchilla, "Fractals, Height Maps, and Particle Systems," en, in *Beginning WebGL for HTML5*, B. Danchilla, Ed., Berkeley, CA: Apress, 2012, pp. 139–171, ISBN: 978-1-4302-3997-0. DOI: 10.1007/978-1-4302-3997-0_6. [Online]. Available: https://doi.org/10.1007/978-1-4302-3997-0_6 (visited on 02/14/2024).

[11] I. Dunn and Z. Wood, *Graphics Programming Compendium*, 2017. [Online]. Available: https://graphicscompendium.com/intro/01-graphics-pipeline (visited on 03/19/2024).

[12] A. Evans, M. Romeo, A. Bahrehmand, J. Agenjo, and J. Blat, "3D graphics on the web: A survey," *Computers & Graphics*, vol. 41, pp. 43–61, Jun. 2014, ISSN: 0097-8493. DOI: 10.1016/j.cag.2014.02.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849314000260 (visited on 05/19/2024).

[13] Figma, *What browsers does Figma support?* en-US. [Online]. Available: https://help.figma.com/hc/en-us/articles/360039827194-What-browsers-does-Figma-support (visited on 05/13/2024).

[14] C. Fox, *Computer Architecture*, en. May 2024, ISBN: 978-1-09-818217-5. [Online]. Available: https://learning.oreilly.com/library/view/computer-architecture/9781098182175/ (visited on 05/14/2024).

[15] E. Fransson and J. Hermansson, *Performance comparison of WebGPU and WebGL in the Godot game engine*, Backup Publisher: Blekinge Institute of Technology, Faculty of Computing Pages: 102, Jun. 2023. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1762429&dswid=-2408.

[16] Google, *ANGLE - Almost Native Graphics Layer Engine*. [Online]. Available: angleproject.org (visited on 05/19/2024).

[17] K. Group, *Shader*, 2019. [Online]. Available: https://www.khronos.org/opengl/wiki/Shader (visited on 03/18/2024).

[18] S. Guha, *Computer Graphics Through OpenGL, 2nd Edition*, en. Aug. 2014, ISBN: 978-1-4822-5839-4. [Online]. Available: https://learning.oreilly.com/library/view/computer-graphics-through/9781482258394/ (visited on 05/21/2024).

[19] M. Hamzaturrazak, E. M. A. Jonemaro, and A. Pinandito, "Performance Analysis of 3D Rendering Method on Web-Based Augmented Reality Application Using WebGL and OpenGL Shading Language," in *Proceedings of the 8th International Conference on Sustainable Information Engineering and Technology*, ser. SIET '23, New York, NY, USA: Association for Computing Machinery, Dec. 2023, pp. 637–643, ISBN: 9798400708503. DOI: 10.1145/3626641.3626949. [Online]. Available: https://dl.acm.org/doi/10.1145/3626641.3626949 (visited on 05/13/2024).

[20] M. Hidaka, Y. Kikura, Y. Ushiku, and T. Harada, "WebDNN: Fastest DNN Execution Framework on Web Browser," in *Proceedings of the 25th ACM international conference on Multimedia*, ser. MM '17, New York, NY, USA: Association for Computing Machinery, 2017, pp. 1213–1216, ISBN: 978-1-4503-4906-2. DOI: 10.1145/3123266.3129394. [Online]. Available: https://dl.acm.org/doi/10.1145/3123266.3129394 (visited on 03/13/2024).

[21] B. Jones, *Toji/webgpu-clustered-shading*, original-date: 2020-12-19T04:48:18Z, Jan. 2024. [Online]. Available: https://github.com/toji/webgpu-clustered-shading (visited on 01/29/2024).

[22] B. Jones and F. Beaufort, *Your first WebGPU app*, en, Mar. 2024. [Online]. Available: https://codelabs.developers.google.com/your-first-webgpu-app (visited on 03/15/2024).

[23] S. Kang and J. Lee, "Developing a Tile-Based Rendering Method to Improve Rendering Speed of 3D Geospatial Data with HTML5 and WebGL," *Journal of Sensors*, vol. 2017, pp. 1–11, Oct. 2017. DOI: 10.1155/2017/9781307.

