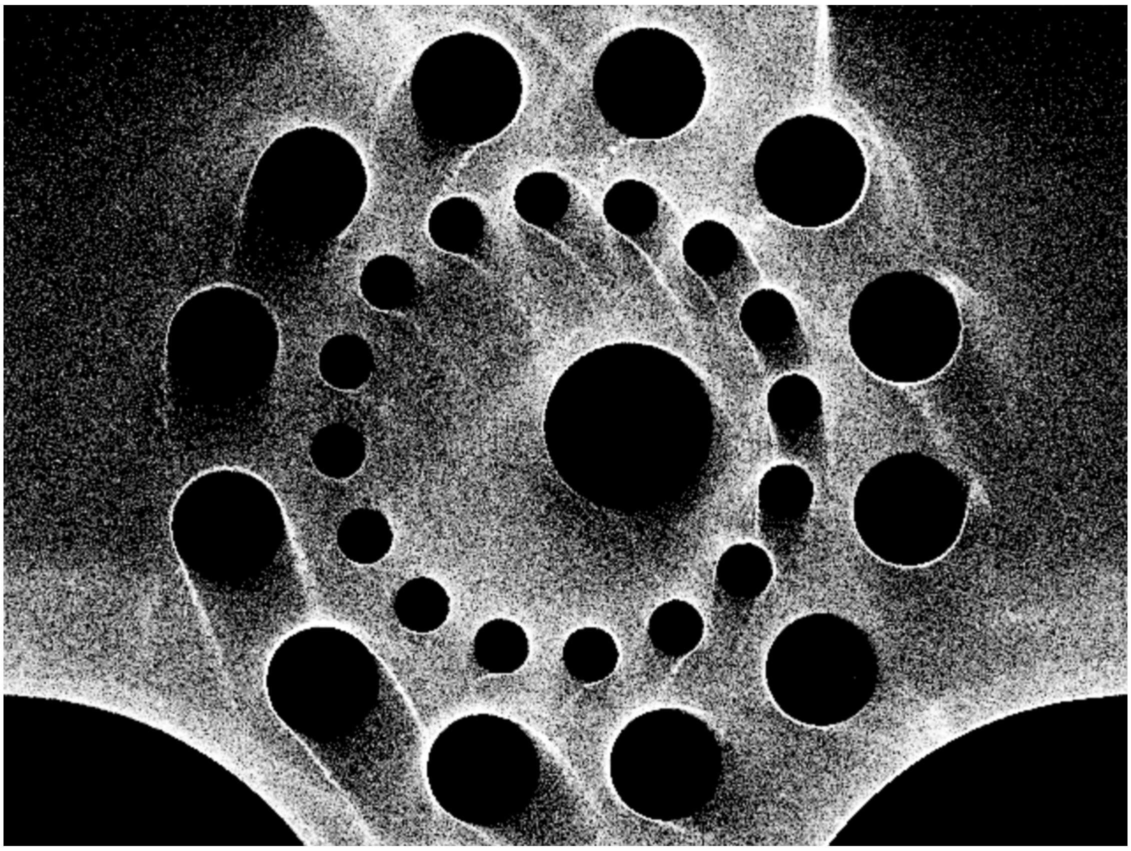

Optimizing a GPU-Accelerated Particle System for WebGL 1.0 using Extensions

Master’s thesis in Computer science and engineering

Lars Andersson, Oskar Lenschow

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Optimizing a GPU-Accelerated Particle System for WebGL 1.0 using Extensions

Lars Andersson

Oskar Lenschow

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Optimizing a GPU-Accelerated Particle System for WebGL 1.0 using Extensions
Lars Andersson, Oskar Lenschow

Supervisor: Ulf Assarsson, Department of Computer Engineering
Advisor: Mathias Westerdahl, Defold
Examiner: Erik Sintorn, Department of Computer Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An image of the particle system running on a web browser using an implementation covered in this thesis.

Typeset in LaTeX
Gothenburg, Sweden 2021

Optimizing a GPU-Accelerated Particle System for WebGL 1.0 using Extensions
Lars Andersson, Oskar Lenschow
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

An important part of games, and graphics in general, are particles. These can be used to represent a large variety of effects, such as smoke, water or sand. As gaming is popular on a wide variety of platforms with differing performance capabilities, these particle systems need to be implemented differently depending on the platform, both in what algorithms are chosen, as well as what methods are used to successfully implement them. This thesis explores the possibilities of optimizing a GPU-accelerated WebGL 1.0 particle system with collision and particle physics, through the use of suitable extensions, in order to create a system that outperforms an implementation with standard WebGL 1.0. The results show that implementing floating-point textures, multiple render targets, and instancing, does in fact increase performance to varying degrees, depending on what platforms and browsers are used.

Keywords: Particle System, GPU-Acceleration, WebGL 1.0, Optimization Technique, Web Browser, Extension.

Acknowledgements

Thank you to our friends and families for their continued support throughout this project and thesis.

We want to acknowledge the Defold Foundation for giving us the opportunity to work with them, while also providing us with support, tips, and much needed discussions around the topic.

We also want to thank our supervisor at Chalmers, Ulf Assarsson, for his thorough feedback, and Erik Sintorn for being our examiner, as well as being our guide at the beginning of this project, helping us shape it into something useful.

Lars Andersson, Oskar Lenschow, Gothenburg, December 2021

Contents

1 Introduction

Within computer graphics today, particle effects and particle systems are active fields of research and development. They are used to simulate a multitude of effects: smoke, snow, fog, fire, water, dust, and explosions. As these systems often require anywhere from thousands, up to millions, of particles to properly represent reality, these systems can benefit from being GPU-accelerated, in order for the particles to be simulated in parallel. To represent particle systems, there are different sets of particle types. Particles that make up homogeneous media, as with fog, act a certain way. This includes how they collide with each-other and the environment, and the way they interact with light. The particle types in heterogeneous or other types of participating media act similarly, but not in a homogeneous manner. An example of this is smoke [30].

Many particle systems that have more advanced behavior have some form of collision detection and particle physics. Particles used to simulate water need to be contained within the constraints of the surrounding container, and falling marbles have to bounce off the surface they land on. This particle type can also be used to create semi-rigid, deformable bodies, examples being strands of hair, or a piece of cloth. Particles can also be used for less advanced systems, such as simpler visual effects that do not have any impact on the surrounding environment. Examples of this are sparks from a gunshot, or simpler, non-interactive smoke. These are not as complex as particles within participating media are, but are therefore simpler to simulate and render, and are less reliant on high-performance hardware.

One area of computer graphics that often takes advantage of such particle systems is gaming. Many games today require particle systems to simulate the previously mentioned effects. To GPU-accelerate particles for gaming, graphics libraries such as OpenGL or DirectX are used, depending on which platform the game is to be built for, and what hardware support is required. A widely used graphics library is WebGL 1.0. It is, together with WebGL 2.0, the standard library that is used in web game development, and several game engines offer exportation to WebGL 1.0 if needed, such as Defold [7] and Unity [34]. The advantage of WebGL 1.0 over WebGL 2.0 is its extensive compatibility list, and that it can operate on most devices that are equipped with a GPU and that can run a web browser, as long as WebGL 1.0 enabled on that browser [33].

As WebGL 1.0 supports many different devices, it is also more restricted, as it implements less experimental techniques that would require more modern hardware. This in turn means that simulating particles for web-based games becomes restricted in the same manner. As this project was done in collaboration with the Defold Foundation and the Defold Engine, where WebGL 1.0 support is included, research was required in order to develop a physically based particle system that maintained high performance while still preserving low-end support. This thesis showcases the implementation and testing of the most suitable WebGL 1.0 extensions for optimizing a GPU-accelerated particle system with particle physics and environmental collision for the web, with favorable results.

1.1 Problem Definition

Several solutions already exist for creating and optimizing particle systems, however, a consistent problem is that more modern libraries and advanced techniques are used that are not supported on older hardware or for web development. The main aim of this master thesis is to find and test the most suitable WebGL 1.0 extensions to optimize a GPU-accelerated particle system with particle physics and environmental collision, while comparing the performance to a native WebGL 1.0 particle system with no extensions or improvements. This can be divided into two main questions:

To what extent can a GPU-accelerated particle system with particle physics and environmental collision for WebGL 1.0 be optimized through the use of available extensions?

What WebGL 1.0 extensions are the most appropriate for optimizing a particle system, and of these chosen extensions, which ones are the most beneficial for the performance of the particle system?

1.1.1 Delimitation

To delimit the workload, a specific type of particle system will be focused on. As the particle system will be GPU-accelerated, the particles are stateful, meaning that data is stored per particle for future use. This is used to simulate simple particle physics, as well as collision with the environment. Any other particle system types will not be considered.

All methods will be tested in 2D, rather than in 3D. As the primary focus of the system is particle movement and environmental collision, rather than light interaction or volumetric behaviour, they act similarly in 2D as they would in 3D. Creating the system in 2D simplifies the workload required to properly implement and test the different extensions and compare them.

2
Related Work

As mentioned previously, many solutions exist for creating and optimizing particle systems or similar, GPU-accelerated applications, using many different techniques. This chapter lists work that has been done previously to create different types of particles, their use cases, together with the different optimization methods that have been used for them.

Particle systems is a broad and active field of research. It has been used to study concepts ranging from the behavior of plasma trough plasmons, a quasiparticle that represents the quantum of plasma oscilation [4], to the brownian motion of nano-particles [35].

Within the field of particle systems lies the sub-field of particle systems in computer graphics. These particle systems are used to visually, and often behaviourally, represent smoke, fog, fire, water, dust, and explosions [1]. They have also been used more recently to simulate snow [8].

Graphics based particle systems can both be pre-rendered offline, as well as simulated in real-time. Pre-rendered particles have been used in movies, one example being the MPM method used by Disney [32] for physically accurate snow simulation. Typically, the techniques that are used for offline rendering tend to be substantially more advanced, and physically correct, compared to real-time implementations, as the time constraint is not on a frame-by-frame basis.

2.1 Particle System Optimizations

Many methods have been used to optimize particle systems within computer graphics. One such methods is through the use of GPU-acceleration, where workload is moved from the CPU to the GPU when possible. To achieve this, usually a pixel shader is utilized to parallelize the workload of simulating the particles. However, there has been progress made within the computer graphics field that has allowed the use of different approaches more suited for general computing, for example NVIDIAs CUDA, or compute shaders in OpenGL. These approaches fit well within the purpose of simulating particle systems, and have seen a lot of research [6]. For example, one major advantage of compute shaders is the ability to keep all information stored upon the GPU, and share information between GPU threads within thread groups, as each thread group has a set amount of memory shared among the threads [1]. In a paper investigating advantages of acynchronous compute where compute shaders are utilized alongside the regular graphical stages, a performance gain up to about 34% is possible to achieve when implementing a particle system utilizing asynchronous compute, compared to sequential execution [6]. In another paper, particle system performance is compared between a CPU based implementation and a GPU based implementation using CUDA, where the performance increase of using a GPU is significant [21]. As these types of shaders are not available for use in WebGL 1.0, they can not be used to optimize a particle system based on this.

In a paper by Hegeman et al. they propose an effective method of simulating particle-based fluids using a GPU, where a dynamic quadtree structure is utilized to accelerate the computation of forces between particles [20]. In another paper, fluid is again simulated on the GPU, but the focus is instead put on a new method of modeling particle coupling, using Lagrangian mechanics to remove the need for global sorting [22].

Another method that has been used to create particle systems is the use of a geometry shader [1]. Rather than optimizing performance however, this is used to simplify the user experience. This shader is used to generate the vertices for particles that are represented through billboards. However, as performance is not the focus of this shader type, other methods are preferable.

In addition to the shader techniques mentioned above, there is a newer shader type that replace parts of the graphics pipeline. This shader is called a "Mesh Shader", which improves performance using a more optimized approach, as it is more scalable, flexible, and can parallelize work more efficiently due to meshlets [23]. This can be used to represent particle systems. However, as with the previously mentioned compute shader, this technique is not available for use within WebGL 1.0.

As the graphics library WebGL 1.0 is more restricted than many other alternatives, other methods are required. To optimize GPU usage within WebGL 1.0, a technique called N-buffering has been used [5] and proved to be successful in certain scenarios with the right combination of buffers. It works by having more than the two standard buffers (back buffer and front buffer) for rendering and displaying a frame. Instead, multiple back buffers can be rendered to, before one frame is displayed on the front buffer. This technique however does not put focus on particle systems specifically, and the optimizations made are more general purpose.

In this master thesis, focus is instead put on the extensions available for WebGL 1.0 to explore the optimization options that are appropriate for particle systems. As the particles will be GPU-accelerated, have particle physics and collision with the environment, the extensions are chosen based on this. As WebGL 1.0 only has access to regular vertex and fragment shaders with texture data, the extensions are based on color data, texture count, and the amount of render calls required. The specific extensions that were picked are further explained and motivated in 3.4.

3
Particle Optimization

This chapter covers the required theory behind graphics programming on the GPU, the required steps to properly simulate and render particles in WebGL 1.0, together with the extension options that were chosen. The implementation details of the chosen techniques are covered as well.

