Master of Science in Engineering: Game and Software Engineering
June 2023

Performance comparison of WebGPU and WebGL in the Godot game engine

Emil Fransson
Jonatan Hermansson

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Engineering: Game and Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Emil Fransson
E-mail: emfa17@student.bth.se

Jonatan Hermansson
E-mail: johm18@student.bth.se

University advisor:
Senior Lecturer, Yan Hu
Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. For rendering graphics on the web, WebGL has been the standard API to employ over the years. A new technology, WebGPU, has been set to release in 2023 and utilizes many of the novel rendering approaches and features common for the native modern graphics APIs, such as Vulkan. Currently, very limited research exists regarding WebGPU:s rasterization capabilities. In particular, no research exists pertaining to its capabilities when used as a rendering backend in game engines.

Objectives. This paper aims to investigate performance differences between WebGL and WebGPU. This is done in the context of the game engine Godot, and the measured performance is that of the CPU and GPU frame time. The tests consist of six games for analyzing real-world cases and a number of synthetic test cases that target specific parts of the rendering pipeline. To perform the comparisons a WebGPU backend Rasterizer was implemented with the intended scope of being able to render basic 2D games.

Method. The existing WebGL Rasterizer in Godot was deconstructed to match the scope of the intended rendering functionality. The WebGPU Rasterizer was then implemented in its image and the performance of the implementations was measured in different scopes. These scopes include the frame time on the GPU and CPU and some essential rendering functions on the CPU side. Lastly, the means were calculated, and a t-test was performed to validate the significance of the difference between Rasterizers.

Results. The results show that WebGPU performs better than WebGL when used as a rendering backend in Godot, for both the games tests and the synthetic tests. The comparisons clearly show that WebGPU performs faster in mean CPU and GPU frame time. This held true also for 95% lowest frame time. The results varied for the cases of the mean 1% high frame time, with WebGPU generally performing better. The results for the essential rendering functions saw WebGL performing consistently better.

Conclusions. In conclusion, WebGPU outperformed WebGL. In most of the tests conducted, substantially and with high statistical significance. In order to better realize the performance benefits of WebGPU in the environment of game engines the implementation could be further expanded on in order to support more advanced games and 3D scenes. Still, the findings of this thesis show WebGPU as a strong contender to WebGL for web rendering.

Keywords: Game Engine, Performance Overhead, Rendering, WebGPU, WebGL

Sammanfattning

Bakgrund. För att rendera grafik på webben har WebGL varit det vanliga API:et att använda under åren. En ny teknik, WebGPU, är planerad att släppas 2023 och använder många av de nya renderingstekniker- och funktioner som är vanliga för moderna grafik-API:er, som Vulkan. För närvarande finns mycket begränsad forskning om WebGPU:s rasteriseringsförmåga. I synnerhet finns ingen forskning gällande dess användning som renderingsbackend i spelmotorer.

Syfte. Detta arbete syftar till att undersöka prestandaskillnader mellan WebGL och WebGPU. Det görs i sammanhanget av spelmotorn Godot, och den uppmätta prestandan är bildtid på CPU:n respektive GPU:n. Testerna består av sex spel för att analysera mer verkliga sammanhang samt ett antal syntetiska testfall som riktar sig mot specifika delar av renderingspipelinen. För att utföra jämförelserna implementerades en WebGPU rasteriserare med den begränsade förmågan att kunna rendera grundläggande 2D-spel.

Metod. Den befintliga WebGL-rasteriseraren i Godot demonterades för att matcha omfattningen av den avsedda funktionaliteten. WebGPU-rasteriseraren implementerades sedan i dess avbild och prestandan för implementationerna mättes i de olika testen. Mätningarna inkluderar bildtiden på GPU och CPU samt några viktiga renderingsfunktioner på CPU-sidan. Slutligen beräknades medelvärden och ett t-test för att validera signifikansen av skillnaden mellan rasteriserarna.

Resultat. Resultaten visar att WebGPU presterar bättre än WebGL när den används som renderingsbackend i Godot, både för speltesterna och de syntetiska testerna. Jämförelserna visar tydligt att WebGPU presterar genomsnittligt bättre. Detta gällde även för de 95% lägsta bildtiderna. Resultaten varierade mer för fallen med den genomsnittliga 1% höga bildtiden. Resultaten för de specifika renderingsfunktionerna visade dock att WebGL konsekvent presterade bättre.

Slutsatser. Sammanfattningsvis överträffade WebGPU WebGL. I de flesta genomförda tester avsevärt och med hög statistisk signifikans. För att bättre inse prestandafördelarna med WebGPU i spelmotormiljö kan implementeringen utökas ytterligare för att stödja mer avancerade spel och 3D-scener. Ändå visar resultaten i denna avhandling att WebGPU är ett bra alternativ till WebGL för webbrendering.

Nyckelord: Spelmotor, Prestandakostnader, Rendering, WebGPU, WebGL

Acknowledgments

We would like to thank our supervisor Yan for providing us with valuable knowledge and insights in writing this thesis. Furthermore, we would like to thank our friends and co-advisors Fritjof and Mikael at Macaroni Studios for providing us with this opportunity and a place to conduct our work. Without the help of these people, this thesis would not have been possible.

Emil
I would personally like to thank my friend and thesis partner Jonatan for the excellent teamwork and his ability to always remain positive and motivated in the effort of completing this thesis work on time. It has been a pleasure.

Jonatan
I would like to extend my deepest gratitude to my friend and thesis partner Emil, who has not only been paramount in his contribution to the thesis but also in keeping my own motivation high over the course of the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern video games leverage sophisticated graphics application programming interfaces (APIs) to render highly detailed worlds. They accomplish this at interactive frame rates by utilizing powerful graphics processing units (GPUs) with which modern computers are equipped. Commonly used APIs include Direct3D [18] for machines running Windows, Metal [4] for Apple products, and Vulkan [25] and OpenGL [15] as a cross-platform alternative.

The APIs mentioned all target native platforms and, as evident, there are many choices available to developers on these. However, when it comes to rendering on the web, the choices narrow significantly. Previously, WebGL was the lowest-level alternative for rendering on the web [9]. It is based on the aforementioned OpenGL native API and adopts the same workflow and syntax.

The most modern of the mentioned APIs work closer to the hardware they are targeting than ever before, allowing developers a higher degree of control and opportunity for low-level optimization previously unachievable. However, this additional control places much more responsibility on the developer. E.g. regarding resource management and synchronization. This is in contrast to how the GPU drivers handle it in the background with older APIs and API versions.

This thesis consists of an implementation of a rendering backend for the game engine Godot using the currently latest low-level web graphics API WebGPU, see section 1.1.2, and comparing its performance in various test cases to the performance of the WebGL backend currently implemented in Godot.

1.1 Background

This section presents vital information pertaining the background of some of the major concepts of this thesis.

1.1.1 WebGL

WebGL is a cross-platform, open-source API for rendering interactive 2D and 3D graphics on the web, with an initial release in March 2011. A typical WebGL program consists of JavaScript-written control code and shader code facilitated by the OpenGL Shading Language (GLSL). Additionally, Emscripten may compile C/C++ OpenGL code into WebAssembly, allowing the WebGL API to be interacted with through lower-level languages (see section 1.1.4) [16].

WebGL is a mature API supported by many different hardware products and browsers and has been applied in many environments and fields, such as rendering backends in the gaming industry and for visualization purposes in medicine and in geospatial applications.

As it is built as a subset of the OpenGL API, its strengths consists of a general ease of use (compared to alternative graphics API:s such as Vulkan), cross-platform support and being open-source. However, it also inherits the drawbacks prevalent in OpenGL, such as a notorious driver overhead and a stateful syntax with many necessary render pipeline constructs having to be set repeatedly, further increasing the overhead. Furthermore, WebGL has no support for general compute [16].

WebGL currently exists as a possible rendering backend in the Godot engine for rendering graphics on the web platform.

1.1.2 WebGPU

WebGPU is a new graphics API that aims to bring a more modern API workflow to web platforms with its first draft of specifications being released in 2021 [27]. Like the previously mentioned modern APIs, it aims to enable the developer to work closer to the hardware of the machine it is running on. The API utilized by the web browser is determined by the operating system on which it is executed. Depending on the specifications of the system, the web browser may utilize either the Direct3D 12, Vulkan, or Metal APIs.

As with these APIs, WebGPU provides developers with relatively direct access to previously inaccessible low-level GPU resources. It also employs a stateless syntax which leads to fewer API calls, invoking less API overhead when compared to the stateful syntax of WebGL, inherited by OpenGL.

WebGPU is not meant to be a replacement for the WebGL API. Rather, it has been referred to as its successor [14]. It also provides functionality previously unavailable to WebGL through compute shaders that enable hardware-accelerated general computation in the browser. However, general computation is not a part of the comparison in the thesis as WebGL does not formally support compute shaders.

One of the notable syntactic similarities that can be found between WebGPU and APIs like Direct3D 12 and Vulkan is the idea of bundling state. This lies at the core of these APIs and is best exemplified through the pipeline state object. This state structure holds a complete configuration of the graphics pipeline, excluding bound resources. It is then used to bind the entire pipeline as a bundle through a single API call. In less modern APIs, different parts of the pipeline state are set through separate API calls, including different shaders, depth-test states, and so on. This, invariably, leads to a larger API call overhead on the central processing unit (CPU) side of the application and is one of the main bottlenecks modern APIs like WebGPU try to avoid.