[24] B. Kenwright, "Web Programming Using the WebGPU API," in *ACM SIGGRAPH 2023 Courses*, ser. SIGGRAPH '23, New York, NY, USA: Association for Computing Machinery, Jul. 2023, pp. 1–184, ISBN: 9798400701450. DOI: 10.1145/3587423.3595543. [Online]. Available: https://dl.acm.org/doi/10.1145/3587423.3595543 (visited on 01/27/2024).

[25] P. Kipfer, M. Segal, and R. Westermann, "UberFlow: A GPU-based particle engine," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '04, New York, NY, USA: Association for Computing Machinery, Aug. 2004, pp. 115–122, ISBN: 978-3-905673-15-9. DOI: 10.1145/1058129.1058146. [Online]. Available: https://dl.acm.org/doi/10.1145/1058129.1058146 (visited on 02/28/2024).

[26] B. Klajdi, "Comparative performance analysis of Vulkan and CUDA programming model implementations for GPUs," Ph.D. dissertation, Jun. 2019. [Online]. Available: https://core.ac.uk/reader/323473500 (visited on 05/19/2024).

[27] A. Kolb, L. Latta, and C. Rezk-Salama, "Hardware-based simulation and collision detection for large particle systems," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '04, New York, NY, USA: Association for Computing Machinery, Aug. 2004, pp. 123–131, ISBN: 978-3-905673-15-9. DOI: 10.1145/1058129.1058147. [Online]. Available: https://dl.acm.org/doi/10.1145/1058129.1058147 (visited on 02/28/2024).

[28] Leo, *WebGPU performance—is it what we expect?* en, Nov. 2023. [Online]. Available: https://medium.com/source-true/webgpu-performance-is-it-what-we-expect-b1c96b1705e1 (visited on 02/06/2024).

[29] M. Leung, *WebGL-powered maps features now generally available*, en, Jun. 2022. [Online]. Available: https://mapsplatform.google.com/resources/blog/webgl-powered-maps-features-now-generally-available/ (visited on 05/13/2024).

[30] S. Li, *Introducing WebGL*, Apr. 2019. [Online]. Available: https://developer.ibm.com/tutorials/wa-webgl1/ (visited on 02/26/2024).

[31] S. Marschner and P. Shirley, *Fundamentals of Computer Graphics, 4th Edition*, en. Oct. 2018, ISBN: 978-1-4822-2941-7. [Online]. Available: https://learning.oreilly.com/library/view/fundamentals-of-computer/9781482229417/ (visited on 03/18/2024).

[32] K. Matsuda and R. Lea, *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL*, en. Addison-Wesley, Jul. 2013, ISBN: 978-0-13-336492-7. [Online]. Available: https://books.google.se/books?id=3c-jmWkLNwUC&printsec=copyright&redir_esc=y#v=onepage&q&f=false.

[33] C. Maughan and M. Wloka, "Vertex shader introduction," *NVIDIA Technical Brief*, 2001. [Online]. Available: https://developer.download.nvidia.com/assets/gamedev/docs/NVidiaVertexShadersIntro.pdf (visited on 03/18/2024).

[34] K. Ninomiya, Z. Mo, K. Russell, and Google, *WebGL 2.0 is Here: What You Need To Know*, Khronos Webinar, Apr. 2017.

[35] T. Parisi, *WebGL: Up and Running*, en. O'Reilly Media, Inc., Aug. 2012, ISBN: 978-1-4493-2648-7. [Online]. Available: https://learning.oreilly.com/library/view/webgl-up-and/9781449326487/ (visited on 03/15/2024).

[36] W. T. Reeves, "Particle Systems—a Technique for Modeling a Class of Fuzzy Objects," *ACM Transactions on Graphics*, vol. 2, no. 2, pp. 91–108, Apr. 1983, ISSN: 0730-0301. DOI: 10.1145/357318.357320. [Online]. Available: https://dl.acm.org/doi/10.1145/357318.357320 (visited on 03/18/2024).

[37] K. Russell and C. Wallez, *WebGL and WebGPU Updates*, Jul. 2022. [Online]. Available: https://www.khronos.org/assets/uploads/developers/presentations/WebGL__WebGPU_Updates_Jul_22.pdf.