3.1 GPU Programming

As a GPU executes differently from a CPU, using hundreds or thousands of cores to execute in parallel, GPUs have to be programmed in a specific way. A CPU can be programmed in a serial language such as C or Java, while a GPU is programmed using shader languages, where the same shader code is executed in parallel across the available GPU cores. There are several alternative languages for programming shaders. Examples are HLSL (High-Level Shading Language) by Microsoft [27] and GLSL (OpenGL Shading Language) by Khronos Group [18]. Shader code is written for shader programs, which are programs that executes directly on the GPU itself. As the most commonly used and most important shader programs are the ***Vertex Shader*** and the ***Fragment Shader***, these are explained in more detail:

> ***Vertex Shader***. Most often the first programmable shader program in the rendering pipeline that, given vertices as input, outputs one processed vertex per vertex input. These vertices are later passed to a post-processing stage, after optionally going through tessellation or a geometry shader, where the resulting primitives are rasterized to create fragments for the fragment shader, covering the complete mesh.
>
> ***Fragment Shader***. Also known as the pixel shader, is a program that executes once per fragment received from the rasterization stage, and outputs one single color for that specific fragment.

To access the GPU from the CPU, graphics API's are required. These are used to set up instructions and information needed by the GPU, such as the vertices and their corresponding data, in order for it to know what data to process, and how to render said data. Examples of these instructions are what frames that should be rendered to, or what type of filtering that should be used. As with shader languages, there exists several alternative graphics APIs, many of which correspond to a specific shader language. Examples of these are DirectX by Microsoft [26], OpenGL by Khronos Group [16], and the API used in this thesis project, WebGL by the Mozilla Foundation and Khronos Group [28].

The APIs listed above are mostly used for graphics programming specifically. They can still be used for general purpose GPU programming, but as mentioned in Chapter 2, there are other GPU APIs meant for general purpose computing. Examples being CUDA by NVIDIA [3], or the compute shader by OpenGL [2]. These can be used for arbitrary computations that do not directly require the rendering of triangles or pixels. Given that this report covers the optimization of a particle system meant for WebGL 1.0, the implementation is limited to the techniques and shaders available in the WebGL 1.0 API, along with the language GLSL. This means that the only available shaders are the vertex shader and the fragment shader, and no compute shading alternatives exist.

3.2 GPU-Accelerated WebGL 1.0 Particles

The main idea of GPU-accelerated particles is the utilization of graphics hardware, in order to parellalize the simulation of each particle. This enables the rendering of a much greater amount of particles compared to a system simulated completely on the CPU. These particles can be stateless, where the behavior of the particle is completely calculated based on an initial value of the particle, and elapsed time. More advanced particles however, are the stateful particles, in which each particle possesses several attributes. These attributes can be anything, such as position, velocity, mass, orientation, etc. This type of data can be processed through the use of a general purpose shader, such as the compute shader or CUDA, but as WebGL 1.0 is more restricted, the only available shaders are the vertex shader and the fragment shader. In order to decouple the particles and their attributes from the CPU, each particle-related data is stored within textures, where each texel-color represent the data of one or more particles. As an example, an RGB texture can be utilized to let each texel r/g/b color represent the x/y/z coordinate of a particle, in which the world the system exists within. Each particle is updated in parallel through a pixel shader, which reads and writes to the textures containing the particle data.

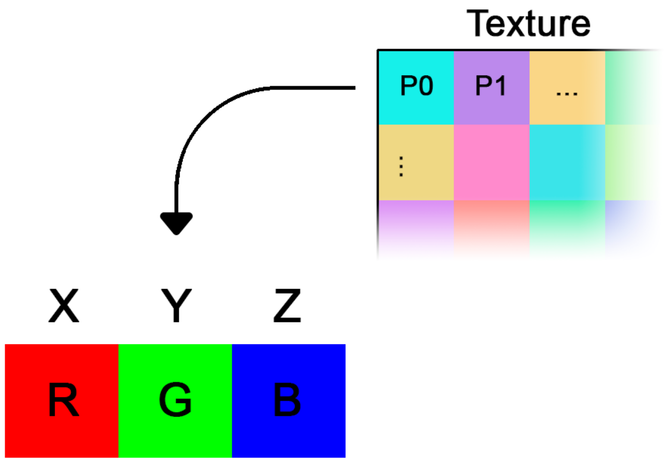

Figure 3.1: Illustration of data storage within a texture. Each pixel in the texture is represented as an RGB color, where each color channel represents a coordinate element of a particle.

3.3 Baseline Solution

A baseline solution was created using native WebGL 1.0, in order to later be upgraded using extensions. The purpose of this implementation was to have a working solution that could be used to compare the performance increase of each extension. An important demand of the baseline was for it to be compatible with a broad spectrum of platforms, including different browsers and hardware. One major restriction in native WebGL 1.0 is the limited color depth within the textures [14], which is explained in more detail in chapter subsection 3.4.1. One byte of data per attribute is too small, even for a baseline, as this would result in a world with a maximum of 256 (2^8) available positions in each dimension. In the baseline implementation, RGBA textures with a color depth of one unsigned byte per color channel are utilized, where attributes that demanded a higher precision of data are encoded within two bytes per attribute. The baseline solution utilizes only two attributes for each particle, namely 2D positions and velocities. This results in two double buffered textures in which one contains the positions of particles, and the other one contains the velocities. The values are also scaled to fully utilize the dynamic range of the two bytes, instead of a one-to-one relationship of values. The scalar is calculated based on the the size of the particle system. The usage of two attribute textures means that two draw passes are needed for each update of the particle system, as native WebGL 1.0 does not allow rendering to multiple textures. The first draw pass updates the positions of the particles, and the second updates the velocities.

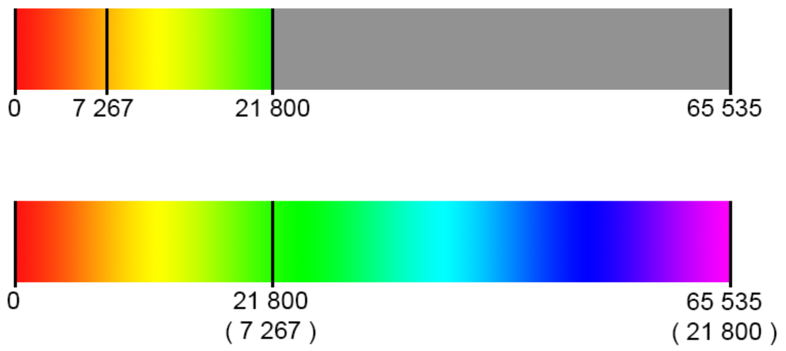

Figure 3.2: Illustration of utilizing the full dynamic range of two bytes. The top image represents a particle system consisting of a maximum of 21 800 coordinates, and the bottom image represents the same system, scaled up to 65 535, increasing the accuracy of the coordinates.

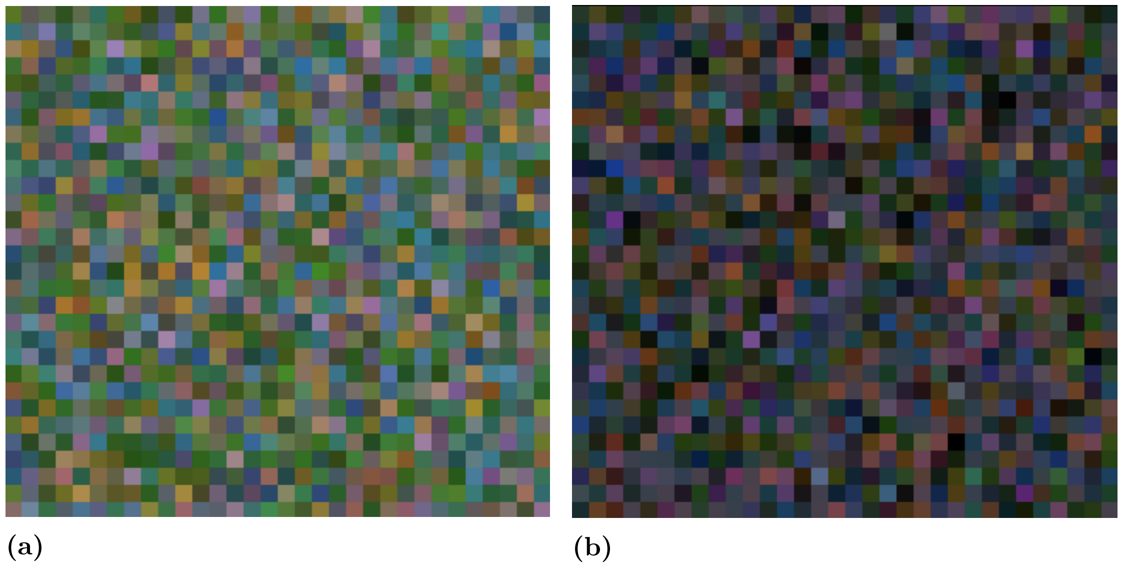

Figure 3.3: Textures containing attribute data for 1024 particles. a) Position data, x-coordinate encoded in red and green, y-coordinate encoded in blue and alpha. b) Velocity data, vx encoded in red and green, vy encoded in blue and alpha.

When rendering the particles, particle indices are stored as vertex attributes for the vertex shader to utilize. These indices contain the UV-coordinates of each particle position within the attribute textures, which allows the vertex shader to translate the indices into vertices, by sampling the attribute textures.

In figure 3.4, the pipeline for the baseline solution is shown. As the obstacle updating is mostly CPU-bound, and because no extensions for optimizing the last draw call exist, most optimizations will be applied to the particle update stage of the pipeline. The baseline solution can be divided into two separate stages. The first stage covers the setup of the system, and the second stage is the periodic behaviour of a frame. When initializing the system, the WebGL context is first retrieved, shaders compiled and framebuffers set up, followed by the initialization of the particle attribute textures and the obstacle texture. The second stage begins by updating all the obstacles, this covers moving, rotating and drawing the obstacles to the obstacle texture. This is followed by updating the particles. Each particle samples the position and velocity from the corresponding attribute textures, moves according the physics rules given and then samples the obstacle texture at the new position in order to detect a collision. If a collision is detected, the new position and velocity is calculated. The final position and velocity is then rendered to the back buffers of the position and velocity textures. Due to the render target limitation of native WebGL 1.0, this work needs to be carried out twice together with a texture swap. Finally, when both the obstacles and particles are updated, the new particle positions are rendered to the canvas, followed by some statistical calculations, and scheduling of the next frame by a requestAnimationFrame call. Figure 3.5 shows the baseline solution running with 1 million particles and 31 dynamic obstacles.

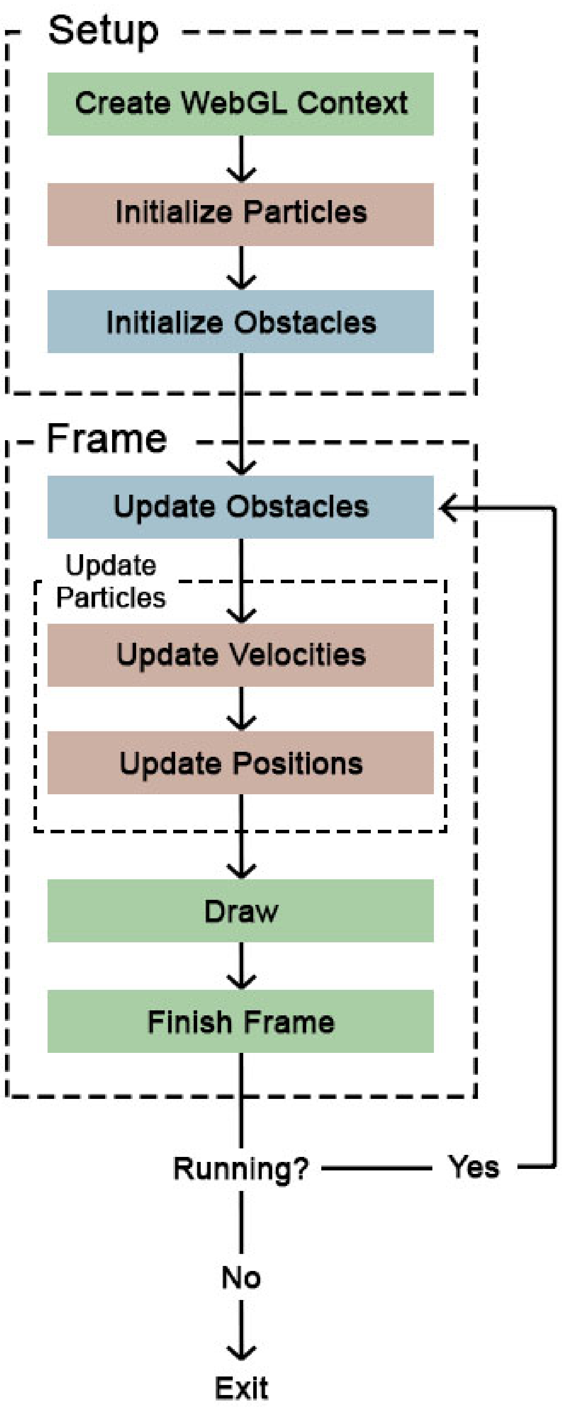

Figure 3.4: Flowchart of the baseline implementation.