While WebGPU is closer in syntax to that of the modern native graphics APIs, it still has to adhere to the fundamental security and compatibility standards required of web APIs. This is inherently detrimental to the performance of WebGPU when compared to native APIs directly. However, WebGL also follows these requirements, making a comparison between the two apt.

1.1.3 Godot

Godot is an open-source game engine first released in 2015. It has since had many updates and the newest version, 4.0, was recently released as of writing this thesis [11], with many new features and an entirely new rendering pipeline leveraging the aforementioned Vulkan API, along with a host of updates to the existing legacy rendering backends.

Godot is multifaceted in the advantages it affords the work when used as a foundation for implementing a rendering backend. Firstly, a pre-established architecture can be followed during implementation, keeping comparisons between rendering APIs fair. Secondly, the currently implemented WebGL rendering backend can be assumed to be fairly well optimized and thus serves as a good benchmark for the performance of WebGL rendering engines in the industry.

The reason for choosing Godot over another game engine mainly comes down to it being open-source, meaning its code is openly available and able to be modified. This makes open-source software ideal for an in-engine implementation. Other big game engines, such as Unity and Unreal Engine, are either closed-source or only provide access to the source code under a specific license.

1.1.4 Emscripten and WebAssembly

Godot is primarily written in the programming language C++. To run it on the web, it needs to first be compiled into WebAssembly [20], a low-level bytecode language meant to narrow the gap between web and native performance.

One caveat of WebAssembly is that it does not have direct support for calling web APIs, including WebGPU. To circumvent this issue, JavaScript must be employed to invoke these APIs. The compiler Emscripten [8], which Godot uses to compile its C++ code to WebAssembly, provides headers that allow the C++ code to interact with certain web APIs, among them WebGL and WebGPU.

The communication between C++ and JavaScript introduces a necessary overhead. However, this overhead exists for both APIs and cannot be avoided without writing the entire application in JavaScript. The possible difference in overhead for the two APIs caused by the intercommunication with JavaScript can be viewed as a part of the operative cost of that API.

1.2 Aim and Objectives

The aim of the thesis is to examine and quantify the performance difference between WebGPU and WebGL when employed as a rendering backend for the game engine Godot, with the purpose of determining how the API:s perform relative to one another in such an environment. This is accomplished through a careful reconstruction of a scaled-down subset of the already implemented backend in the engine using the WebGPU API. Then, measurements of their performance are taken in various test scenes.

The test scenes include scenes measuring performance in very specific rendering scenarios as well as scenes more generally representative of real, though simple,

games. More specific information regarding the various test cases is discussed in chapter 3.

With this as the aim of the thesis, the **objectives** of the thesis are as follows:

1. Investigate the WebGPU API and best practices.
2. Investigate the Godot 4.0 rendering backend.
3. Deconstruct the existing WebGL render backend to align with the scope of the Minimum viable product (MVP).
4. Implement a WebGPU render backend in the Godot game engine, adhering to the Godot 4.0 rendering backend as closely as possible.
5. Set up a test environment in the Godot game engine, suitable for the implemented WebGPU MVP solution.
6. Profile CPU & GPU frame time, meaning how long each frame takes to render, of the implemented rendering backend and the existing WebGL backend.

1.3 Thesis Scope

The scope of the thesis amounts to implementing and profiling a WebGPU renderer based on the Rasterizer backend architecture existing in Godot 4.0. This is done under the hypothesis that such a Rasterizer should be able to outperform the existing WebGL equivalent due to its modern architecture. The capabilities of the implemented Rasterizer should be limited to rendering textured sprites and panels allowing for simple, yet complete, 2D games with accompanying user interfaces. It should also look and play identically to the WebGL-rendered equivalent from start to finish.

Due to time constraints, features such as 3D-rendering and advanced (post-processing) effects are omitted from the scope of this work. While they are natively supported in Godot Engine (and by the WebGL Rasterizer), such features will be deconstructed for the WebGL backend. This is done to ensure that both Rasterizers act on identical data in an identical environment and are fairly profiled.

Another reason for opting to target simple 2D games and scenes is that a vast majority of games developed for Godot are made in 2D. As the render backend will be based on the updated Rasterizer architecture in Godot 4.0, it is assumed that the existing collection of games made with it will be 2D and simple in scope and complexity. This makes for a good fit for the intended solution and ensures that supported games exist which are possible to render with the features included with the implemented Rasterizer.

1.4 Glossary

This thesis features frequently used terminology pertaining to Godot, WebGPU, rendering, and more. The reader should refer to Table 1.1 where these terms are explained in alphabetical order.

Table 1.1: A glossary of frequently used terminology

Term	**Explanation**
Bind Group	"*A bind group is a WebGPU construct that represents a bundled set of resources that can be bound as a set through a single graphics command.*"
Blit	"*Blit is a lexicalized form of BitBlt, Bit Block Transfer, that refers to moving a block of bits from one place in graphics memory to another. In more modern computer graphics the term is generalized to simply refer to any copying of data from one place in graphics memory to another, such as a texture being sampled by a point sampler.*"
Command Encoder	"*The command encoder is a WebGPU construct that allows the recording of graphics commands. The command encoder is later submitted to the graphics queue to have its commands executed on the hardware.*"
Polygon	"*A Polygon is a Godot-defined render item type that is involved with rendering panels, buttons, and other user interface objects. The type is essential for rendering most 2D games.*"
Rasterizer	"*A Rasterizer (with capital R) is Godot's naming convention for render backends inheriting from the Renderer Compositor class and implementing the necessary functionality pertaining the rendering of scenes. Both the WebGL render backend and the implemented WebGPU render backend are Godot Rasterizers. The new Vulkan backend, in contrast, is not as it implements the newer RenderDevice framework instead.*"
Rect	"*A Rect is a Godot-defined render item type that is involved with rendering (textured) rectangular shaped sprites. The type is essential for rendering 2D scenes.*"
Render Item	"*A Render Item is a Godot-defined type that acts as a base class from which all different supported render item types (such as rects and polygons) must inherit. It contains data and logic essential for rendering.*"
*Storage	"*A Storage is a Godot-defined type acting as asset manager for some particular asset type, optimizing the workflow and memory usage in dealing with those assets. A notable example is TextureStorage.*"
WGSL	"*WGSL stands for WebGPU Shader Language and is, as it states, the shader language used by WebGPU shaders.*"

1.5 Ethical, Societal and Sustainability aspects

The implementation and data-gathering phases of the work are conducted purely digitally and no other people will be involved in any part of the entire process.

The utilized games and game-related assets are all freely available to use under allowing licenses.

Furthermore, there is no physical product that requires any kind of materials to produce. In terms of sustainability, the only aspect that could be argued for is that intensive graphics applications use a lot of energy. However, with the assumption that the application will be run no matter the backing rendering engine, we believe the difference in energy use between the rendering engine implemented in this work and the currently implemented WebGL rendering engine in Godot to be negligible.

1.6 Contribution

The results of this thesis can provide valuable insights for developers and other stakeholders of graphics and game engines regarding the performance possibilities in using the latest and most modern rendering API for web rendering, especially in the context of the Godot code environment.

Furthermore, the results can provide grounds for a choice between WebGPU and WebGL, where the former has a more complex API leading to the possibility of longer development times.

Lastly, the work is not exhaustive and can instead be seen as a foundation and the first steps toward further improvement. It can be built upon and used as a basis for comparisons in future research on the same or similar subjects.

1.7 Outline

The thesis structure begins with chapter one and the introduction, detailing background information pertaining to the work, aim and objectives, thesis scope, ethical, societal, and sustainability aspects, and the academic contribution the work is expected to provide. This chapter is followed by a chapter on related work, in which the reader is updated on key research done in the areas of WebGPU and general compute, Vulkan vs. OpenGL comparisons, and more. The chapter also goes into detail on how this work aims to lessen the research gap on the subject. Chapter three details the research question, technical limitations, and the overall research method, in which the full WebGPU Rasterizer implementation details are explained. It also includes how the experiment and data gathering (profiling) was conducted and the validity and reliability of the chosen approach. Chapter four presents the Results and Analysis of the conducted experiment. Chapter five contains a Discussion of the performed work, and the sixth and final chapter presents the conclusions drawn from the work and proposes several optimizations and adjustments in the future work section.

Chapter 2

Related Work

2.1 WebGPU & Compute

With the advent of WebGPU, general computation on the web is now possible to do on the GPU, through the use of configurable compute shaders. It is not possible to set up compute shaders using WebGL. However, a number of user-defined hacks that emulate compute shader capabilities have been suggested over the years.

With general computing capabilities, new possibilities are made available to the audience; for example, the implementation of neural networks in the web browser. Hidaka et al. found that their implementation of a deep neural network (DNN) using WebGPU performed around 36 times faster (91 ms over 3297 ms) compared to another popular DNN implementation for the web that makes use of the emulated compute capabilities of WebGL [13].

Aldahir researched the compute performance differences (Mandelbrot set generation and matrix multiplication) of CUDA and WebGPU, with WebGPU set up to run compute operations in a cluster of web browsers. The results showed CUDA being faster and more efficient than WebGPU. However, the authors added that WebGPU is still in early development and hence not as stable and mature as CUDA. Also, WebGPU, along with WebRTC, displayed good scalability with over 75% efficiency for building clusters of web browsers [2].