[38] A. Shaw, *Computer Graphics and Multimedia*, eng, May 2021. [Online]. Available: https://alg.manifoldapp.org/projects/computer-graphics-and-multimedia (visited on 02/26/2024).

[39] T. Sherif, *PicoGL.js*, en, 2017. [Online]. Available: https://github.com/tsherif/picogl.js/blob/master/src/timer.js (visited on 05/03/2024).

[40] W. Y. Su and J. C. Hart, “A programmable particle system framework for shape modeling,” in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH ’05, New York, NY, USA: Association for Computing Machinery, Jul. 2005, 277–es, ISBN: 978-1-4503-7833-8. DOI: 10.1145/1198555.1198658. [Online]. Available: https://dl.acm.org/doi/10.1145/1198555.1198658 (visited on 02/27/2024).

[41] G. Tavares, *WebGL GPGPU*, en. [Online]. Available: https://webglfundamentals.org/webgl/lessons/webgl-gpgpu.html (visited on 02/23/2024).

[42] G. Tavares, *WebGL2 Fundamentals*, en. [Online]. Available: https://webgl2fundamentals.org (visited on 02/26/2024).

[43] G. Tavares, *WebGPU Fundamentals*, en. [Online]. Available: https://webgpufundamentals.org (visited on 05/20/2024).

[44] R. M. Toasa G, P. Baldeón Egas, M. Saltos, M. Perreño, and W. Quevedo, “Performance Evaluation of WebGL and WebVR Apps in VR Environments,” in Oct. 2019, pp. 564–575, ISBN: 978-3-030-33722-3. DOI: 10.1007/978-3-030-33723-0_46.

[45] Z. Usta, “WEBGPU: A NEW GRAPHIC API FOR 3D WEBGIS APPLICATIONS,” English, *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLVIII-4-W9-2024, pp. 377–382, Mar. 2024, Conference Name: GeoAdvances 2024 – 8th International Conference on GeoInformation Advances - 11–12 January 2024, Istanbul, Türkiye Publisher: Copernicus GmbH, ISSN: 1682-1750. DOI: 10.5194/isprs-archives-XLVIII-4-W9-2024-377-2024. [Online]. Available: https://isprs-archives.copernicus.org/articles/XLVIII-4-W9-2024/377/2024/isprs-archives-XLVIII-4-W9-2024-377-2024.html (visited on 03/18/2024).

[46] W3C, *Implementation Status*, en. [Online]. Available: https://github.com/gpuweb/gpuweb/wiki/Implementation-Status (visited on 01/19/2024).

[47] W3C, *W3C GPU for the Web Community Group*. [Online]. Available: https://github.com/gpuweb/gpuweb (visited on 05/19/2024).

[48] *WebGPU*, en, Page Version ID: 1209675396, Feb. 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=WebGPU&oldid=1209675396 (visited on 03/12/2024).