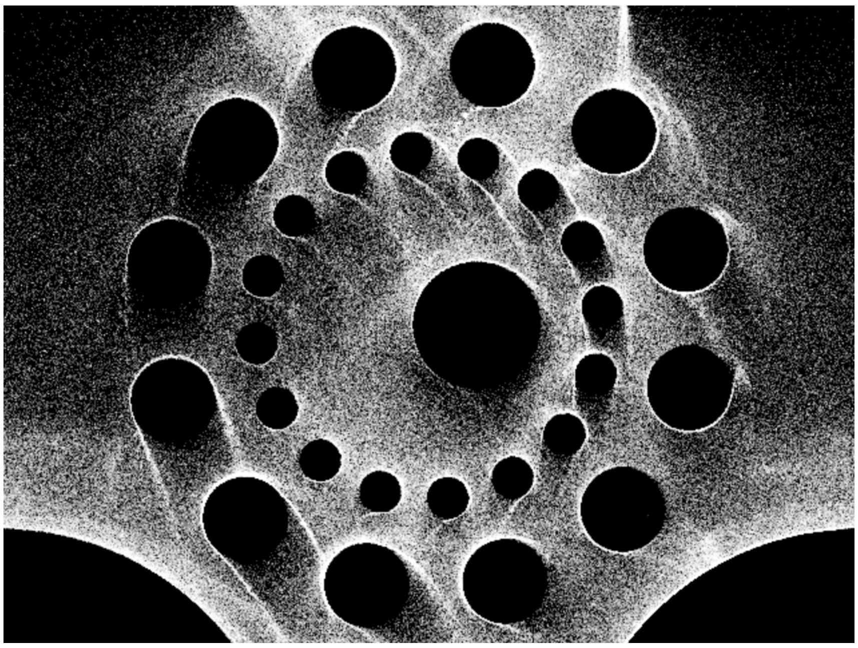

Figure 3.5: Baseline application with 1 million particles and 31 dynamic obstacles.

3.4 Suitable Optimization Techniques

To optimize the particle system through the use of extensions, three main techniques were chosen based on suitability for particle systems, in addition to their availability for WebGL 1.0. The first technique involves the amount of data that can be stored per pixel within a texture, and how increasing it enables the removal of unnecessary computing steps in the system. The second technique chosen was instancing, increasing performance by either decreasing draw calls for the obstacles that interact with the particles, or by decreasing the memory required to set up a draw call. The last techniques involves the amount of render targets that can be rendered to in the fragment shader.

As these extensions are meant to facilitate and improve a GPU accelerated, physics and collision based particle system, both a physics algorithm and a collision detection technique needs to be implemented. The chosen methods are, however, arbitrary, as the primary focus is to improve the system surrounding these algorithms, rather than the algorithms themselves. The only requirement is that both the physics and collision detection is calculated on the GPU, to maintain GPU acceleration, and to ensure that the extensions improve the foundation upon which these algorithms lie.

3.4.1 Color Depth

An implementation in native WebGL 1.0 comes with limitations. The color depth of textures is one of these limitations. At most, one byte of data is the limit to store per color channel [14], which consequently leads to an extremely low resolution of data per texel, unnecessary large sizes, or too many textures.

There exists extensions which increases the color depth threshold to either a 32 bit float [12] or a 16 bit half float [13], if the hardware supports it. When increasing the color depth, more data can be stored within a single pixel, possibly reducing the amount of both draw-calls and texture look-ups. The functionality of these extensions is enabled by default in native WebGL 2.0, and OpenGL ES 3.0+ [15][24]. One way of utilizing this is to either store a higher precision value within each pixel, or possibly encoding multiple values within one single pixel.

To expand the maximum possible data storage within each pixel for the attribute textures, OES_texture_half_float extension [13] was enabled. This allowed each pixel to use 16bit of storage within each color channel, instead of 8bit, ultimately eliminating the need for encoding attributes over multiple channels. In order to fully utilize the increased storage, the scaling implemented within the baseline solution was still kept.

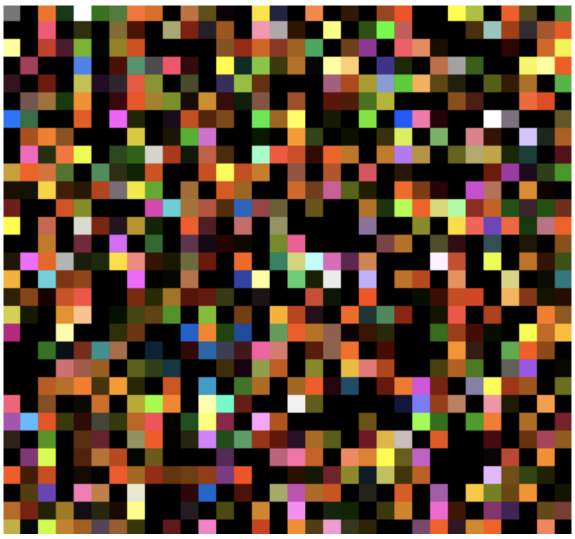

Figure 3.6: Texture containing attribute data for 1024 particles with half float precision. Position data encoded in red and green, velocity data encoded in blue and alpha.

Given the fact that the texture contains more data, one downside of this method is that more memory is needed for one texture. For simpler implementations that only require one byte of data, using textures with higher color depth might therefore be redundant.

The expected performance outcome of this technique is that the time spent updating the particles only takes 37.5% of the time compared to the baseline solution, as only two reads followed by one write is required, instead of six reads and two writes.

3.4.2 Instancing

Instancing is a broad term within computing, with the main purpose of reusing the same data multiple times with what are called instances. Within computer graphics, instancing is a technique that can be used to increase performance when rendering multiple objects that have the same vertex data [29].

In order to allow particles to collide with the environment, obstacle data also has to be rendered to textures, in order for that data to be easily accessible by particles in their respective shader. But as the obstacles themselves need to be movable and interactable, they are updated on the CPU. To draw multiple obstacles effectively, using instancing is preferable. Without instancing, two main alternatives exist for drawing the same type of mesh multiple times. The first alternative is to utilize one draw call for each object, and let world transformations such as scale, position or color be represented by uniforms passed to the shader. The major drawback of this method is that separate draw calls for each object impacts performance negatively, as the performance lost scales linearly with the amount of meshes used. However, as separate draw calls are utilized, the memory restrictions of uniforms can be ignored, as the uniform data is replaced each draw call.

As an alternative solution, attributes for each object can be stored within a vertex buffer array, with each entry corresponding to one vertex in a mesh. This reduces the amount of draw calls to one single draw call, assuming that the meshes use the same shader. Replacing the previously mentioned method with this can drastically improve performance. However, one drawback of this method is the amount of memory required. As each vertex in a mesh requires a unique vertex data entry, the memory scales by the number of vertices per mesh. As an example, two quads that have the same mesh and have the same color would require at least 8 color entries, as a quad has at least 4 vertices. Another drawback is that overhead is moved to the CPU, as the vertex data needs to be computed each frame before being passed to the GPU.

To render the type of obstacles covered in this thesis, using point sprites is an alternative, as only one point is required, around which a texture can subsequently be drawn. Another method is to use a vertex id to reuse the same list of vertices multiple times. However, these methods are not supported in WebGL 1.0, and no suitable extensions exist to implement them.

With instancing, performance should in theory be significantly closer, or identical, to the solution using vertex data attributes, while also keeping the memory usage, and the CPU overhead, at a minimum. This works by letting the vertex shader reuse vertex data multiple times, only traversing the buffer array by a specified amount per instance, often once per mesh, instead of once per vertex. Using the previous example of two quads with the same color, this would result in one single color entry, rather than 8, while still only requiring one single draw call.

	Uniforms	Vertex Data	Instancing
Memory	500 Bytes	4 000 Bytes	500 Bytes
Draw calls	100	1	1

Table 3.1: Example of performance and memory comparison between a solution using uniforms for world transformation data, a solution using vertex data, and a solution using instancing. 100 polygons with 8 vertices each, using 3 bytes of color data and 2 bytes of position data.

To test the capabilities of instancing in WebGL 1.0, two different solutions were created. The first one using multiple draw calls and uniforms to minimize memory usage, and the second one using a large vertex data array to minimize performance loss from draw calls. The second solution was used as the baseline for all performance comparisons. To implement instancing as an optimization to these baseline solutions, the WebGL 1.0 extension ANGLE_instanced_arrays [9] was used. As the primary goal of instancing for the particle system was to increase the performance related to the obstacles, all obstacles used in the baseline solution were reworked to be instanced. It is worth noting that the memory usage, compared to performance, is in most cases negligible for the simple type of obstacles used in this thesis. To motivate the implementation of instancing in a realistic project, significantly more complex meshes would be reused.

3.4.3 Rendering Targets

Being able to render to multiple textures simultaneously can be of great advantage, for example, when building a G-buffer for deferred rendering. In a context of GPU-accelerated particles, this technique enables storing a larger quantity of data over a multiple set of textures, within a single draw call. Native WebGL 1.0 is however restricted to one single render target [14], consequently complicating an implementation where particle attributes does not fit within one single texture. The extension WEBGL_draw_buffers [11] changes this however, enabling a fragment shader the rendering to multiple textures simultaneously. One major downside to this extension however, is the absence of browser compatibility on mobile devices.

Similarly to increasing color depth, the expected outcome is that the time required to update the particles is decreased as this extension removes the need to swap textures, removing CPU overhead. In addition to this, three texture reads are removed, as the velocity texture and position texture can be accessed and utilized in the same render pass.

3.5 Extension Compatibility

The following tables illustrate browser compatibility for the three chosen extensions: OES_texture_half_float, ANGLE_instanced_arrays, and WEBGL_draw_buffers, as well as the browser compatibility of WebGL 2.0 for comparison. The first table shows desktop browsers, and the second table shows mobile browsers. Note that all version numbers listed refer to when the extensions and WebGL 2.0 were natively supported. Some features could be enabled in earlier builds through experimental flags. Device compatibility is not covered, as the most common devices support the extensions that the available browser supports. No valid sources could be found to confirm any exceptions.

	Half Float	Instancing	Draw Buffers	WebGL 2.0
Chrome	27	32	36	56
Edge	14	12	17	No
Firefox	29	47	28	51
Opera	15	19	23	43
Safari	8	8	9	No

Table 3.2: Desktop browser compatibility. Each cell represents the browser version in which support was first introduced.

	Half Float	Instancing	Draw Buffers	WebGL 2.0
Chrome Android	27	32	No	70
Firefox Android	Yes*	Yes*	No**	63
Opera Android	14	19	No	46
Safari iOS	8	8	No	No

Table 3.3: Mobile browser compatibility. Each cell represents the browser version in which support was first introduced.
* No valid sources for version numbers could be found.
** Could not be confirmed with sources, instead based on testing.

3.6 Collision Detection

As more complex and precise GPU-based collision detection solutions, such as raytracing, require more advanced hardware, or newer libraries than WebGL 1.0, a texture-based collision had to be implemented. As previously mentioned, the actual method behind the collision detection is arbitrary, as long as it is GPU-accelerated to benefit from any improvements made by the extensions.

Because of the fact that inter-particle collision is not handled, the collision that was implemented was between particles and their surrounding obstacles. Obstacles are stored within an additional texture representing the whole particle system, thus enabling easy collision detection for the particles by sampling their world-coordinate within the obstacle texture from the update shader. The color of each obstacle represent the normal of the obstacle (red and green), together with the distance to the surface of the obstacle (blue) in pixels.

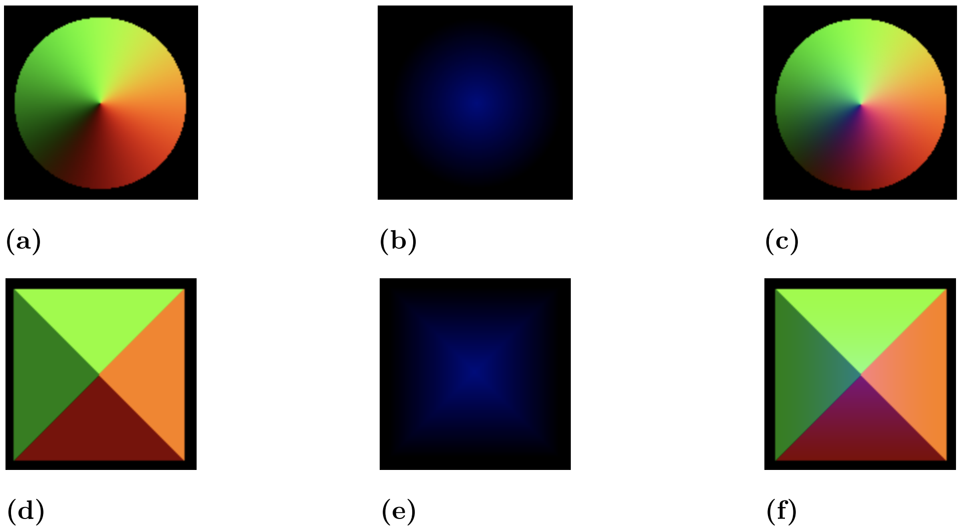

Figure 3.7: Examples of an obstacle. (a & d) color represents the normal vector to exit the obstacle. (b & e) color represents the distance in pixels to the surface of the obstacle. (c & f) final color.