Usher and Pascucci compared the compute capabilities of WebGPU with that of native Vulkan and found them to be remarkably similar in terms of performance on compute-heavy tasks. In the paper, the marching cubes algorithm applied on a scalar field was used as a proxy for compute-intensive tasks. The results display similar performance with WebGPU falling in the same order of magnitude and often even closer to the Vulkan implementation in terms of time-to-render [24].

Dyken et al. investigated the relative performance of rendering large-scale graph layouts on the web using libraries based on WebGPU (GraphWaGu), WebGL (NetV & Stardust), and non-GPU-accelerated equivalents (such as D3 Canvas). Thanks to the compute capabilities of WebGPU, GraphWaGu is the only GPU-leveraged library that is able to compute iterations of the graph algorithms in parallel. At 100.000 nodes and 2.000.000 edges, only GraphWaGu is able to maintain interactive rendering at a frame rate of ten or more. The equivalent frame rate for NetV is three, with StarDust being unable to render the graph layout at all. Pushing GraphWaGu to its limits, a maximum of 200.000 nodes and 4.000.000 edges are rendered, a feat that no other tested library, WebGL-based or otherwise, is able to accomplish [7].

All of the papers mentioned in this section have compared WebGPU to another

established API in terms of its compute performance. It details both how it compares to WebGL, wherein general compute is not readily available, as well as APIs that are more suited to general compute or even designed for it in the case of CUDA. This gives a general idea as to how WebGPU compares to other APIs. However, the aim of this thesis is to find how it performs regarding its rasterization capabilities which is present in WebGL, unlike general compute.

2.2 Vulkan vs OpenGL

Previous research has been done comparing declarative APIs such as OpenGL to the more modern APIs such as Vulkan and determined that Vulkan does indeed perform much faster than OpenGL in metrics such as draw call overhead [21]. In one such study, this was shown to be especially true for low polygon-count meshes but also for higher polygon-count examples, though the difference between the two was smaller in this case. The results show that the more modern API makes better use of the GPU and therefore performs better.

Further research in the same vein was done comparing Vulkan and OpenGL in various micro-benchmarks [10]. The tests found, among other discoveries, that Vulkan was able to push a much larger amount of triangles per second than OpenGL at the cost of a slightly higher power requirement.

In a study measuring the energy efficiency of Vulkan as compared to that of OpenGL, performance was also noted [17]. Performance was measured in frames-per-second and samples were taken at various workloads. The results show that Vulkan boasts far better performance and predictability, especially at higher workloads.

The research papers mentioned, however, used native APIs. This thesis will instead perform a similar comparison between two web APIs, WebGL and WebGPU. This could cause results to differ from the native case. However, the presented research does give an assortment of examples wherein a modern API outperforms a declarative API which hints at what could be expected from a comparison between WebGL and WebGPU as well.

2.3 Reducing the Research Gap

It is evident from the findings presented in this chapter that there has been quite some research done in the field of WebGPU and its general compute capabilities. However, this does not hold true for WebGPU and its rasterization capability counterpart, in particular research involving comparisons of WebGPU and WebGL. Furthermore, at the time of writing this thesis, no research could be found that places its context inside the environment of a game engine. The work presented in this thesis aims to do just that, effectively reducing the research gap on WebGPU as a new rasterization technology for the web in the environment of Godot, grounding the research and results in real usability scenarios.

Chapter 3

Method

This chapter presents and motivates the methods conducted in order to reach the results. The applied methodology belongs to the domain of *Research And Development* and is investigated in three main parts:

1. Detailed explanation of the software implemented for the solution, pertaining to the development of the WebGPU Rasterizer backend.
2. The approach used to quantify the performance differences between the implemented WebGPU rasterizer and the existing WebGL Rasterizer.
3. As assessment of the validity and reliability of the chosen approach.

3.1 Research Question

As of writing this thesis, very limited research can be found overall on the subject. In particular, research relating to rasterization performance is absent, with most findings being the work of hobbyists. As this thesis further narrows the scope by aiming to quantify and compare the performance of WebGPU relative to WebGL in a game engine environment the research question becomes as follows:

- *RQ:* What is the difference in performance regarding CPU and GPU frame time between WebGPU and WebGL when used as a rendering backend in the Godot game engine?

By studying the research in chapter 2 and applying knowledge regarding how WebGPU aims to provide a workflow more optimized for the hardware it is targeting, the expectation is that the implemented WebGPU backend should perform better than the WebGL backend. It is, however, difficult to predict how large the difference between the two is. Since no previous research has been done targeting rendering with WebGPU, no hypothesis regarding said difference has been made.

3.2 Technical Limitations

A number of technical limitations were identified during the research phase and have played a part in steering the solution in certain directions. These limitations mainly derive their existence from limitations in the WebGPU API and/or Godot code base and would be too time-consuming or outright impossible to overcome in the time

allocated for this thesis work. Furthermore, the limitations had to be respected in order for the performance profiling to be fair and consistent. The limitations are outlined below.

3.2.1 Selecting a Backend Framework

With the release of Godot 4.0, a new rendering backend was introduced, called the Render Device. It was implemented in order to reflect the modern approach of organizing a renderer. New renderer backends are suggested and urged to inherit from and implement this new framework for the best performance and pipeline workflow. The current official Vulkan renderer does this, as does the (as of writing this thesis) unreleased D3D12 renderer. During the early research and development phase, using the render device backend seemed the most reasonable for this implementation as well. However, a host of critical problems were identified that made WebGPU a bad fit for this modern workflow, of which the most glaring are detailed here:

- In its current state, WebGPU does not support recording render commands from multiple threads. Render Device backends are by default scheduled on different threads, which would invalidate the WebGPU primitives, such as the device and adapter.

- Shaders compiled for use by the Render Device backend end up in SPIR-V format, which would have to be converted to an equivalent wgsl format in order to be used by WebGPU. Capable tools exist for this. However, upon testing, it was determined that WebGPU lacked the necessary shader types in order to be compatible.

- WebGPU does not currently support push constants, which are used frequently with the render device backend. In order to solve this and the aforementioned issue, it would have been necessary to act on and adjust the SPIR-V shader byte code to adhere to the WebGPU standard and form compatible wgsl shaders.

- WebGPU does not currently support synchronously waiting for GPU work to complete. As such, all work pertaining to e.g. fetching texture or buffer data would have to be performed asynchronously, invalidating the workflow of the Render Device backend.

While it in theory could be possible to circumvent these issues, upon discussion with Godot developers, it was further determined that implementing WebGPU as a Render Device backend would diminish the value of a performance comparison between WebGPU and WebGL. This is because WebGL implements the legacy Rasterizer backend, and the render paths in Godot are too different between the two backend types for a comparison between them to yield any data of significance. Combining the fact that implementing a WebGPU backend based on the Rasterizer backend type would make for a fair comparison, and that the glsl-shaders in use by WebGL are directly translatable to equivalent wgsl-shaders, it was determined that the Render Device backend should be foregone in favor of the Rasterizer backend.

3.2.2 Frame buffering

The existing WebGL implementation in Godot is set up with multiple frame buffering in mind (defaulting to three frames), so-called "frames-in-flight", as the API allows for such explicit developer control. The WebGPU API, upon analysis, has no explicit synchronization management between CPU and GPU that allows for manual control over frames in flight. This was determined by reaching out to the Google developers responsible for implementing WebGPU on Chrome Canary. They informed that the Dawn API is responsible for managing the WebGPU render commands on Chrome Canary and handles the implicit resource management for developers when writing buffer and texture data. This is in order to minimize CPU-GPU sync points and keep frames in flight going. With this information in mind, the proposed solution was created with the concept of frame buffering treated as a black box where trust is put in the developers behind WebGPU and the conversations had with them. Simply put, explicit control of frame buffering is not currently supported and is as such omitted from this work.

3.2.3 Managing Synchronous GPU Read Backs

When Godot constructs tile maps to be used in games, it queries a Texture Storage class that manages textures for a certain texture to be used for the tile map. However, it does so after all textures for use by a game have already been loaded into the engine and video memory (VRAM), after which they are removed from regular RAM. Godot expects the texture data to be instantly available to it via a readback of the texture data from VRAM. WebGL manages this by introducing a synchronization point and simply waiting synchronously for the texture data to be available before returning it to the caller. However, as was explained in section 3.2.1, such synchronization cannot be set up using WebGPU. In order to circumvent this issue, every texture has its data stored contiguously in an array in system memory until the run time loop begins, at which point it is certain that the engine is done with such tile map management. The data is then erased and freed. This introduces no extra memory overhead during runtime. However, it is worth noting that some intermediate memory overhead exists at initialization.

Furthermore, this means that during runtime, the WebGPU implementation is technically lacking the ability of the engine to read texture data from VRAM. However, in the use cases presented in this thesis, such a readback is never performed. Consequently, the WebGL and WebGPU implementations should be indistinguishable in this regard during application runtime.