Appendix A
Additional plots

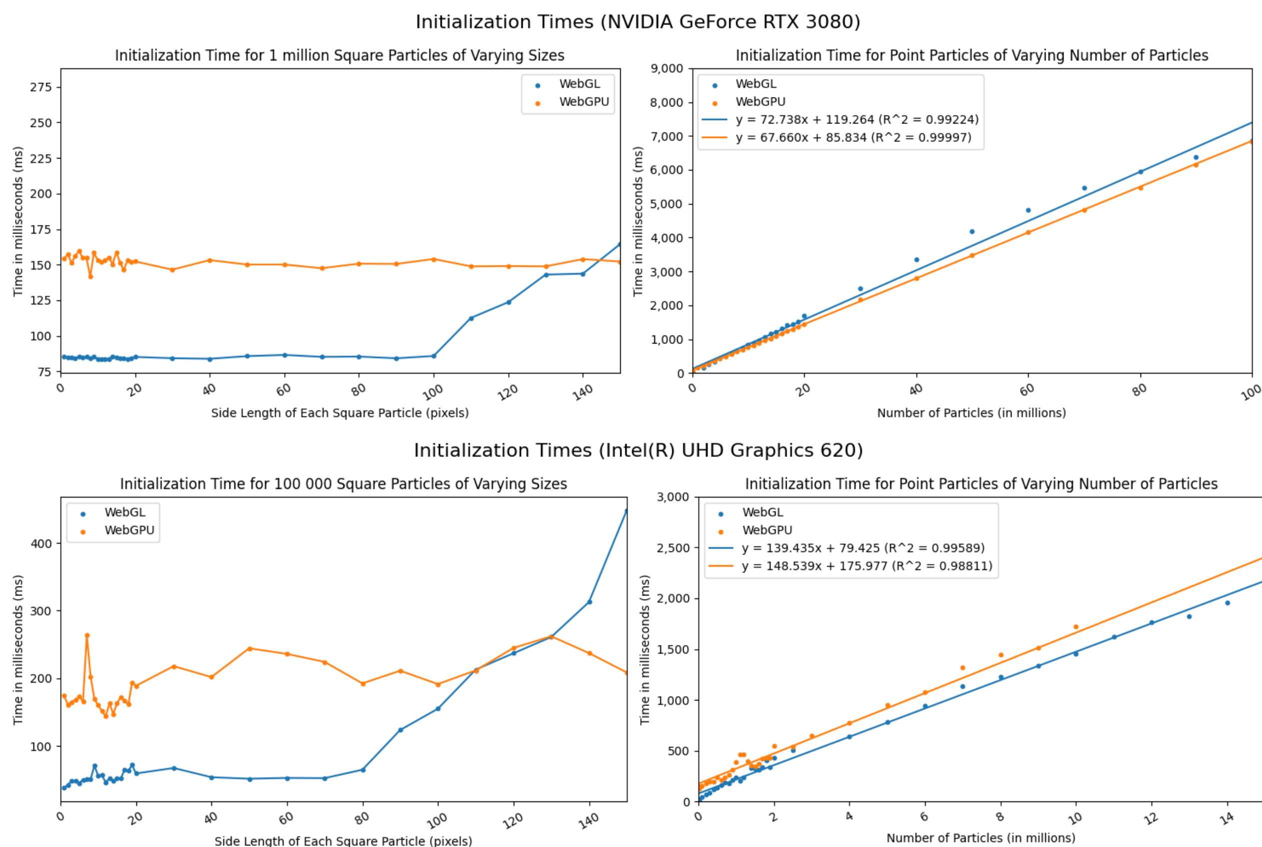
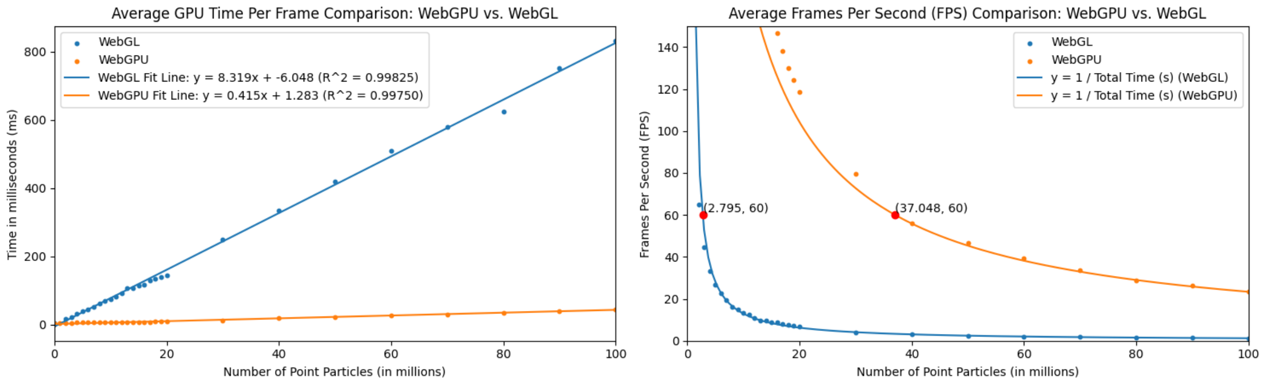