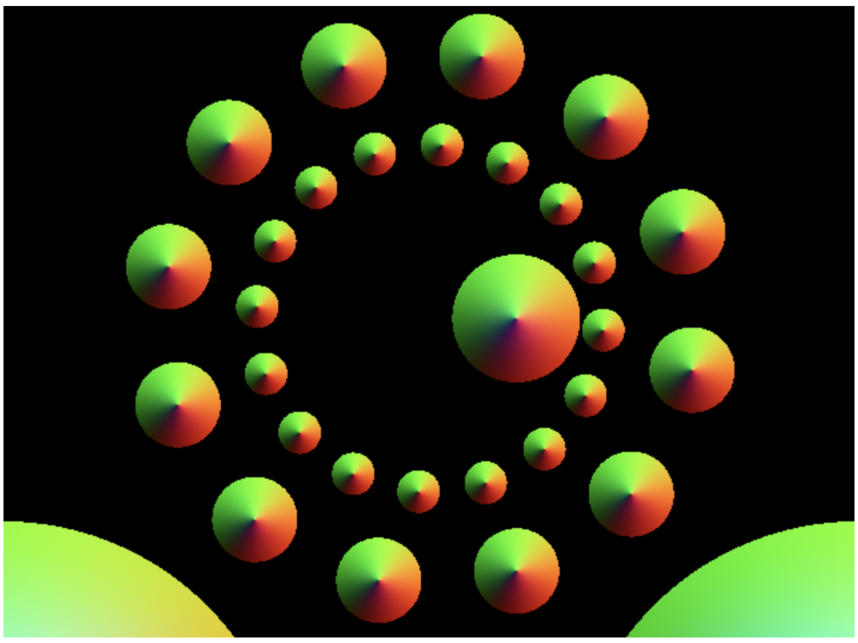

Figure 3.8: The world texture containing all the necessary obstacle data for particles to handle collision.

3.7 Physics Algorithm

In order for the particle system to emulate reality, a suitable algorithm is needed. There exist many different algorithms for a multitude of purposes, as smoke requires an algorithm that would not be appropriate to simulate water. As the primary focus of the particle system in this project is to collide with the environment, and not to interact with light, the physics algorithm should reflect a particle type that is suitable for this purpose.

As with the collision detection, the actual physics behind the particles is arbitrary, as long as some stateful behavior between frames is maintained for each particle. The chosen way to include both this type of behaviour, together with environmental collision, while also maintaining behavior that fits in, for example, a game, is the use of an algorithm that forces constant movement based on a particles previous position, and the surrounding obstacles. The chosen algorithm is therefore one that simulates solid particles that fall towards the y-axis using a gravitational force, with wind along the x-axis. An example is to visualize this is sand falling, bouncing and blowing along on surfaces. Equations 3.1 and 3.2 show how the new position P is derived.

$$p_{new\_vx} = p_{vx} + wind \tag{3.1}$$

$$p_{new\_x} = p_x + p_{new\_vx} \tag{3.2}$$

$$p_{new\_vy} = p_{vy} + gravity \tag{3.3}$$

$$p_{new\_y} = p_y + p_{new\_vy} \tag{3.4}$$

4 Method

This chapter will cover the basics of the testing environment, such as how the test suite was constructed, what platforms and browsers were tested on, and what metrics were utilized.

4.1 Test Suite

The main aspect to cover during benchmarking is the execution time that is spent on the GPU. Measuring this time within the context of a WebGL application, however, proved especially difficult due to the high amount of control over the rendering pipeline that is handled by the browser, in contrast to the limited control the developer has. In addition to that, each different browser has a unique implementation of WebGL, with performance differing quite substantially between browsers. In order to get an accurate time measurement of the workload performed on the GPU, the extension EXT_disjoint_timer_query was utilized [17]. This extension granted the ability to asynchronously query the time elapsed over a specific set of GL commands, without stalling the rendering pipeline. However, one major drawback of this extension is the browser compatibility. Due to the nature of the extension, most browsers require specific security flags to be toggled off in order for the extension to function, and certain browsers, such as Safari, or most mobile browsers, offer no support of the extension.

A test suite was constructed, simulating a 640x480 world consisting of 31 dynamic obstacles, and automated to run for 3000 frames, repeated a set amount of times, collecting measurement data. After each test iteration, the amount of particles were increased by a set amount, depending on the underlying platform that the test was currently running on. The reason for the different particle count is that a desktop PC with a dedicated GPU can simulate and render a significantly higher amount of particles than that of an older phone. The testing was done individually for each browser, platform, and extension technique combination, with the last iteration being one that had a frame time close to 16ms. This was to maintain a reasonable testing interval within 60 frames per second.

To test the performance difference between the optimization methods for the obstacles, another test suite was constructed. This test is similar to the particle test suite, with the main difference being that for each iteration, the amount of obstacles were increased, instead of the particles. As with the particle test suite, each increment had to be different in size per platform, depending on the underlying hardware.

In order to get an accurate measurement without stalling the rendering pipeline, and without compromising the actual frame time, only one frame was captured at a time. As the measurements were conducted asynchronously, the rendering could be carried out normally, while awaiting measurement data, only capturing a new frame when a result had been made available.

```

function draw(dt)
{
    ...
    if (measurement_data_available)
    {
        storeMeasurementData();
        ext.beginQueryEXT(...);
        frame(dt);
        ext.endQueryEXT(...);
    }
    else
    {
        frame(dt);
    }

    requestAnimationFrame(draw);
}

function frame(dt)
{
    updateObstacles(dt);
    updateParticles(dt);
    renderParticles(dt);
}
```

Listing 4.1: Simplified JavaScript code, showing the structure of the main program

The main metric that was used for benchmarking is the average time in milliseconds, together with the standard deviation, spent on the GPU per frame. When benchmarking the obstacle optimizations, memory requirement was used as a metric as well, in order to compare it with the average frame time.

4.2 Platforms

The different platforms used for testing were chosen based on performance difference, in addition to what was available for usage during the project. To properly test the different solution, a relatively modern computer with a dedicated GPU (NVidia Geforce GTX 1070) was used as a high-range testing platform. To cover mid-range performance, two different laptops were used, both using similar Intel Iris Plus integrated GPUs. These were the Lenovo Yoga C940 and the 2018 MacBook Pro 13". The reason for having two laptops with such similar performance is that the Lenovo runs Windows 10, and the MacBook runs Mac OS Big Sur, so that the testing could be performed on more than one computer operating system. For low-range platform testing, a OnePlus 3 android phone was chosen, as the Qualcomm Adreno 530 GPU is significantly weaker those previously mentioned, and because mobile performance is one main reason for optimizing the particle system in WebGL 1.0.

4.3 Browsers

The browsers were chosen based on compatibility for testing. As the compatibility list for the optimization extensions is not the same as the list for disjoint timer extension, these browsers do not represent what browsers support the chosen methods, but instead shows what browsers support the test suite. The table showing what browsers support the different optimization techniques can be found under section 3.5.

	Chrome	Edge	Opera	Firefox
Desktop (GTX 1070)	Yes	Yes	Yes	Yes
Lenovo Yoga C940	Yes	Yes	Yes	Yes
MacBook Pro 2018	Yes	Yes	Yes	Yes
OnePlus 3	No	No	No	Yes

Table 4.1: Browsers and platforms tested on.

5
Results

The following chapter is divided into two different sections that cover separate parts of the particle system. The first section presents the performance results of the techniques used to optimize the particles, while the second section covers the optimizations implemented for the surrounding obstacles.

5.1 Particle Performance

To properly benchmark each platform, the particle count interval had to be different for each test case, as a desktop PC with a dedicated GPU outperforms and older android phone significantly. Each step to increase the particle amount also had to be based on this interval, to maintain a similar resolution for each result. Each individual test was performed up until the frame time exceeded that of 16 ms (60 frames per second), to maintain a maximum value for each test, and a minimum performance requirement.

All chromium based browsers (Google Chrome, Opera, Microsoft Edge) performed similarly on almost all tested platforms, with the average performance difference being around 5% for the baseline solution, across all browser, per platform. On the Lenovo however, the performance difference of the baseline solution between Google Chrome and Microsoft Edge was as high as 74%. As this is an alarmingly large difference, the same tests were run multiple times, yielding very different results each time. After a restart of Microsoft Edge on the Lenovo laptop, the performance could decrease or increase by around 30%, without any clear cause. This was however only an issue with Microsoft Edge and Mozilla Firefox on specifically the Lenovo, with all other platforms and browsers yielding consistently stable and similar results each time. Therefore, these results were not taken into account when evaluating the optimization techniques.

When excluding the results from Microsoft Edge and Mozilla Firefox, it can be seen in Figure 5.1 that floating point textures significantly outperforms both the baseline solution and the implementation using multiple render targets, by 26% on average, on Google Chrome. The frame time of multiple render targets, regardless of particle count, is on par with the baseline solution, with no notable performance gain. On Opera on the other hand, all implementation alternatives, including the baseline, are within 4% of each other, and no technique yielded any significant performance increase, as can be seen in Figure 5.2.

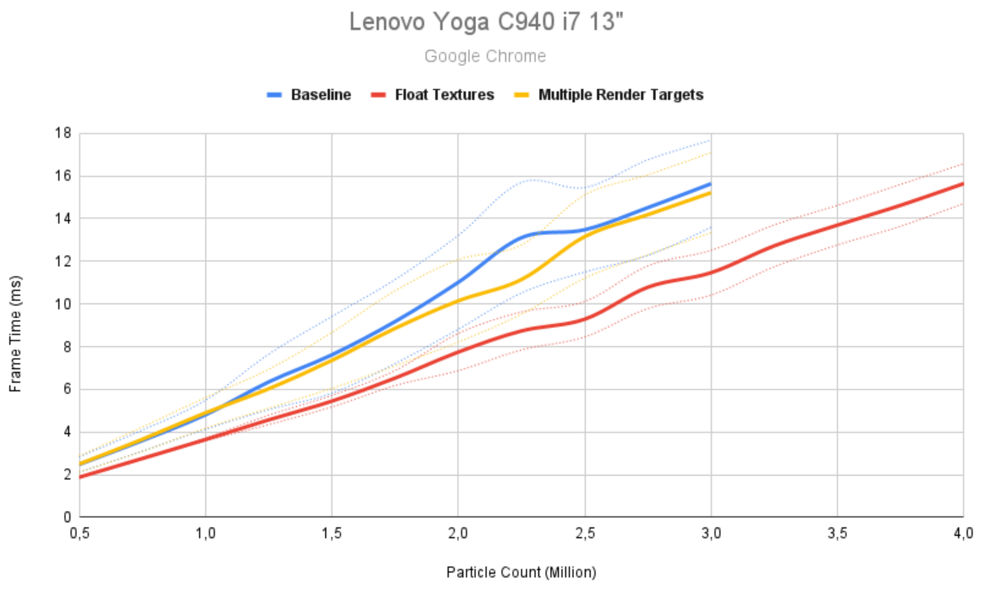

Figure 5.1: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the Lenovo Yoga C940 running Google Chrome.

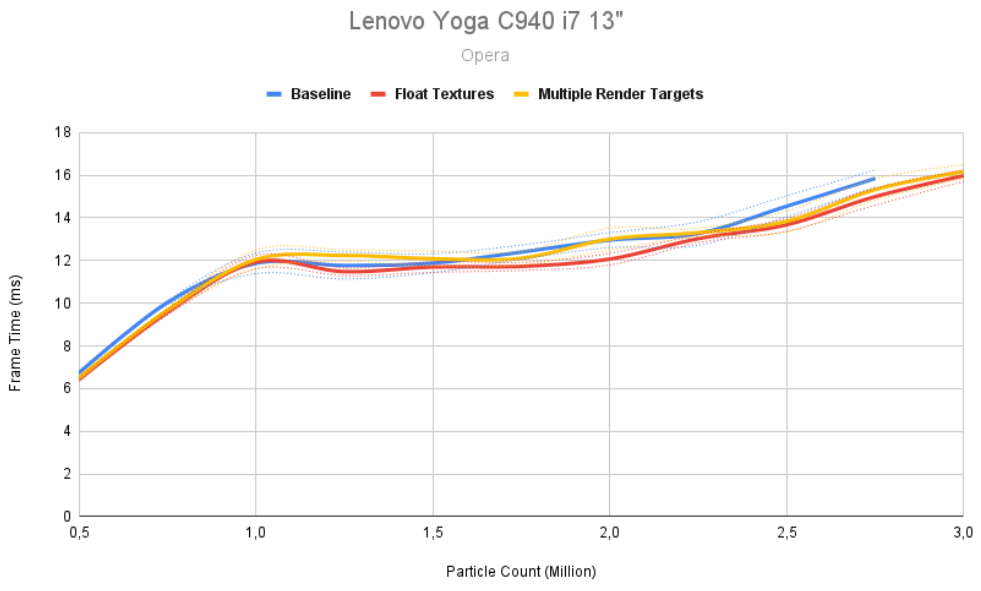

Figure 5.2: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the Lenovo Yoga C940 running Opera.

The testing on the MacBook yielded the most consistent result across all browsers. On average, the multiple render targets solution increased performance by 10% compared to the baseline, and the float solution increased performance by 14%. The performance gain is more significant from 3.5 million particles and above, reaching as high as 26% on Mozilla Firefox (Figure 5.3) using floating point textures.