3.2.4 Performance Measuring

For security and privacy reasons, browsers limit the precision of high-resolution timestamps. In Google Chrome, this limit is $5\mu s$, which matches the recommended highest precision a browser should provide [26]. Furthermore, a slight randomness is added to these timestamps. Due to these limitations of performance measurement, short functions that run for upwards of tens of microseconds become impossible to measure in a way that provides meaningful data for the study. Thus, these functions are not

measured in the thesis.

3.3 Scope of Implementation

This section explains the rationale behind the scope that is being aimed for in the implementation. Generally, it comes down to selecting which features to include and which to omit from the so-called "Renderer Compositor", from which all current and new Rasterizers must inherit and implement.

Broadly speaking, what to support should be determined by selecting within five categories: The renderer(-s) to implement, the storage(-s) to implement, which types of render items the renderer should support, what shader variants to support, and what, if any, utilities to include. The reader should refer to Figure 3.1 for a visualization.

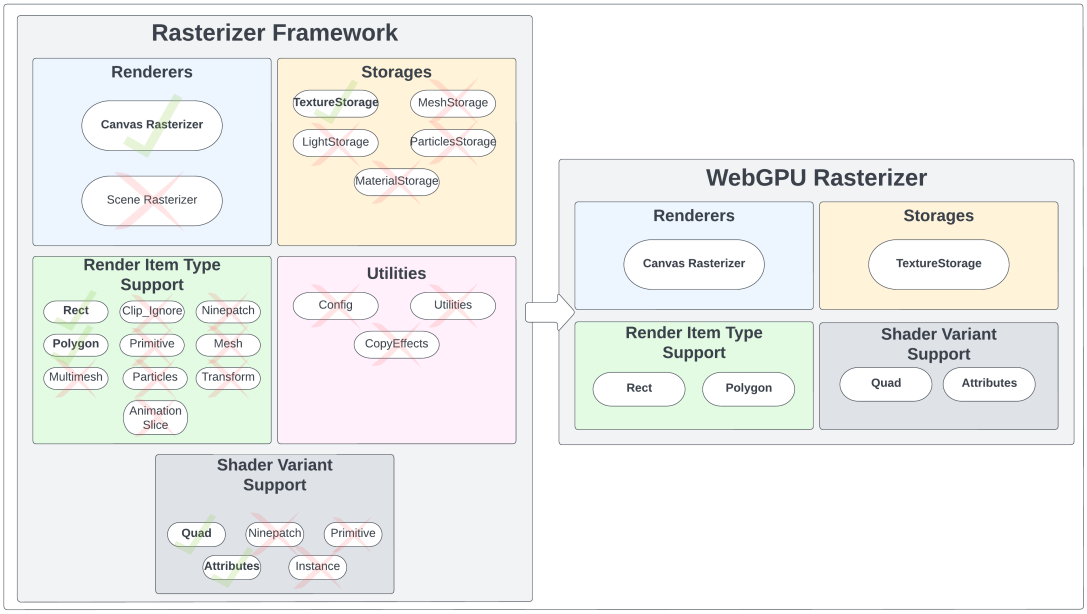

Figure 3.1: Shows a side-by-side comparison of Rasterizer frameworks, with the left-most framework displaying the full set of *possible* implementation features, and the right-most framework displaying the *actual* implemented features for the WebGPU Rasterizer, based on the aimed-for scope.

3.3.1 Supported Renderer and Storage

As was hinted at in section 1.3 - "Thesis Scope", and further discussed in section 3.2.1 - "Selecting a backend Framework", the developed renderer implements the Rasterizer backend, just like the WebGL renderer backend does. However, while this implementation deals with simple 2D rendering only, the Rasterizer backend allows for many more features, such as 3D rendering, advanced post-processing, and

more. The Rasterizer backend is split into a number of different objects that manage these features; a "Canvas Rasterizer" that deals with all things pertaining to 2D rendering, a "Scene Rasterizer" that manages all things related to 3D rendering, and a number of storage classes that act as asset managers for different Godot-defined resource types, some notable examples being textures (TextureStorage), materials (MaterialStorage) and meshes (MeshStorage).

Of these, the Canvas Renderer was implemented, along with the texture storage class, as these in conjunction are sufficient (and required!) to render 2D games. The Scene Renderer only deals with 3D graphics, which is outside the scope of this thesis work. Of the storages, MeshStorage and MaterialStorage are used for 3D rendering, and so are naturally omitted. As rendering lights and particles are outside the thesis scope, so are the equivalent storage classes.

3.3.2 Supported Render Item Types

While Godot defines and supports a number of render item types (ten to be precise, of which five correspond to 2D render items) only two have been selected for being supported by this implementation - the "rect" item type and the "polygon" item type. These two have been selected due to their particular importance for rendering 2D scenes. The rect type makes it possible to render sprites to the screen, and the polygon type makes it possible to render panels, buttons, and other UI elements on the screen. Together, these two types are sufficient for rendering simple 2D games. Although not all render item types in Godot are used for 2D rendering, supporting more would mean improving the general visuals and features of some 2D games (and being able to support more existing 2d games overall). However, due to the limited time for implementation, additional render item type support was omitted.

3.3.3 Supported Shader Types

The WebGL backend Rasterizer has support for a number of glsl-shaders, of which only the "Canvas Shader" is used for 2D rendering. At compile time, this big shader is further split into numerous minor shaders, in Godot referred to as "shader variants". In total five shader variants are compiled for 2D rendering, each corresponding to and managing the final color computations for a single render item type. As this implementation supports two render item types it will also limit its scope to supporting the equivalent two shader variants used for rendering those two types, namely the "quad" variant, used for rendering rect render items, and the "attributes" variant, used for rendering polygon render items.

Furthermore, as blitting is a mandatory render technique used in Godot, this implementation also sets up a shader for achieving that. This is further detailed in section 3.4.4. In total, this means that three shaders are implemented and used in this implementation, which is sufficient for achieving the desired results.

3.3.4 Supported Utility Features

The Utility features available to implement for a Rasterizer (Config, Utilities, and CopyEffects) are mainly concerned with making a Rasterizer generalizable for all

systems and platforms, in the sense that after setting them up they can be referred for system and API-relevant data. Examples include limits, such as the texture dimension limits for a given system, or the maximum length of vertex buffer layout attributes. Another example is retrieving the video adapter API version. While these features are important for a Rasterizer built to be deployed and released, for this thesis work it falls outside of the scope, as the test system is well known. Therefore, no time was spent on including these features.

3.4 Implementation

As noted in section 1.2 - *Aim and Objectives*, the implementation phase consisted of a deconstruction effort of the already existing WebGL Rasterizer, followed by an implementation of an equivalent WebGPU Rasterizer, staying as close as possible to the architectural design decisions of the WebGL Rasterizer. This was followed up by setting up a test environment for the profiling of scenes and games suitable for the scope of the MVP WebGPU Rasterizer, and then the actual profiling.

3.4.1 Deconstructing Godot

In order for the implemented WebGPU Rasterizer and the existing WebGL Rasterizer to be eligible for performance comparisons, the overall computation work they do must be as identical as possible. More precisely, these prerequisites must be aimed for:

1. The shaders used must be as close as possible in terms of instruction count, branching, and operations. Exactly the same work must be done in the shaders.

2. The shader pressure, in terms of data types and data layout, must be as close as possible. In other words, the shaders must act on the same (amount of) data, and the data must be sent to VRAM structured identically.

3. No optimizations are allowed for the WebGPU Rasterizer on CPU-side or GPU-side, which would put it at an unfair advantage over the WebGL Rasterizer. As an example, if the existing WebGL Rasterizer batches instance data and uploads to the GPU even though that data has undergone no change on the CPU, then the WebGPU Rasterizer must honor this Godot design decision and do the same, without adding any optimizations.

4. The CPU workflow must be as identical as possible in terms of computations and branching.

5. The run time allocations should be as identical as possible.

For achieving the aforementioned prerequisites the work began with deconstructing the WebGL Rasterizer to a state where it would match the MVP aimed for as close as possible; the Rasterizer should be able to render simple 2D games of predetermined complexity and nothing more.

First, the unneeded glsl-shader variants were decoupled fully. The remaining two shaders dealing with rects and polygon rendering were then stripped of all functionality outside of the scope of the MVP, which largely amounted to code dealing with light calculations and particle calculations. The equivalent WebGPU wgsl-shaders were then carefully crafted in the image of these shaders, in order to be as close as possible in terms of data and computations. As the shader details are of considerable importance for the implementation they are included in their full display in this thesis. The reader may refer to figures A.1 - A.8 in the appendix section for the complete implementation details for these. A careful and experienced reader can verify that the wgsl and glsl-shaders are, in fact, translations of each other. This fulfills prerequisite one and also sets the shaders up for fulfilling prerequisite two.

Following the deconstruction of the shaders the same process was applied to the WebGL Rasterizer. Logic and data relating to 3D rendering were decoupled first. This meant fully decoupling the WebGL Scene Rasterizer and the storages only dealing with 3D rendering. Following this all logic dealing with unsupported Render Item types was removed, leaving only the rect and polygon types. By this time the WebGL shader variants dealing with types other than rects and polygons were decoupled as well. As the only assets that fall into the MVP are textures the remaining storage classes, aside from the TextureStorage, were now removed as well. Finally, the Utility objects were removed. Looking at Figure 3.1, the leftmost section showing the discarded and included Rasterizer features act as a good overview of the results of the deconstruction; the WebGPU Rasterizer was based on the deconstructed WebGL equivalent.