Figure A.1: Initialization Time Results



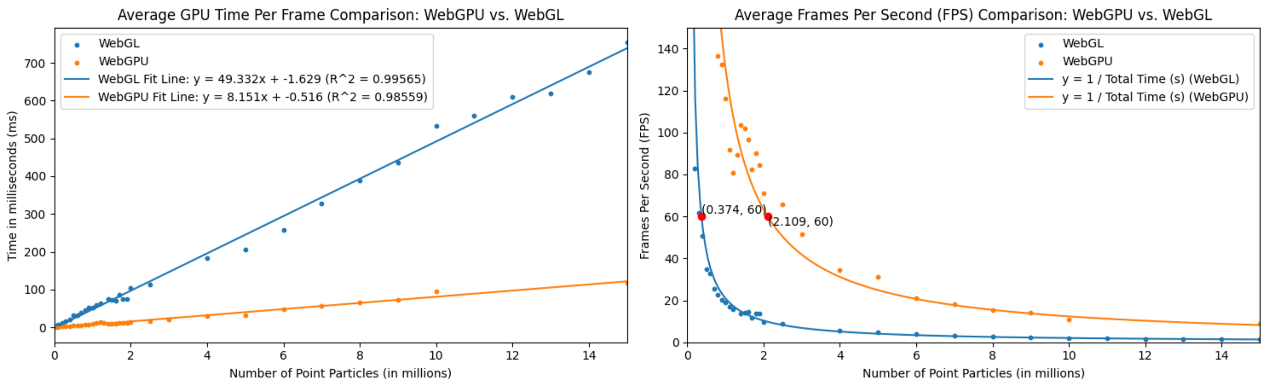
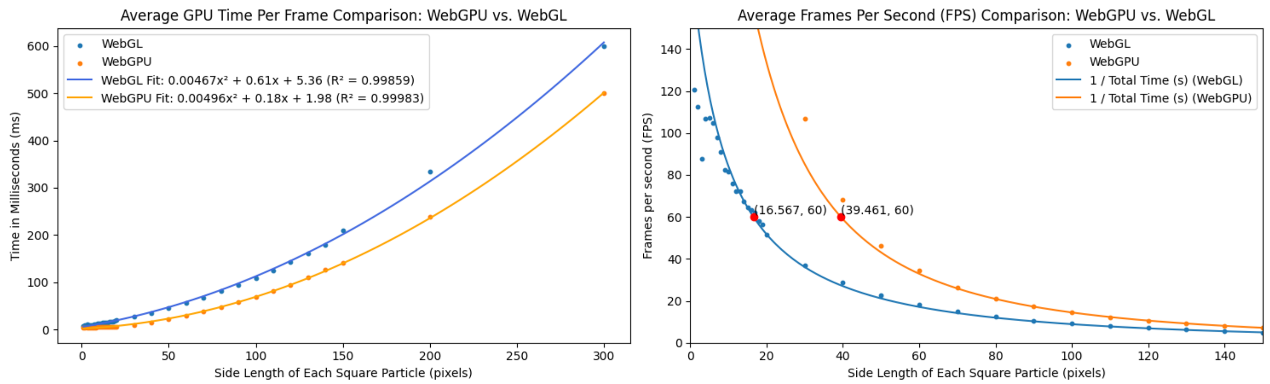


Figure A.2: Point Particle system Results

1 Million Square Particles of Varying Size (NVIDIA GeForce RTX 3080)



100 000 Square Particles of Varying Size (Intel(R) UHD Graphics 620)