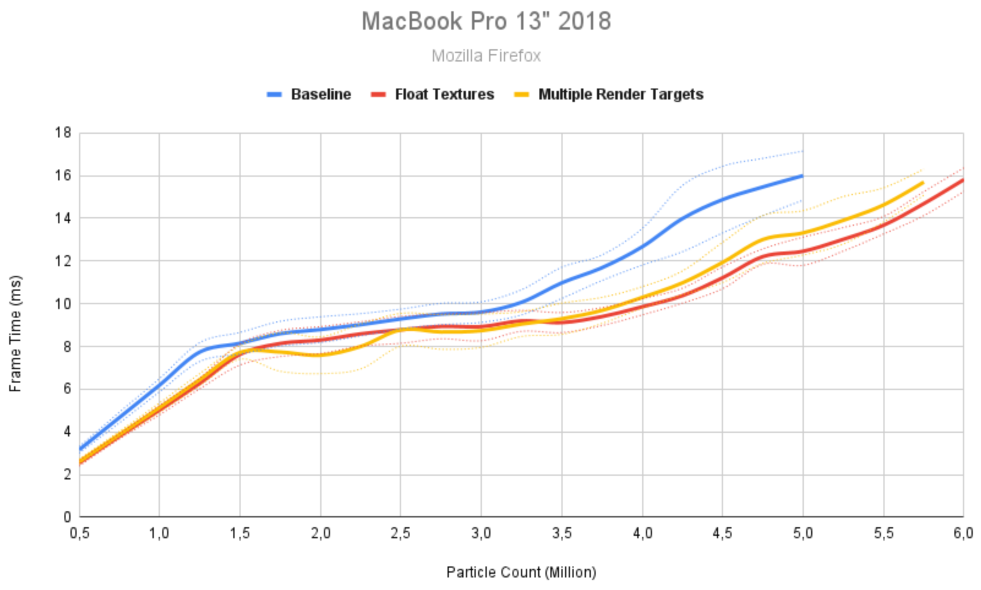

Figure 5.3: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the MacBook Pro running Mozilla Firefox.

As can be seen from the results of the desktop PC (Figure 5.4 and 5.5), having a dedicated, high-end GPU increases the possible particle amount significantly, compared to the other chosen testing platforms. The results also show that the optimization techniques have a slightly smaller impact on the results with a high-end GPU, with multiple rendering targets yielding an average of 5% performance increase on the Chromium based browsers, and a 16% increase on Firefox. Floating point textures increased performance even less than multiple render targets, with an average of 3% increase on Chromium based browsers. On Firefox however, floating point textures are on par with multiple render targets. Neither technique yielded a worse average results than the baseline in any of the test cases. Worth noting is that the floating point textures solution gives a more stable, linear result with lower, on average, standard deviation per data point.

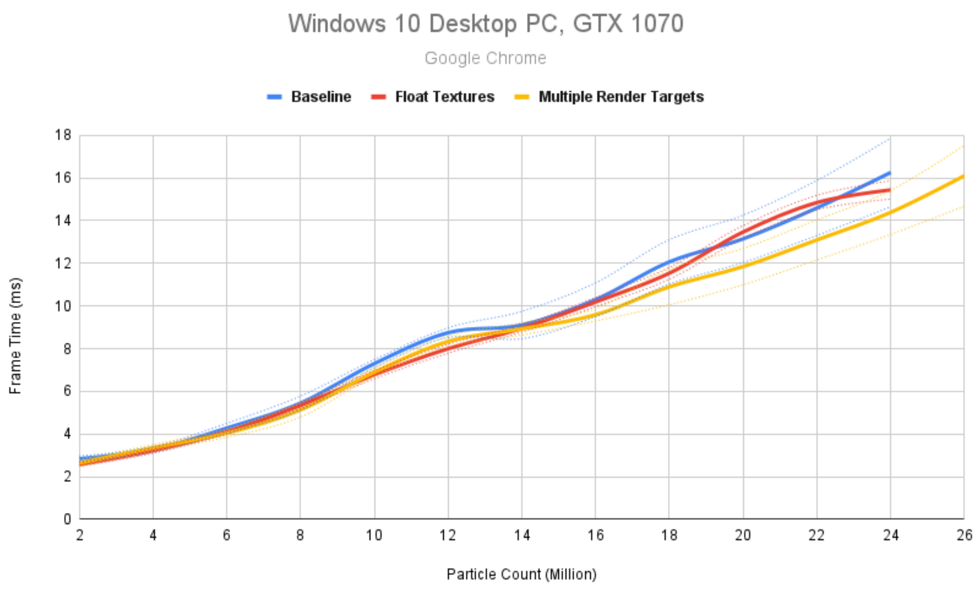

Figure 5.4: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the Desktop Workstation running Google Chrome.

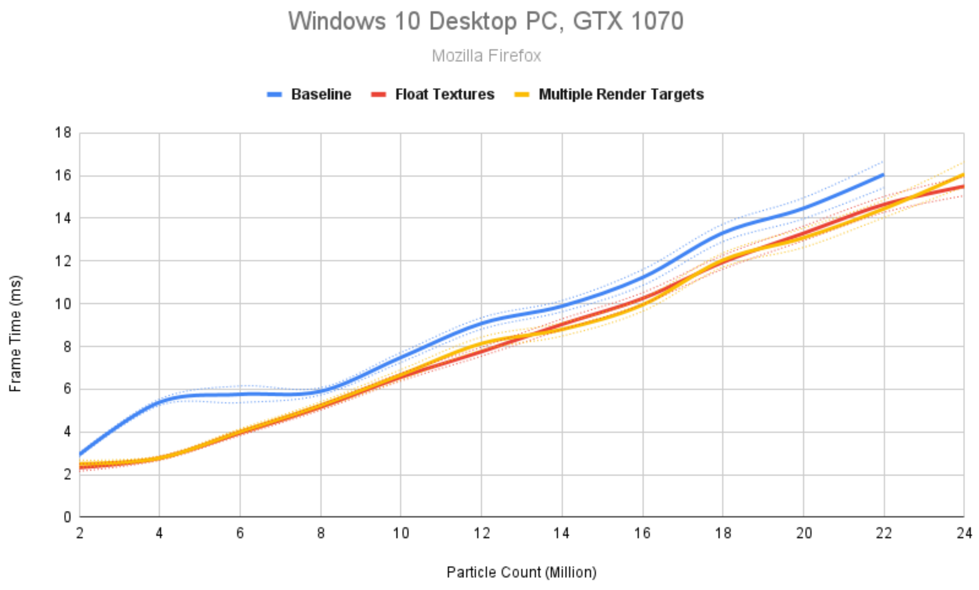

Figure 5.5: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the Desktop Workstation running Mozilla Firefox.

The most important performance increase, given the fact that these optimizations are made primarily for low-end platforms and WebGL 1.0, comes from floating point textures on the OnePlus 3. As can be seen in Figure 5.6, the frame time was 14%

faster on average, with a maximum difference of 24%. This allowed the solution using floating points textures to generate and render around 100 000 particles more than the baseline solution before dropping below 60 frame per second, going from 400 000 particles to 500 000. As the extension that enables multiple render targets is not available on most android devices or browser, this could not be included in the test. The browser used to measure the results on the OnePlus 3 was Mozilla Firefox, and more specifically the Nightly build version, as this has remote debugging enabled. The main reason behind this is that Firefox Nightly has support for the time query extension that was used to measure the time spent on each frame.

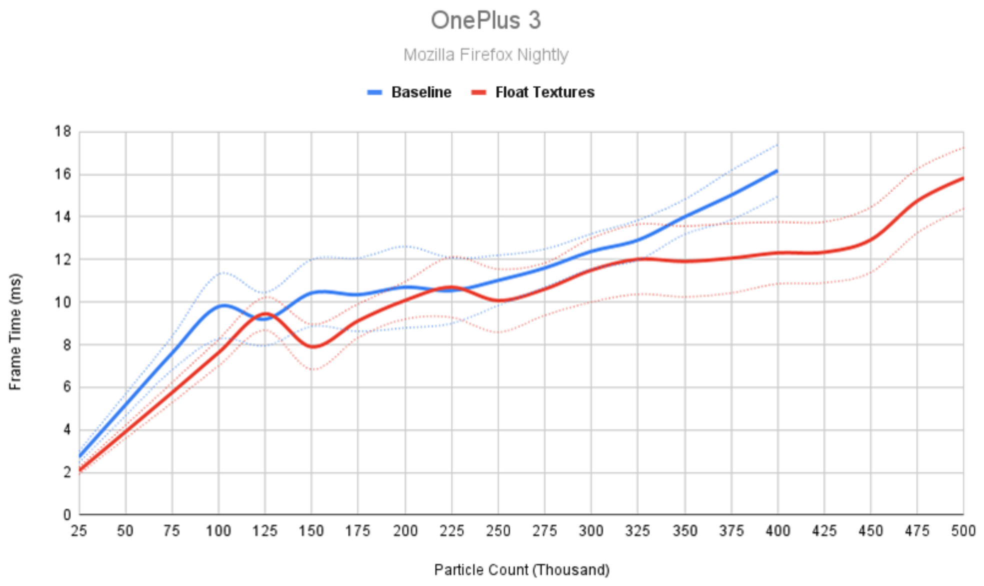

Figure 5.6: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the OnePlus 3 running Mozilla Firefox Nightly.

A more detailed frame time breakdown can be seen in figures 5.7, 5.8, 5.9 and 5.10. As expected, the main contributor of the decreased frame time is the step in which the particles are updated. For floating point textures, the required time to update the particles is 40% of the time required by the baseline solution, due to the decreased amount of texture reads and writes. When utilizing multiple render targets however, the results are different than expected, as the performance is either on par with floating point textures, or yields no notable performance increase. The possible reasons for this is further discussed in section 6.2.

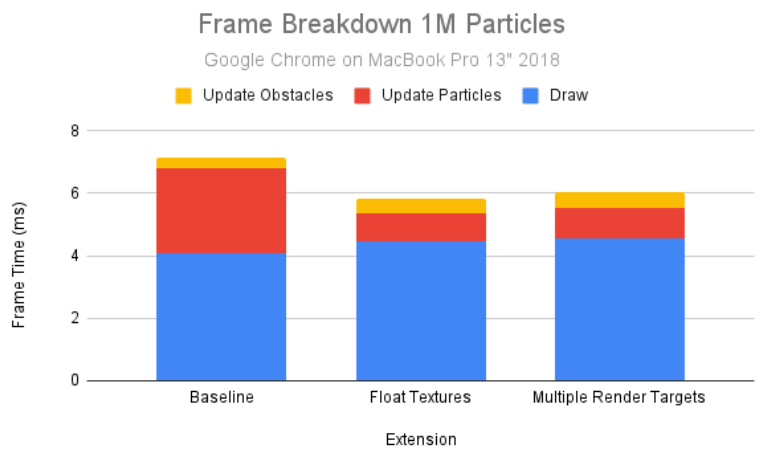

Figure 5.7: Time per frame spent executing GL-commands for one million particles, divided into the three main stages of a frame on the MacBook Pro running Google Chrome.

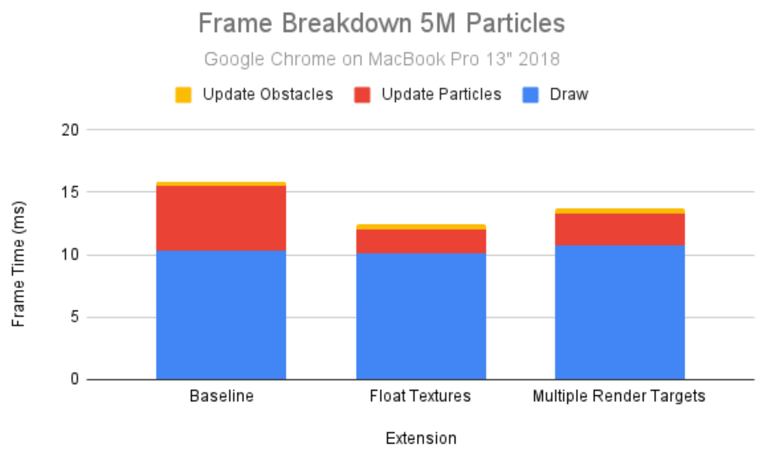

Figure 5.8: Time per frame spent executing GL-commands for five million particles, divided into the three main stages of a frame on the MacBook Pro running Google Chrome.

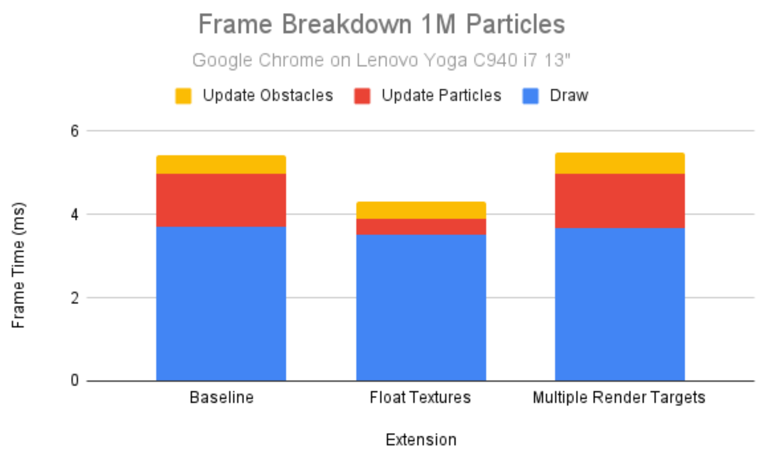

Figure 5.9: Time per frame spent executing GL-commands for one million particles, divided into the three main stages of a frame on the Lenovo Yoga C940 running Google Chrome.