With the WebGL Rasterizer deconstructed to a minimal state for supporting the MVP, the WebGPU Rasterizer was now crafted in its image. As all Godot Rasterizers must implement the so-called *Renderer Compositor* and override and implement the same functions this means the overall render path will be the same for them all by default, as the engine will communicate with the Rasterizers in the same manner. For achieving a result where the internal details of the render paths are as close as possible for WebGPU and WebGL two main things had to be considered: the details of the render loop and the applied render techniques. For details regarding the implementation of the render loop, the reader is referred to section 3.4.3 - *The Render Loop*, and for the implemented render techniques the reader is referred to section 3.5.4 - *Render Techniques*.

While the implemented WebGPU Rasterizer should be fair in order for the produced profiling results to be of any significance, due to natural differences between the API:s of WebGL and WebGPU the Rasterizer should still be set up in such a way as to play to the strengths of WebGPU. This mainly pertains to the separation between runtime constructs and initialization constructs, in which there are considerate differences between the possibilities of the APIs. For details regarding how these constructs were set up for WebGPU, the reader is referred to the following section -3.4.2 *Minimizing Runtime WebGPU Structures*.

3.4.2 Minimizing Runtime WebGPU Structures

A lot of information is available to the renderer at the time of its initialization. Using this information there has been an attempt to minimize the amount of work that

needs to be performed during the application runtime. Put another way, the goal is to maximize the amount of work that can be done beforehand such that there is no impact on runtime performance after initialization.

Because of how WebGPU allows the binding of bundled state, a lot of GPU data structures can be built during initialization. This includes the pipeline state, shader modules, bind group layouts, and fixed-size data buffers. See Figure 3.2 for an overview of what structures are built during initialization and what structures are built during runtime.

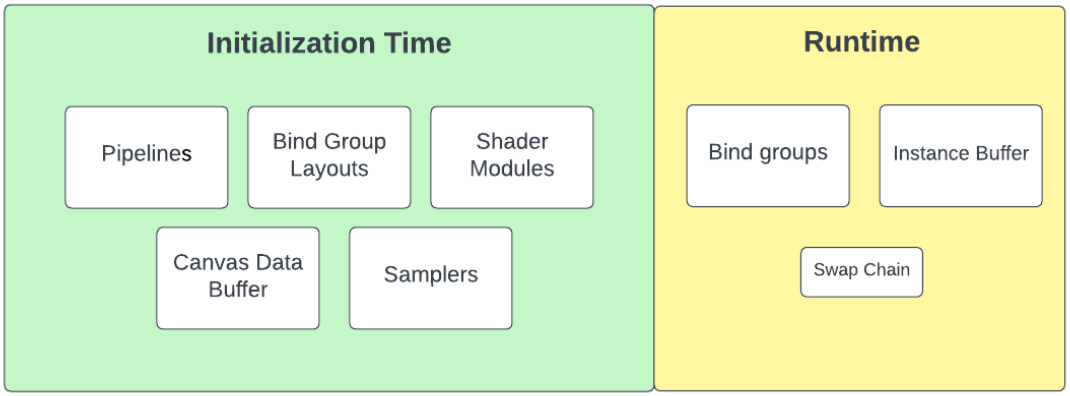

Figure 3.2: Overview of GPU structures that are built during initialization to the left and during runtime to the right. Smaller boxes indicate that the data structure does not have to be rebuilt continuously.

In WebGPU, the pipeline state comprises the state pertaining to the hardware render pipeline. It mainly contains five large modules of data which themselves consist of smaller submodules, see Figure 3.3. These five are the pipeline layout, the vertex state, the fragment state, the primitive state, and the depth stencil state. In the two-dimensional rendering case, the depth stencil state is unused.

The pipeline layout chiefly defines a bind group layout which is used by shaders to access bound GPU resources. The bind group layout consists of bind group layout entries that describe the kind of resource that the bind group can have bound to it. This is clearly known at initialization time, considering the shaders are fixed and, consequently, so are their bound resource slots.

The fragment and vertex states are tightly coupled with the pipeline layout in that they contain compiled shader modules. The shaders need to have bind groups that match those of the bind group layout previously established. The fragment state further contains information relating to the render targets to which it can output which themselves contain a blend state such that alpha blending can be properly supported.

The vertex state contains a description of the vertex attributes passed to the shader through a bound vertex buffer. This is optional and is, for example, not used in the render pipeline concerned with the rect render command.

Lastly, the primitive state contains a description of the kind of primitive the pipeline should render and whether or not any culling should be performed on back-

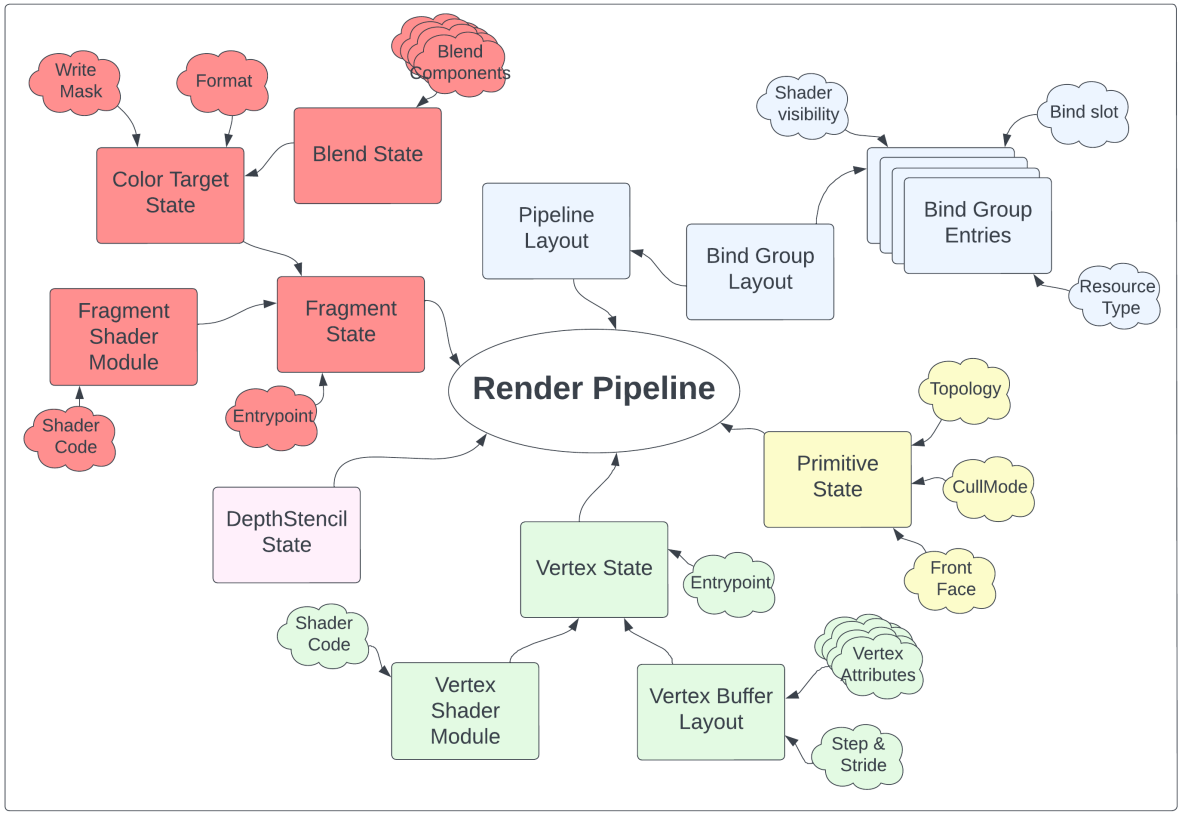

Figure 3.3: Overview of the state contained within the WebGPU render pipeline.

facing or front-facing primitives. For all pipelines present in the implementation presented by this thesis, the primitives used are triangles and the culling is set to cull back-facing geometry.

After all the state has been set at initialization time, there still remain some GPU structures for which the runtime has responsibility. These are bind groups (not layouts), the instance buffer, as well as the swap chain in the case where it needs to be rebuilt, such as when the canvas is resized. The resizing of the swap chain is an implementation detail left to Emscripten and the API that implements WebGPU in the web browser (Dawn for Chrome Canary). Aside from initializing it once in the application it is not manually managed in runtime.

Bind groups are created for every batch depending on the texture and sampling state of the batch in question. The reason these bind groups cannot be built at initialization time is due to how Godot handles item commands. Specifically, in the rect item command, see section 3.3.2, where it simply stores its texture and sampler state as a part of the item command instead of as a bundled material data structure which could have a bind group built for it at time of initialization. Therefore, it is indeterminable how the bind groups will look before the item commands are received by the renderer, resulting in them having to be built at runtime within the render loop.

The instance buffer is rebuilt at least once per frame. Every invocation of `canvas_render_items` initiates a new batch of batches and a new instance buffer is associated with the batches in the current invocation. The instance buffers that were created during a frame are then destroyed before the next frame is rendered.

It is also possible that the swap chain needs to be resized. If this is the case, the implementation would need to rebuild the swap chain during runtime as well. However, in a regular application, such a rebuilding of the swap chain should not need to occur very often and should have a negligible overall impact on the performance of the implementation as a whole.