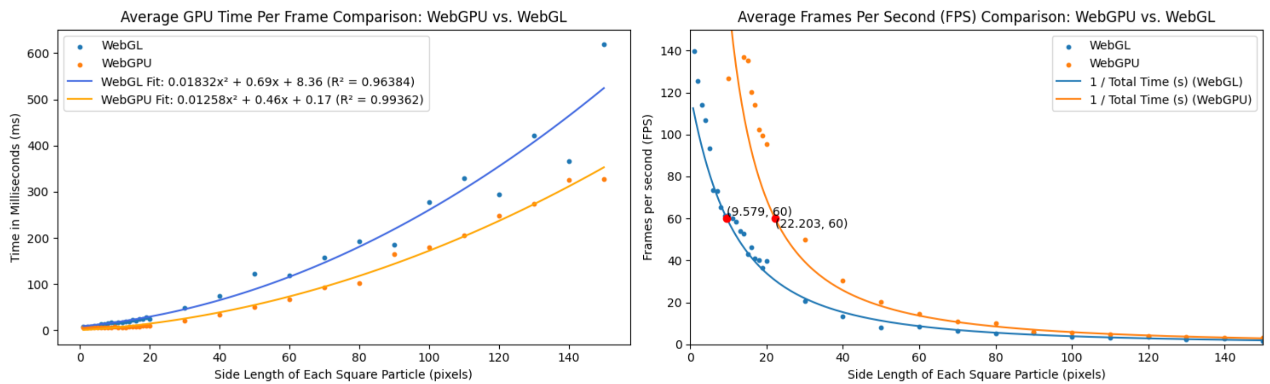
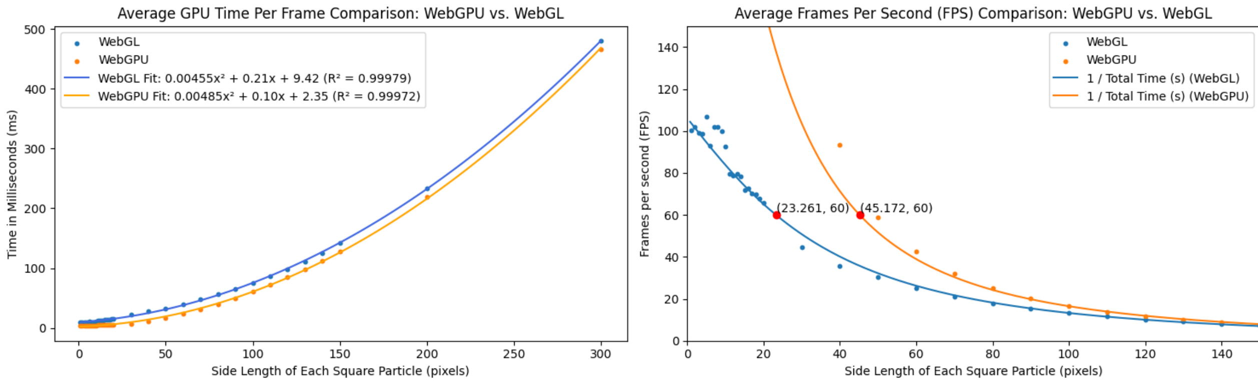


Figure A.3: Square Particle system Results

1 Million Textured Square Particles of Varying Size (NVIDIA GeForce RTX 3080)



100 000 Textured Square Particles of Varying Size (Intel(R) UHD Graphics 620)

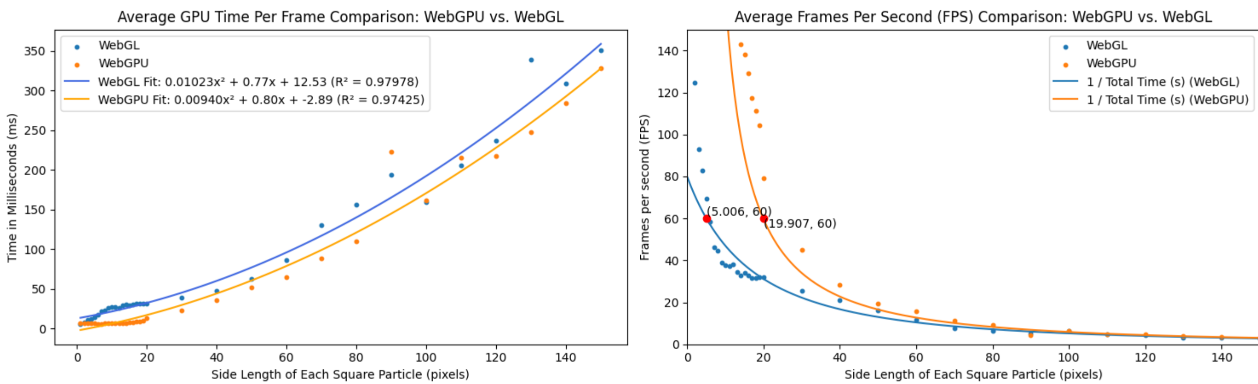


Figure A.4: Point Particle system Results

TRITA-EECS-EX-2024:966