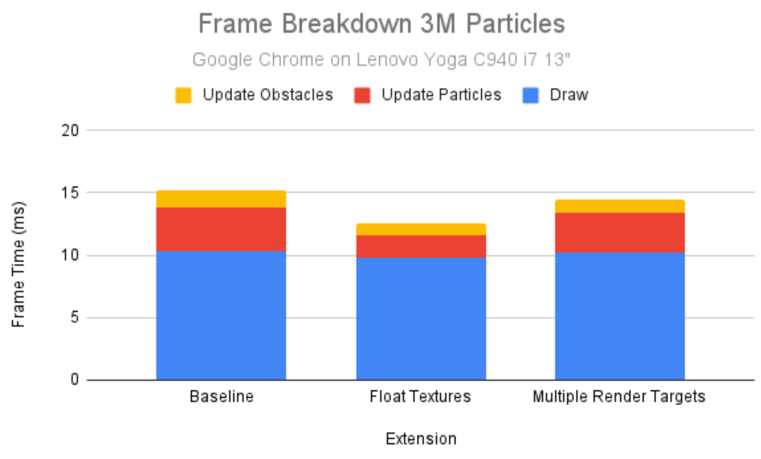

Figure 5.10: Time per frame spent executing GL-commands for three million particles, divided into the three main stages of a frame on the Lenovo Yoga C940 running Google Chrome.

Based on these results, it is clear that the level of optimization differs depending on the platform, and sometimes browser. In most cases however, at least one of the chosen methods does outperform the baseline solution, while in some of the cases, both outperform the baseline solution. This, together with the fact that no results yielded a baseline that performs better than the chosen methods, it is possible to optimize a GPU-accelerated particle system using either floating point textures or multiple render targets. The major downside of using multiple render targets being that it is not supported on most mobile browsers, and as that is one of the main areas in which WebGL 1.0 is prominent, implementing floating point textures could be preferred. A summary of the results can be found in Table 5.1.

	Float Textures	Render Targets
Desktop		
Chrome	3.45%	6.37%
Edge	5.31%	7.53%
Opera	2.03%	1.74%
Firefox	16.76%	16.06%
Lenovo		
Chrome	27.59%	3.43%
Edge	-	-
Opera	3.35%	0.94%
Firefox	-	-
MacBook		
Chrome	13.65%	10.29%
Edge	12.77%	7.92%
Opera	13.34%	9.26%
Firefox	14.63%	14.04%
OnePlus		
Firefox	13.87%	-

Table 5.1: Average performance increase compared to the baseline solution.

All results that are not directly referred to, or those that had to be excluded from the evaluation, can be found in the appendix. This includes all tested browsers that performed similarly enough to other browsers, and the unreliable results from the Lenovo.

5.2 Obstacle Performance

As with measuring the particle performance, different intervals with differing amounts of obstacles had to be used, as the desktop PC outperforms the OnePlus 3 significantly. As the results from testing particle performance on the Lenovo laptop were skewed on both Microsoft Edge and Mozilla Firefox, and also due to the fact that the OnePlus 3 performance could only be measured on one single browser, the tests for obstacle performance were only run on one browser per platform.

The solution using uniforms and multiple draw calls, instead of vertex data, performs significantly worse than the other solutions, which was expected. As the desired result of instancing is to decrease the frame time, while maintaining the memory usage of uniforms, the expected performance increase of instancing was expected to be close to that of the solution using vertex data. This was the case for the desktop PC (Figure 5.11) and the OnePlus 3 (Figure 5.14), as the average difference between the two solutions was close to 2% on these platforms. However, on the MacBook (Figure 5.13) and the Lenovo (Figure 5.12), the performance difference was around 24%. Interestingly, this performance difference was in favor of instancing for the Lenovo, and in favor of vertex data using the MacBook.

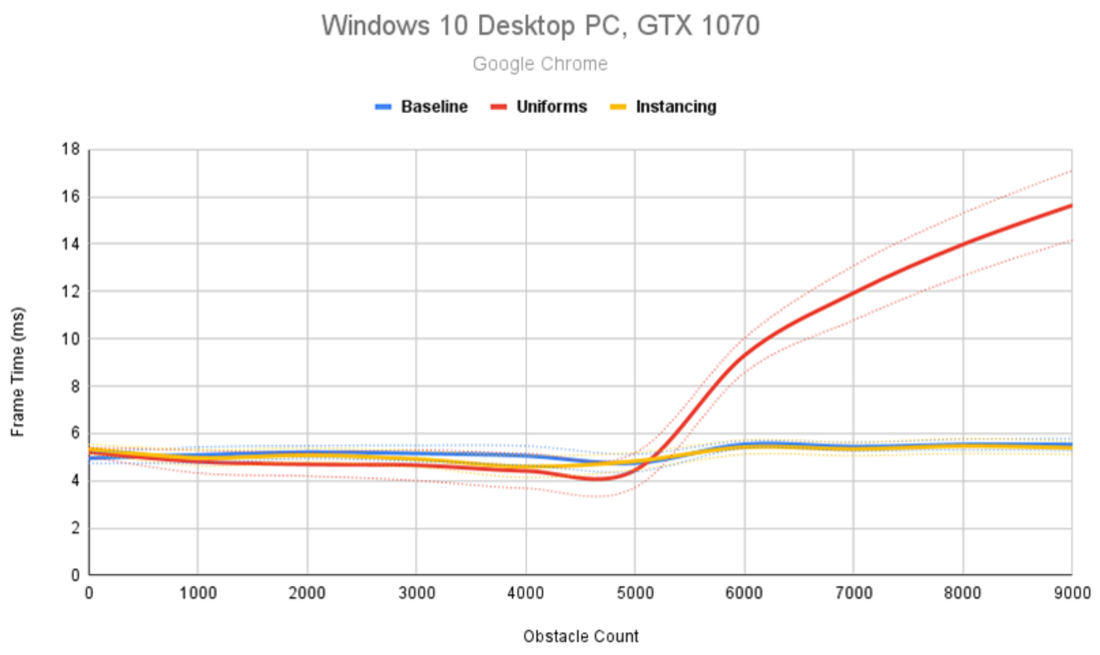

Figure 5.11: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the Desktop Workstation running Google Chrome.

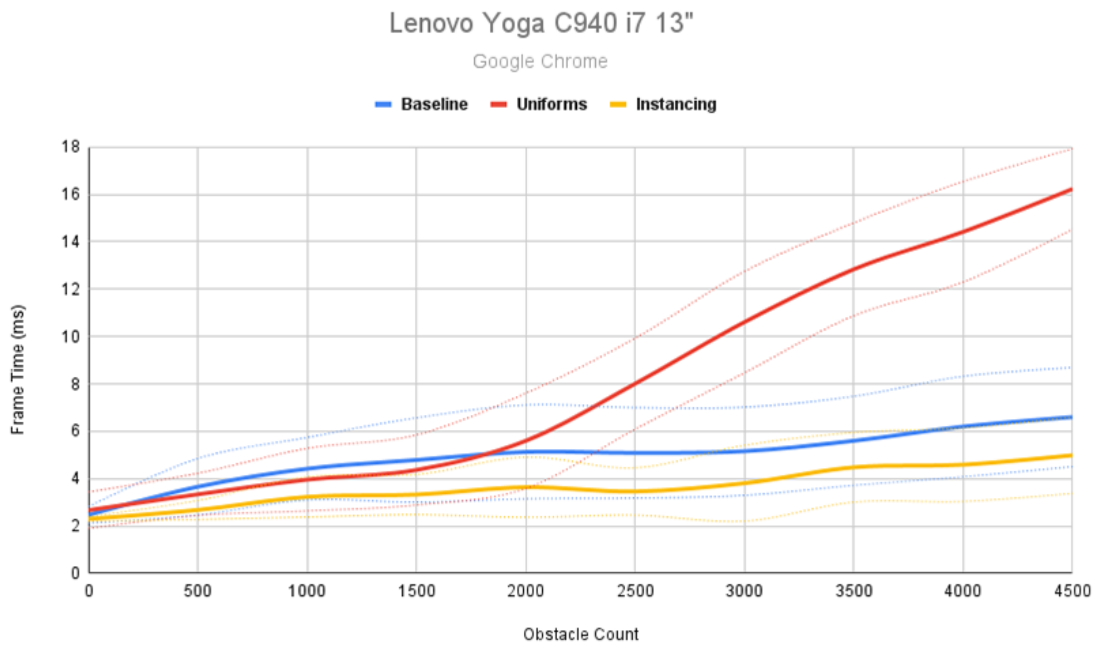

Figure 5.12: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the Lenovo Yoga C940 running Google Chrome.

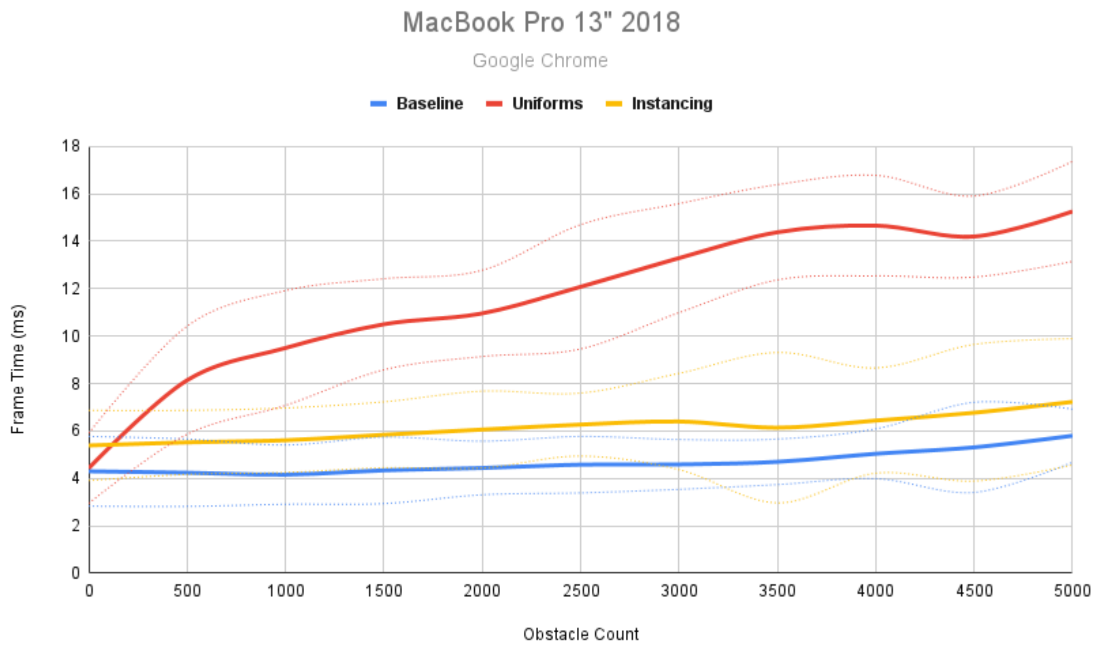

Figure 5.13: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the MacBook Pro running Google Chrome.

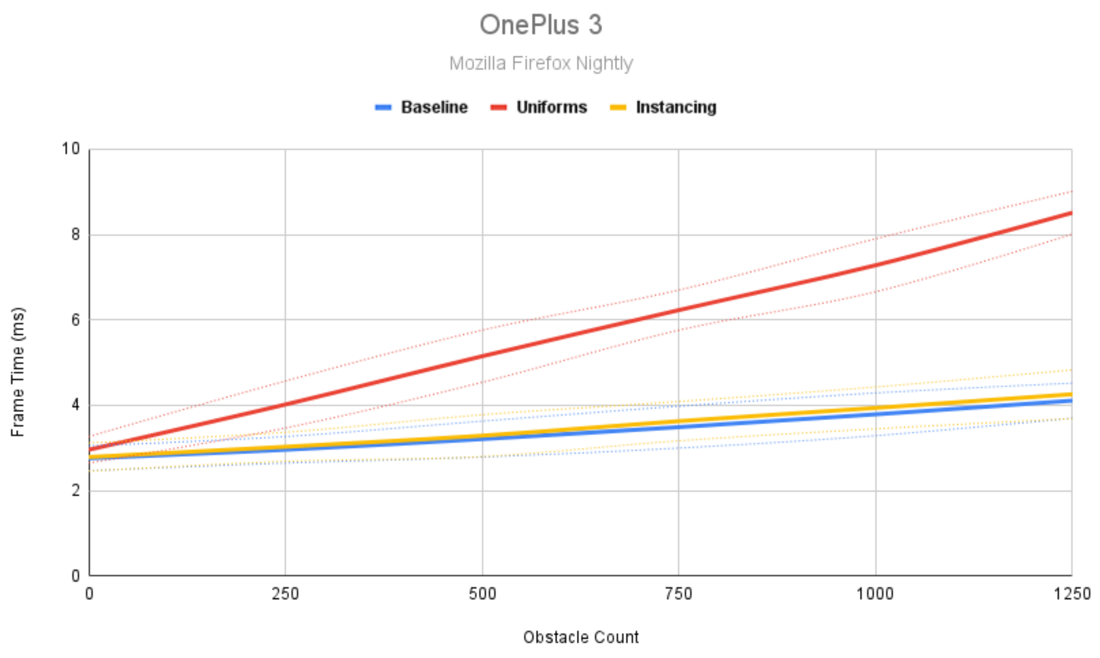

Figure 5.14: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the OnePlus 3 running Mozilla Firefox Nightly.