Lastly, It should be noted that for the buffers being built at initialization time, the data they contain will most likely have to be updated every frame. This also holds true for the instance buffer that is allocated during runtime. It is only the allocation of memory that can be done during the initialization of the renderer.

3.4.3 The Render Loop

The Rasterizer enters the render loop through the `begin_frame` function and finishes upon the `end_frame` function returning. These two functions are both very simple. The former creates a WebGPU command encoder and sets this to be used by the current iteration of the render loop. The latter closes the command encoder and submits it to the WebGPU queue to be executed.

In between these two invocations, all items visible on the canvas are rendered to an off-screen texture. Optionally, before the end of the loop, this off-screen texture is blitted to the back buffer of the swap chain and presented by the swap chain. More information pertaining to the blitting of the main render target to the swap chain is available in section 3.4.4. A high-level overview of the entire render process can be seen in Figure 3.4.

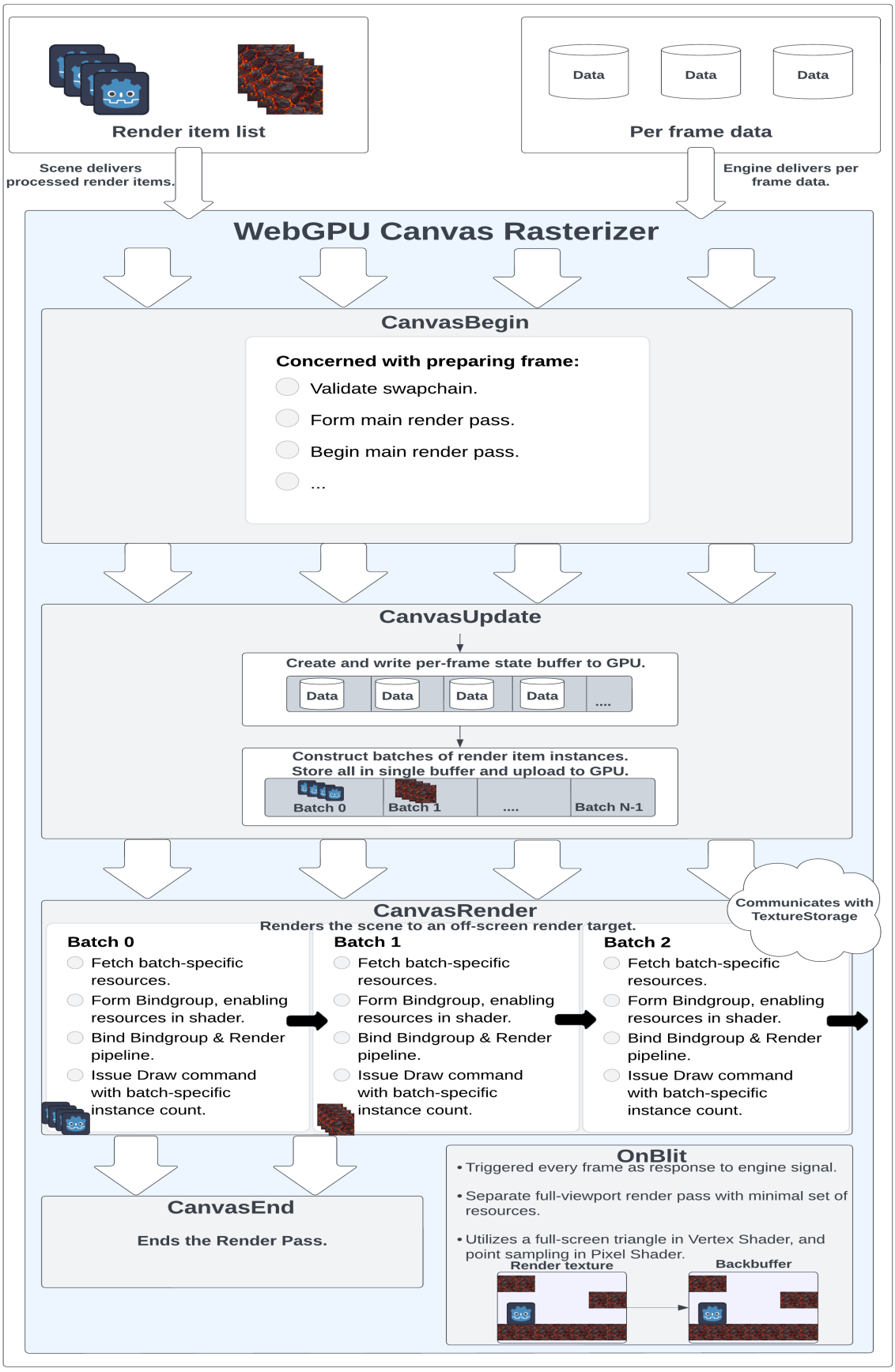

Figure 3.4: Overview of the WebGPU implementation of the Godot render loop.

The renderer receives a call to the `canvas_render_items` function and is passed a linked list data structure of the items it is expected to render as well as a render target texture onto which it should render them. This function comprises four steps. These are the `CanvasBegin`, `CanvasUpdate`, `CanvasRender`, and `CanvasEnd` functions. The Rasterizer begins the process by entering the `CanvasBegin` function wherein it validates its swap chain state. If the swap chain size does not match the one requested, the swap chain is rebuilt. After that, a RenderPassDescriptor with the requested render target as its RenderPassColorAttachment is set up, and using said descriptor, the RenderPassEncoder is built and the main WebGPU render pass is initiated.

Next up, the `CanvasUpdate` function is called. Here, the per-frame data of the render pass is updated. This includes time, transformations of the canvas and screen, and similar render pass-global data. The per-frame data buffer contains some information that is technically unused by the implementation. However, it is made to be the same size as the one that WebGL uses to make the comparison as fair as possible. The `CanvasUpdate` function also batches all the render items before rendering them, see more about batching in section 3.4.4. These batches are iterated over in the third function, `CanvasRender`, where for each batch, the right GPU resources are fetched and bound, including the large instance buffer containing instance-specific data. The resources in the current implementation are limited to textures and these are fetched from a TextureStorage singleton that acts as the owner of all textures in the application. The items contained within the batch are then rendered to the aforementioned render target texture.

Finally, the `canvas_render_items` function enters `CanvasEnd`, wherein the render pass encoder is simply closed and the render pass is finished. Control is then given back to the engine and the engine can decide whether or not to blit the render target to the screen or not. The seemingly most common case is that a blit is requested. Lastly, the command encoder is closed and it is submitted to the WebGPU queue to execute.

3.4.4 Render Techniques

In order for measurements between the performance of the two APIs to be as fair as possible, the WebGPU rendering backend has to adhere to the rendering techniques that Godot employs. The techniques that concern the scaled-down version of the Rasterizer backend include batching and instancing as well as the forcing of render target blitting.

Batching/Instancing – Batching refers to a technique where similar items are gathered and rendered as a batch to make sure all items that use the same resources are rendered at once and avoid binding the same resource multiple times unnecessarily. This technique can be further enhanced by GPU features like instancing to render all items in a batch with a single draw call.

In Godot, batching is done in the rendering backend. However, before items are submitted to be rendered, they have already been sorted such that similar items are submitted contiguously. The batching in the rendering backend is then performed by setting up a batch data structure that contains data that is common to the entire batch. Furthermore, data that is unique to each item in a batch is stored in an

instance buffer. The batch notes where in the instance buffer its items start and how many items it contains, such that the right data is used when later accessing the instance buffer on the GPU during rendering.

The deciding factors of whether a batch needs to be split differs between item types, but in general, batches are split when two subsequent items are sufficiently different. The most obvious example of two items being sufficiently different is them being of different item types. For example, rects and polygons cannot be part of the same batch.

In the scaled-down version of the Rasterizer backend batches of rects can only be split if they differ in the texture they use. This is because no other properties that would warrant a split in the full implementation are supported in the scaled-down version. For polygons, batches are always split due to Godot not supporting the batching of polygons even if they represent the exact same polygon. This holds true both for the scaled-down rendering backend as well as for the fully implemented one.

Lastly, once all the items of the scene have been correctly batched the batches are iterated over. For each batch, its batch-global data is bound through constructing WebGPU bind groups, whereafter a single draw call is issued with an instance count equal to that of the number of items contained in the batch. The instance buffer is also bound by building a bind group and this is then accessed at the right index depending on the render items index of the first item in the batch.

Render Target Blitting – Godot uses two separate render target textures when rendering a two-dimensional scene. One of these is used for rendering the scene items, whereas the other is the image that is ultimately presented to the user of the application. For each frame, Godot decides whether or not the main render target should be blitted to the presented render target. This had to be adhered to in the WebGPU implementation, such that both it and the WebGL implementation perform uniform work.

For blitting, a separate pipeline was set up with a vertex shader that simply renders a triangle covering the entire back buffer, see Figure 3.5, and a fragment shader that textures this triangle using the main render target texture. In effect, this copies the main render target to the render target being presented. However, it is not a direct copy of the data. Rather it scales the image to match the desired viewport size such that the engine can provide users with a fixed aspect ratio, for example.