In contrast to all other test results, the obstacle testing only reaches a frame time of around 9ms on the OnePlus 3, instead of 16ms. The reason for this is that, due to the high amount of obstacles, the loading and initialization of the obstacles took long enough for the program to trigger a loading time-out error.

Based on these results, the performance gained is negligible on modern systems, when used with the type of obstacles in this project. As high as 9 000 obstacles were used to reach a memory usage of 55kB. Though older hardware is more restricted, it is not restricted enough for this amount of memory to make a significant difference. If complex meshes with hundreds of vertices were to be reused and compatibility was not considered, implementing instancing should be considered. In this scenario however, these results still show a trend, and that instancing can in fact outperform the baseline solution in frame time on some browsers and platforms.

6 Discussion

This chapter includes a discussion around the results together with potential future work. Note that this includes speculation regarding some of the results, and not everything that is stated is based on fact. Following the discussion are considerations concerning ethics, and a final conclusion

6.1 Alternative Extensions

One additional extension examined was the EXT_frag_depth [10], which enables rendering to a depth texture through the fragment shader. In a scenario where one additional attribute value is required, this extension could prove beneficial. However, in this projects specific use case, one single storage spot for attribute data was not useful, as the EXT_draw_buffers extension solves the same issue but with more storage possibilities.

Besides using fragment depth, no other extensions examined were deemed suitable for this project. Most extensions are meant to be used as tools for the developer, rather that optimizations techniques.

6.2 Results

Floating point textures, multiple render targets, and instancing, are all viable alternatives for optimizing a GPU-accelerated particle system in WebGL 1.0. As they come in the form of extension, the original idea of them was most likely to improve either performance or make implementation easier for developers, so it might not be entirely unexpected that they do increase performance. What is interesting is rather how well it improves the specific area of particles, as GPU-accelerated particles in WebGL 1.0 require the use of textures for data.

An interesting, unexpected outcome is the differing results between platforms and browsers. Based on the data collected, utilizing a dedicated GPU for a desktop PC seems to slightly decrease the need for these extensions, which points towards a difference in how the techniques are implemented on a hardware level. The performance gap between browsers on the Lenovo however, does not necessarily indicate that the extensions are implemented differently, as the baseline solution also varies greatly between browsers. Rather, this might indicate that they access the GPU differently, and have varying levels of WebGL optimization.

Another interesting observation can be made in all of the MacBook results, the Lenovo Opera results, and the results from the OnePlus 3. This is the fact that the frame time always seems to plateau after a certain amount of particles, only to start rising again after adding even more particles. One possible explanation for this could be due to how much of the GPUs resources are allocated to the browser. At first, not much processing power is required, so the results scale linearly when increasing the particle amount. After a certain threshold is reached, more GPU resources are allocated to the browser, letting the application use the amount needed for each particle count, meaning that frame time is not increased, as the GPU power available increments at the same rate as the particle count. As soon as this reaches a maximum value, or if the GPU throttles, the frame time starts rising linearly again, as particles are increased. Do note that this is entirely speculation, and requires further investigation.

The last thing to note is that on some platforms, floating point textures are on par with the baseline solution while multiple draw buffers increases performance, while on another platform, the opposite is true. As one extension removes the requirement of having multiple textures, and the other removes the need for multiple draw passes, the differing results indicate that some platforms implement texture handling more efficiently, while others might implement the use of shaders and the GPU in WebGL in an effective way. This too requires more research.

6.3 Future Work

Besides the research points noted above, there are many areas of work similar to what this thesis covers, that can be explored. As this project covers the use of WebGL 1.0 and its extensions, these techniques can be implemented in WebGL 2.0, where they are supported natively. The resulting performance increase or decrease can then be compared to an implementation in WebGL 1.0.

A compute shader functionality was recently under development for WebGL 2.0, called WebGL 2.0 Compute [19]. As mentioned in 2.1, compute shaders have been used to optimize particle systems. However, shortly after development started, focus was shifted to developing the specifications for WebGPU [25]. WebGPU is a relatively new framework for GPU programming on the web that is currently being developed. The main goal of WebGPU is to increase the developers control of web based GPU programming and increase performance compared to regular WebGL. Using similar techniques as shown in this report, together with techniques using compute shaders, particle systems could be optimized with WebGPU.

Suggested future work related to this specific thesis is to implement the particle system in 3D, and scale the different implementation to fit the additional requirements. As this thesis did not cover lighting within particle systems, nor inter-particle collision, these areas could be explored as well. Instancing could also be utilized to optimize the particles, rather than the obstacles. In the test suite constructed in this project, obstacles were rendered with a center point and a size, only requiring one entry of vertex data per object rather than 4 or 6 for a standard quad. If the same technique was applied to the particles, they could benefit from all the perks of instancing, and be rendered in different sizes and colors based on one center vertex point and accompanying vertex data.

6.4 Ethical Considerations

As particle effects most often consist of many small, moving particles that can reflect and emit light, stroboscopic effects can occur. According to Newal Salet et al. [31], stroboscopic effects during concerts performed in darker atmospheres more than tripled the risk of epileptic seizures. As this thesis presents methods of improving particle system performance, this has the possibility of contributing to more issues within this area.

Another notable factor is that any type of system that contributes to a better gaming experience can increase the risk of the game being more addictive, as the overall enjoyment of the game can improve. Proving that a particle system is a major factor in this is difficult, but as video game addiction is a very real issue [36], it is good to keep in mind while developing such a system.

6.5 Conclusion

In conclusion, this thesis aimed to examine if a GPU-accelerated, WebGL 1.0 particle system with collision and particle physics, could be optimized through the use of appropriate techniques available with extensions. The thesis also aimed to analyse the relevant extensions, and give reasons for why the chosen ones were appropriate for particle system optimizations within the constraints of WebGL 1.0. The results showed that using extensions that enabled floating point textures, multiple render targets, and instancing, were advantageous for most systems, and that they rarely decrease performance for any tested system. The only case that decreased performance was when instancing was implemented on the Macbook. However, when comparing this result with memory usage and the other platform results, it can still be seen as an improvement, depending on what is prioritized.

Bibliography

[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018.

[2] Pat Brown, Daniel Koch, John Kessenich, and Members of the ARB working group. Arb_compute_shader specification. [Online]. Available: `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_compute_shader.txt`, 2018, Date Accessed: 2021-03-09.

[3] NVIDIA Corporation. About cuda. [Online]. Available: `https://developer.nvidia.com/about-cuda`, 2021, Date Accessed: 2021-03-26.

[4] Wen Jun Ding, Jeremy Zhen Jie Lim, Hue Thi Bich Do, Xiao Xiong, Zackaria Mahfoud, Ching Eng Png, Michel Bosman, Lay Kee Ang, and Lin Wu. Particle simulation of plasmons. *Nanophotonics*, 9(10), 2020.

[5] Palm E. Graphics' card utility with webgl and n-buffering. B.S. Thesis, Computer Engineering, KTH, Stockholm, 2014.

[6] Kim Enarsson. Particle simulation using asynchronous compute : A study of the hardware. Master's thesis, BTH, Department of Computer Science, 2020.

[7] Defold Foundation. Defold. [Online]. Available: `https://defold.com/`, 2021.

[8] Prashant Goswami, Christian Markowicz, and Ali Hassan. Real-time Particle-based Snow Simulation on the GPU. In Hank Childs and Steffen Frey, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2019.

[9] Khronos Group. Webgl angle_instanced_arrays khronos ratified extension specification. [Online]. Available: `https://www.khronos.org/registry/webgl/extensions/ANGLE_instanced_arrays/`, 2015, Date Accessed: 2021-03-06.

[10] Khronos Group. Webgl ext_frag_depth khronos ratified extension specification. [Online]. Available: `https://www.khronos.org/registry/webgl/extensions/EXT_frag_depth/`, 2015, Date Accessed: 2021-06-15.

[11] Khronos Group. Webgl webgl_draw_buffers khronos ratified extension specification. [Online]. Available: `https://www.khronos.org/registry/webgl/extensions/WEBGL_draw_buffers/`, 2016, Date Accessed: 2021-03-08.

[12] Khronos Group. Webgl oes_texture_float khronos ratified extension specification. [Online]. Available: https://www.khronos.org/registry/webgl/extensions/OES_texture_float/, 2017, Date Accessed: 2021-03-07.

[13] Khronos Group. Webgl oes_texture_half_float khronos ratified extension specification. [Online]. Available: https://www.khronos.org/registry/webgl/extensions/OES_texture_half_float/, 2017, Date Accessed: 2021-03-07.

[14] Khronos Group. Webgl specification. [Online]. Available: https://www.khronos.org/registry/webgl/specs/latest/1.0/, 2020, Date Accessed: 2021-03-10.

[15] Khronos Group. Webgl specification. [Online]. Available: https://www.khronos.org/registry/webgl/specs/latest/2.0/, 2020, Date Accessed: 2021-03-10.

[16] Khronos Group. Opengl the industry's foundation for high performance graphics. [Online]. Available: https://www.opengl.org/, 2021, , Date Accessed: 2021-07-15.

[17] Khronos Group. Webgl ext_disjoint_timer_query extension specification. [Online]. Available: https://www.khronos.org/registry/webgl/extensions/EXT_disjoint_timer_query/, 2021, Date Accessed: 2021-05-01.

[18] Khronos Group. Core language (glsl). [Online]. Available: https://www.khronos.org/opengl/wiki/Core_Language_(GLSL), 2021, Date Accessed: 2021-06-12.

[19] Khronos WebGL Working Group. Webgl 2.0 compute specification (wip, obsolete). [Online]. Available: https://www.khronos.org/registry/webgl/specs/latest/2.0-compute/, 2021, Date Accessed: 2021-05-16.

[20] Kyle Hegeman, Nathan A. Carr, and Gavin S. P. Miller. Particle-based fluid simulation on the gpu. 2006.

[21] Pyarelal Knowles. Gpgpu based particle system simulation. Master's thesis, RMIT University, Department of Computer Science and Information Technology, 2009.

[22] Andreas Kolb and Nicolas Cuntz. Dynamic particle coupling for gpu-based fluid simulation. 2005.

[23] C. Kubisch. New rendering techniques for real-time graphics: Turing - mesh shaders. SIGGRAPH, 2018.

[24] Jon Leech and Benj Lipchak. Opengl es version 3.0.6. [Online]. Available: https://www.khronos.org/registry/OpenGL/specs/es/3.0/es_spec_3.0.pdf, 2019, Date Accessed: 2021-03-10.

[25] Dzmitry Malyshau and Kai Ninomiya. Webgpu w3c working draft (wip). [Online]. Available: https://www.w3.org/TR/2021/WD-webgpu-20210526/, 2021, Date Accessed: 2021-05-16.

[26] Microsoft. Directx graphics and gaming. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/directx, 2021, , Date Accessed: 2021-07-15.

[27] Microsoft. High-level shader language (hlsl). [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl, 2021, Date Accessed: 2021-06-12.

[28] Mozilla and individual contributors. Webgl: 2d and 3d graphics for the web. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API, 2021, , Date Accessed: 2021-07-15.

[29] Hubert Nguyen. *GPU Gems 2 - Chapter 3. Inside Geometry Instancing.* Addison-Wesley Professional, New York, NY, USA, 2005.

[30] Henrik Wann Jensen Ronald Fedkiw, Jos Stam. Visual simulation of smoke. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, 2001.

[31] Newel Salet, Marieke Visser, Cornelis Stam, and Yvo M Smulders. Stroboscopic light effects during electronic dance music festivals and photosensitive epilepsy: a cohort study and case report. 2019.

[32] Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. A material point method for snow simulation. 2013.

[33] Unity Technologies. Webgl browser compatibility. [Online]. Available: https://docs.unity3d.com/Manual/webgl-browsercompatibility.html, 2019, Date Accessed: 2021-03-17.

[34] Unity Technologies. Unity documentation - webgl graphics. [Online]. Available: https://docs.unity3d.com/Manual/webgl-graphics.html, 2021, Date Accessed: 2021-02-27.

[35] Tom A. J. Welling, Sina Sadighikia, Kanako Watanabe, Albert Grau-Carbonell, Maarten Bransen, Daisuke Nagao, Alfons van Blaaderen, and Marijn A. van Huis. Observation of undamped 3d brownian motion of nanoparticles using liquid-cell scanning transmission electron microscopy. *Particle & Particle Systems Characterization*, 37(6), 2020.

[36] Kimberly Young. Understanding online gaming addiction and treatment issues for adolescents, 2009.

A

Appendix 1

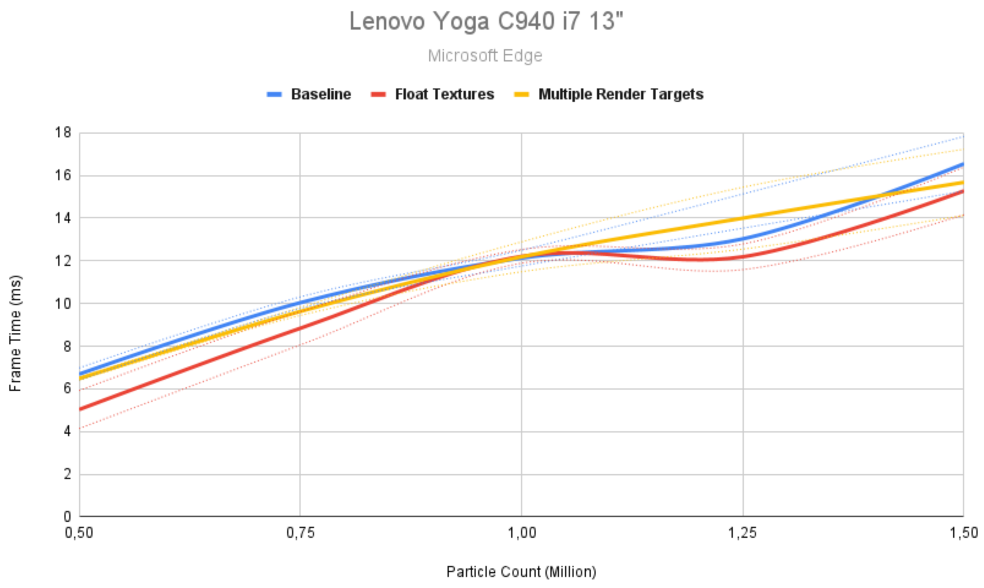

Figure A.1: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the Lenovo Yoga C940 running Microsoft Edge.

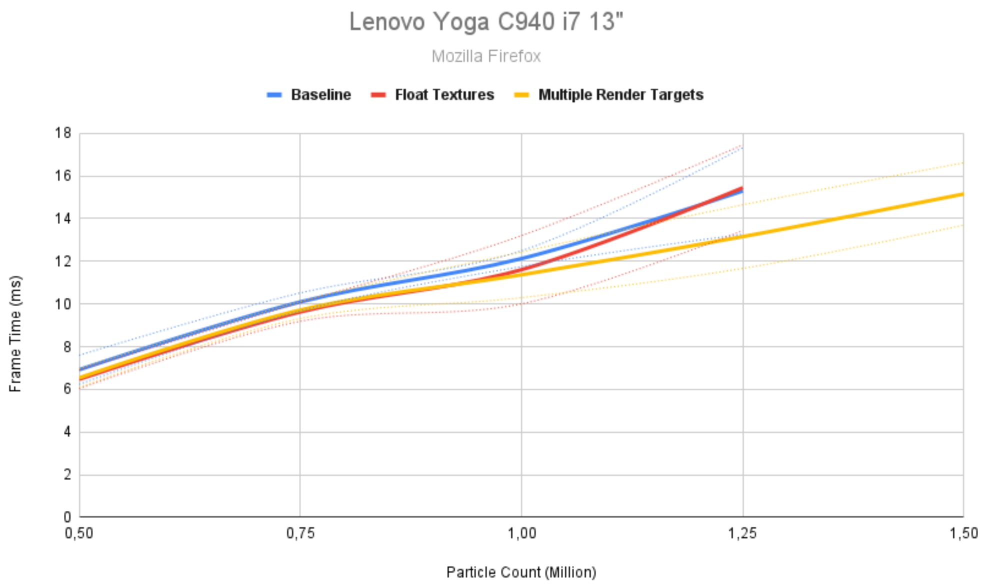

Figure A.2: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the Lenovo Yoga C940 running Mozilla Firefox.

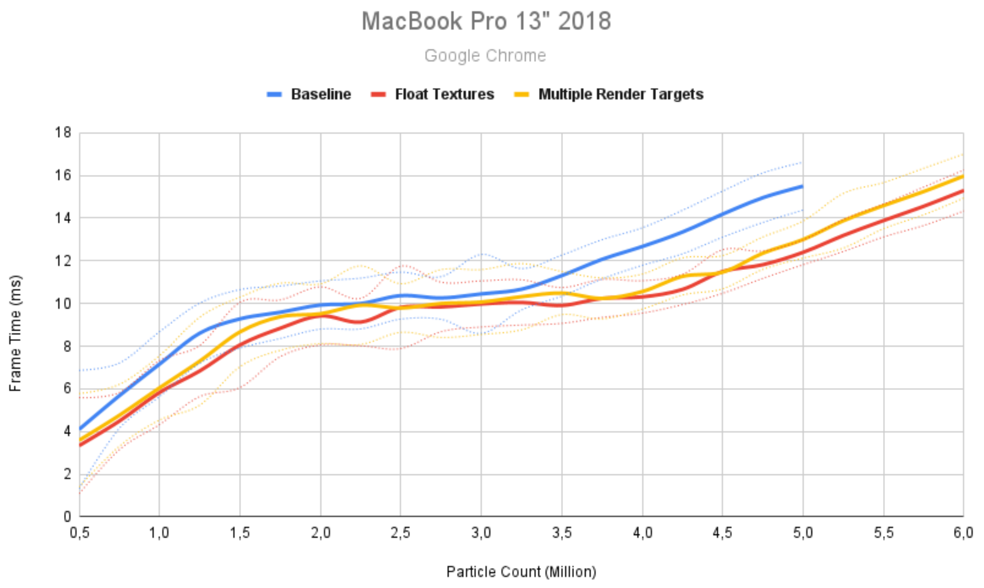

Figure A.3: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the MacBook Pro running Google Chrome.

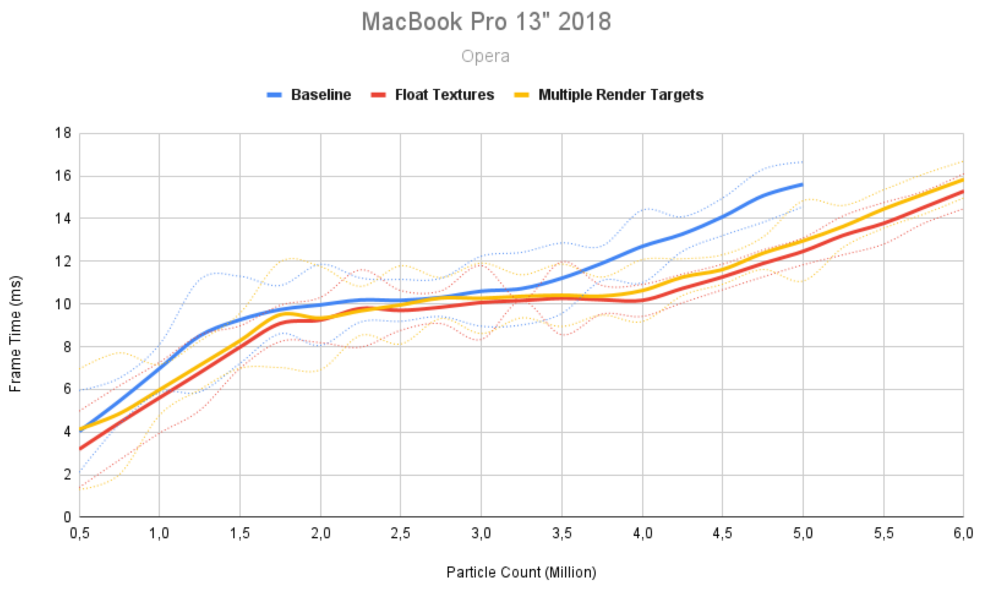

Figure A.4: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the MacBook Pro running Opera.

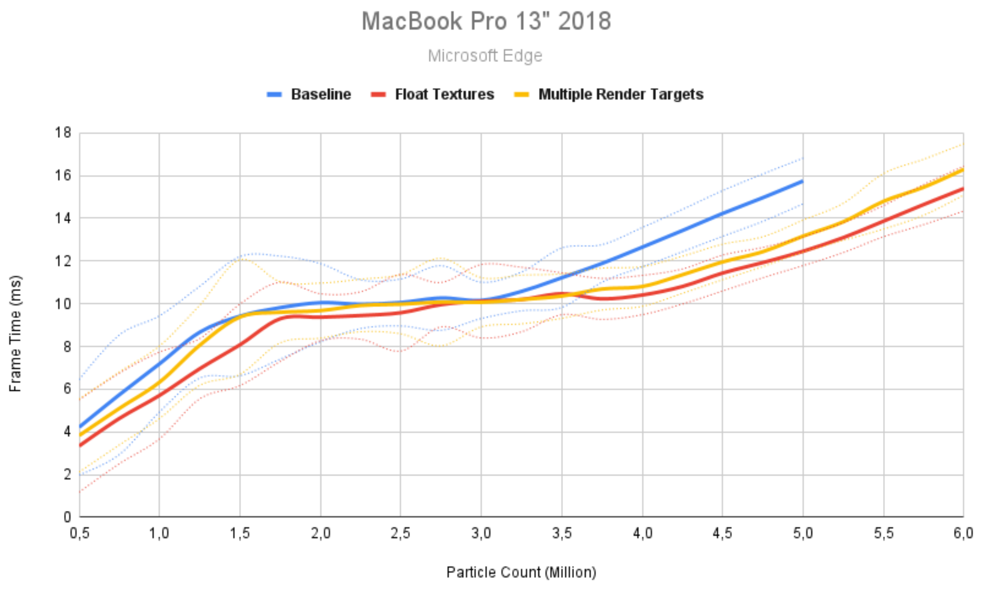

Figure A.5: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the MacBook Pro running Microsoft Edge.

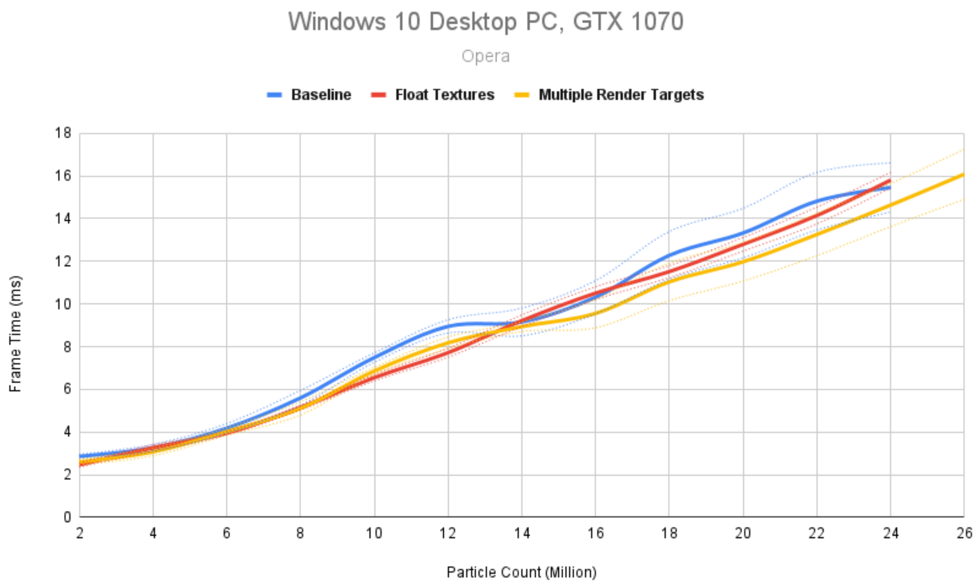

Figure A.6: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the Desktop Workstation running Opera.

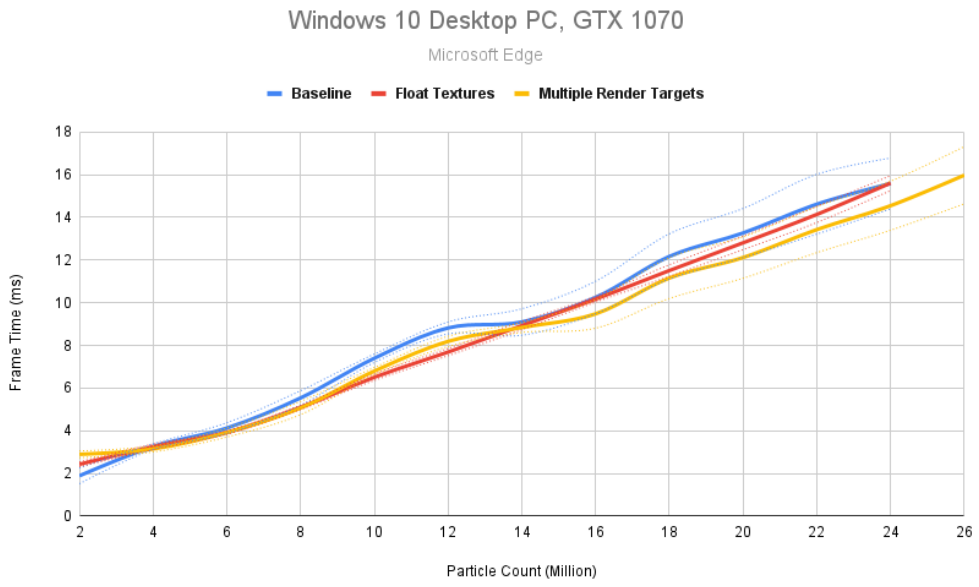

Figure A.7: Time per frame spent executing GL-commands including updating obstacles, particles and rendering on the Desktop Workstation running Microsoft Edge.