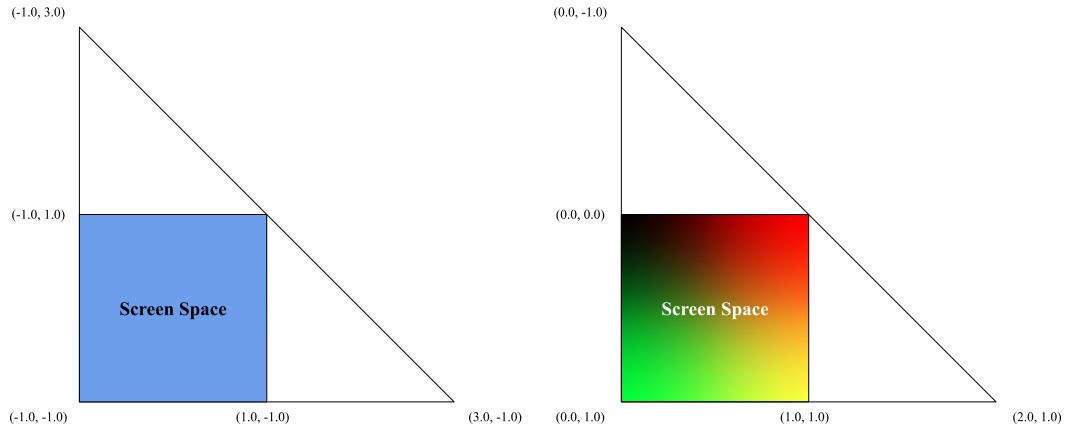

Figure 3.5: A full-screen triangle's positional coordinates (left) and texture coordinates (right).

The blitting is set up as a separate render pass and as such, it is initiated by using the frame's command encoder to construct a RenderPassEncoder with the swap chain back buffer as its ColorTargetAttachment. A point sampler and the render target that was previously used are then bound and the aforementioned full-screen triangle is then textured by sampling points in the render target.

3.5 Experiment and Data Gathering

When it comes to the performance of games and graphical scenes the general consensus of how well something performs is how *smooth* it appears to run to the human eye. For obvious reasons, this holds especially true for non-static scenes where at least some elements are in motion. A common performance property to analyze is that of the frame rate in terms of *frames per second* (FPS). It is often included in game benchmarks and is typically well-known to the intended audience. However, FPS is also considered flawed by nature, which generally comes down to its non-linear scaling property. As a trivial example, if a game renders 59 frames over 0.75 seconds and the next frame takes 0.25 seconds to render, in theory, the 60 frames per second generated sounds good, however in practice the user will have experienced a massive negative spike in performance, affecting the user's sense of how smooth the experience is. A better approach is to use the elapsed *frame time*, which is generally measured in milliseconds. This does scale linearly and any drop will feel less dramatic proportionally speaking [23] [6].

Another reason for omitting frames per second as a performance property comes down to the environment in which the experiment takes place; the web platform. When rendering on the web Chrome Canary will impose a forced vertical synchronization that cannot be turned off, which puts a limit on the maximum frames per second possible, generally defaulting to the refresh rate of the monitor. As such, any test involving a scene that would be possible to render faster than the given monitor's refresh rate would incur a loss in data, effectively making the experiment and gathered data void.

The gathered data in the conducted experiment is that of the frame time measured in milliseconds, for the aforementioned reasons. As the WebGL backend and implemented WebGPU backend spans over both the CPU and GPU in terms of work performed, both the CPU work times and GPU work times are measured. The time gathered is for a full frame for the CPU and GPU. However, for the CPU timings additional measurements are taken for narrower scopes in order to filter out Godot-specific operations that are outside of the scope of the Rasterizers but all the same is included in the full frame measured times. These scopes are:

- `canvas_render_items` - function, which includes all CPU Rasterizer work aside from the blitting operation.
- `ConstructBatches` - function, which includes the work of constructing all the batches and creating and uploading the constant buffers to GPU.
- `RenderBatches` - function, which iterates over all batches, optionally creates and binds WebGPU/WebGL constructs, and performs the necessary render calls.
- `blit_render_targets_to_screen` - function, which, as the name suggests, sets up and performs the blitting of the render target to the back buffer.

The timings are gathered as averages over 2000 frames. Along with average frame times, the means of the 1% highest and the 95% lowest frame times are calculated to be able to analyze the performance consistency between the two Rasterizers. The measurements taken will be used to directly answer the thesis research question, stated in section 3.1.

Following the calculation of the aforementioned mean values, a so-called paired t-test is conducted to show whether the WebGPU and WebGL mean values are significantly different. The t-test produces a t-value that can be used to look up a p-value in a table. The p-value denotes how likely it is that the difference between the two mean values is due to happenstance. For this thesis, it was decided that a p-value less than 0.05 denotes that the two mean values are significantly different. The calculation for the t-value is presented in equation 3.1.

$$t = \frac{\bar{x} - \bar{y}}{\sigma\sqrt{n}} \tag{3.1}$$

In the above equation, t is the t-value, $\bar{x}$ is the mean of the first collection of samples, $\bar{y}$ is the mean of the second collection of samples, σ is the standard deviation of differences between pairs of samples, and n is the number of samples in the two collections.

The measurements of elapsed time on the CPU for the various scopes was measured by using the C++ standard library's chrono header. A timestamp was acquired from `chrono::high_resolution_clock` at the start of the relevant scope and another one at the end of it. To calculate how much time elapsed, the start time stamp was subtracted from the end one. This elapsed time was then stored in a vector and is used later when enough samples have been gathered to calculate an average

elapsed time and the aforementioned 1% highs and 95% lows. A code example can be found in Figure 3.6.

```
auto start = std::chrono::high_resolution_clock::now();
/*
    Code related to the relevant scope
*/
auto end = std::chrono::high_resolution_clock::now();
timeElapsedVector.push_back((end - start).count());

if (timeElapsedVector.size() == SAMPLE_COUNT) {
  CalculateAverages();
}
```

Figure 3.6: Example of code used to time a specific scope on the CPU.

For measuring time on the GPU, different methods need to be used for the different APIs. WebGL provides a way of measuring the elapsed time between two points, whereas WebGPU provides a way to queue a timestamp on the command encoder. If one timestamp is acquired at the start of a frame and one at the end, the elapsed time can be acquired in the same way as described for the CPU measurements.

To acquire a timestamp in WebGPU it is first required that the WebGPU device has the timestamp-query feature available. Next, a WebGPU query set can be created with the timestamp query type. Through the *WriteTimestamp* function, a timestamp can then be written to this query set. In the implementation, the query set is created with a capacity of 4000 queries, one for the start timestamp and one for the end timestamp over 2000 frames. When the capacity has been reached, the query results are written to a buffer in VRAM that is then copied to a read-back buffer such that the timestamps can be read by the CPU and averages can be calculated. Mapping a buffer to be read by the CPU is an asynchronous operation in WebGPU. Thus, it has to be done in a callback function. A code example can be found in Figure 3.7.

```
// Gathering Timestamps
{
  commandEncoder.WriteTimestamp(&timestampQuerySet, timestampCount);
  /*
    Relevant GPU scope
  */
  commandEncoder.WriteTimestamp(&timestampQuerySet, timesampCount + 1);
  timestampCount += 2;

  if (timestampCount == 2 * SAMPLE_COUNT) {
    m_CommandEncoder.ResolveQuerySet(timestampQuerySet, 0, 2 * SAMPLE_COUNT, timestampBuffer, 0);
    m_CommandEncoder.CopyBufferToBuffer(timestampBuffer, 0, readBackTimestamps, 0, 2 * SAMPLE_COUNT * sizeof(
      uint64_t));
  }

  // Frame is ended and the command encoder is submitted to the queue.

  if (timestampCount == 2 * SAMPLE_COUNT) {
  readBackTimestamps.MapAsync(wgpu::MapMode::Read, 0, 2 * SAMPLE_COUNT * sizeof(uint64_t), QueryMapCallback,
     this);
  timeStampCount = 0;
  }

}

// Buffer Map Callback
void QueryMapCallback(WGPUBufferMapAsyncStatus status, void *userData) {
  if (status == WGPUBufferMapAsyncStatus_Success) {
    RasterizerWEBGPU *self = (RasterizerWEBGPU *)userData;
    auto *data = (uint64_t*)self->m_ReadBackTimestamps.GetConstMappedRange(0, 2 * SAMPLE_COUNT * sizeof(uint64_t
     ));

    CalculateAverages(data);

    self->m_ReadBackTimestamps.Unmap();
  } else {
  assert(false && "Failed to map time stamp query set buffer");
  }
}
```

Figure 3.7: Example of measuring elapsed time on the GPU with WebGPU.

In WebGL, queries are started with the `glBeginQuery` function and ended with the `glEndQuery` function. Queries, however, are not guaranteed to be finished just because they have been ended. Thus, a buffer of eight query objects is kept such that a query has eight frames to finish before it gets read by the CPU using the `glGetQueryObjectuiv` function. The buffer of query objects is used as a ring buffer where the query object right after the one currently being written to is read by the CPU. That way, it is free to be used for the next frame's time query. A code example can be found in Figure 3.8.

```
static constexpr uint32_t QUERY_COUNT = 8;

glBeginQuery(GL_TIME_ELAPSED_EXT, queryBuffer[queryCount % QUERY_COUNT]);

// Relevant GPU Scope

glEndQuery(GL_TIME_ELAPSED_EXT);

if (queryCount >= QUERY_COUNT) {
  uint32_t queryIndex = (queryCount + 1) % QUERY_COUNT;
  uint32_t timeElapsed = 0;
  glGetQueryObjectuiv(m_QueryBuffer[queryIndex], GL_QUERY_RESULT, &timeElapsed);
  timeElapsedVectorGPU.push_back(timeElapsed);
}
queryCount++;
```

Figure 3.8: Example of measuring elapsed time on the GPU with WebGL.

The experiments conducted belong to one of two categories; simple 2D games or synthetic tests. For the category of simple 2D games six different games that are simple in scope and complexity were selected. As the Rasterizers are limited in scope, and as the games must be supported by the Godot version used in this work,

the games were selected purely based on the engine's and the two Rasterizers' ability to support and render them. The games are:

1. *Snake* [12], in which the player must avoid obstacles and gather apples in order for the snake character to grow longer and longer.
2. *Evader* [19], in which the player must avoid incoming shapes on the highway.
3. *Checkers*[1] [1], in which the player plays the checkers game either versus an AI or optionally versus another player locally.
4. *Falling Cats* [3], in which the player must catch cats falling from a tree before they hit the ground.
5. *Deck Before Dawn* [22], in which the player strategically plays a number of cards every turn with abilities in order to defend a sleeping child from nightmare creatures.
6. *Ponder*[2] [5], in which the player must navigate a duck character in a finite number of sequences in order to collect all ducklings.

The reader may refer to appendix B for footage of all games running on both the WebGL Rasterizer and the WebGPU Rasterizer.

The synthetic tests are selected in order to test specific areas of rendering and how the Rasterizers compare for each one. As such the synthetic tests are further split into four categories for each specific test case. The synthetic test categories are:

1. **Multiple Quads — Multiple tiny textured sprites**

   The test consists of one big draw call of one batch consisting of instances of textured sprites in the order of 10, 100, 1000, 10000, 20000, 30000, 40000, and 50000 sprites rendered on screen simultaneously, using the shaders for rendering the quad render item type (shaders A.5 and A.6 for WebGPU and shaders A.1 and A.2 for WebGL). Every sprite is its own render item, and as such, the processing of all the sprites and the forming of the instance buffer should put considerate pressure on the CPU. As every sprite is instanced and small in size, relatively low pressure should be put on the Vertex shader stage. The number of fragment shader invocations should increase continuously as more and more sprites are rendered.

2. **Full-screen quads — Multiple full screen textured sprites**

   The test and details regarding it are identical to the aforementioned test with the sole difference that every sprite now is full screen sized. In contrast to the test with multiple small-sized sprites, this test should yield orders of magnitude more fragment shader invocations due to the vast increase in pixel fill rate. As such an overall high burden should be placed on the GPU. This is considered the most demanding stress test.

[1]The version used in testing is v1.0.1-0-g7a4203b

[2]The version used for testing is v1.0.0

3. **Multiple Polygons — Multiple tiny polygons**

   The test consists of one batch per polygon (as Godot has every polygon forming its own batch) in the order of 10, 100, 1000, 10000, 20000, 30000, 40000, and 50000 polygons rendered on screen simultaneously, using the shaders for rendering the polygon render item type (shaders A.7 and A.8 for WebGPU and shaders A.3 and A.4 for WebGL). Every polygon consists of 360 vertices each with vertex attribute data corresponding to 32 bytes. Separate vertex and index buffers are bound prior to every draw call, of which there is one per polygon. This test is made to put a more even burden on the CPU and GPU as the CPU work involving the forming of batches and the render-related calls will increase with an increasing number of polygons. The number of vertex and fragment shader invocations will also increase with an increasing number of polygons.

4. **Large polygons — A few polygons, each with 50000 vertices**

   The test consists of one batch per polygon in the order of 40, 80, 120, 160, 200, 240, 280, and 320 polygons rendered on the screen simultaneously, using the shaders for polygons just like the aforementioned test. As the batch count is kept low due to the low number of polygons being rendered the load put on the CPU work should be kept low too. Instead, the aim of the test is to put considerate pressure on the vertex shader stage as the number of vertices to process will increase significantly with each test increasing the polygon count, up to and including 16 million vertices. As the size of the polygon shapes are kept low the fragment shader stage should not receive much pressure. Overall, this test should put a disproportionate load on the GPU, over the CPU.

3.5.1 Hardware and Software Specification

This section describes the machine that was used to run the test cases on. It presents its hardware as well as what versions of relevant graphics drivers were used. The specifications are presented in Table 3.1.

Table 3.1: Information about hardware and software versions of the machine upon which all test cases were run.

Component	
CPU	Intel Core i7 12700H, 2.7GHz
GPU	NVIDIA GeForce RTX 3070 Ti (Laptop Version), 8GB GDDR6
Memory	SK Hynix, 2x8GB DDR4, 3.2GHz
Disk	Samsung MZVL21T0HCLR-00B07, 1TB, 7.0/5.1 GB/s
Monitor Resolution	2560x1440
Monitor Refresh Rate	165Hz
Operating System	Windows 11 Home 22H2
NVIDIA Driver Version	531.41
Emscripten	3.1.30
Chrome Canary	114.0.5715.1
Godot Engine Version	4.0

3.6 Alternative Approaches

It was established in section 3.5 that frame time is the most suitable candidate for what data to gather in this work. Lower frame times result in a smoother experience for users observing scenes being rendered in real-time. An alternative approach that could be postulated could be that of replacing or complementing the quantitative approach with a qualitative one, in which a number of subjects analyze the rendered scenes produced by both Rasterizers. Then, they could be asked to provide feedback on the perceived smoothness as part of some questionnaire. However, such an experiment (and similar ones), would be ill-suited for the intended research goal of the study, as frame time is a precise metric, and human perception is not. For instance, it would be impossible for such a subject to know the distribution of frame time due to different workloads put on the CPU and GPU. For this, and other obvious reasons, such experiment alternatives were discarded.

Originally, this thesis aimed to analyze the performance differences between WebGPU and WebGL in Godot using two metrics, frame time and VRAM usage. Adding VRAM to the analysis another dimension would be introduced in determining the performance of the two APIs; efficiency in terms of memory utilization on GPU. However, upon contacting and discussing the issue with the Google engineers responsible for implementing WebGPU on Chrome Canary through the Dawn API, it was discovered that measuring the VRAM usage with WebGPU is, as of writing this thesis, not possible. This comes down to reasons ranging from non-portability, issues with so-called fingerprinting, and a lack of features exposing it through the underlying APIs. In other words, it is too early in the development phase to be supported in WebGPU. The only method possible, according to the Google engineers, would be to estimate the usage based on the GPU resources created. However, the accuracy of such an estimate is questionable, especially as the underlying resource management for managing multiple frames in flight is entirely implementation dependent and should be viewed as a black box. This would lead to sketchy comparisons with WebGL at best, and therefore it was determined that the data gathering of VRAM usage should be omitted.

3.7 Validity and Reliability of Approach

In order to reach a high validity and reliability of the approach, much time was dedicated to implementing a solution with an emphasis on *fairness*, or rather, *minimizing unfairness*. As has already been discussed in this chapter, this, in large part, had to do with implementing a Rasterizer that was as identical in function to the WebGL equivalent as possible. The existing framework helped in doing so, as it forced the implementation to adhere to the already established render paths and dedicated functionality. One way of validating the result is to compare the produced output of the two Rasterizers. Put simply, for any one game, either both Rasterizers succeed or both fail at rendering the game. Otherwise, one Rasterizer would be more feature complete than the other, which would indicate different computation capabilities. For verifying this, 19 2D games were tested. Of these six games were able to be rendered with both Rasterizers, and the rest failed with no output produced at all.

The six games also produced identical outputs for both of the Rasterizers and were thus selected as candidates for the performance comparisons. The reader may verify that the produced output is indeed identical by looking at appendix B. While the state of the games is not identical (as this is hard to produce due to natural game randomness), one should be able to verify that the games are identical running on the two Rasterizers. By the same token, examples of the visual results of the synthetic tests can be observed in appendix C.

The combined facts that the Rasterizers have succeeded in producing the same output while having been created as mirror images of each other make for a strong case that a solution with an emphasis on minimizing unfairness has been achieved. However, it is of course not guaranteed as some possible aspects might affect the experiments. For one, Godot is a large and complex environment to place the experiment within and it is difficult to realize all possible ways this could affect the experiment.

Another important aspect to consider when profiling games and other real-time applications is the complexity of the scene; e.g. in a 3D open-world game, the complexity of the scene can vary heavily depending on the position and orientation of the virtual camera. However, this will not be an issue for the experiments involved in this thesis work, as it deals only with simple 2D games with static camera views into the scene, resulting in a scene where the complexity is roughly the same. Despite this, due to the random nature of some games, the scene complexity will vary to some degree. For the synthetic tests, however, it is completely identical in every case.

A final point to make is that of producing a result that can be considered generalizable. The aim of the thesis work is to produce knowledge on what can be generalizable in the scope of its environment, that is, the Godot game engine, which influences the nature of the results. It is important to realize that the produced result will give a hint as to what can be expected in an environment outside of Godot (such as another game engine or free-standing application) rather than a promise, as those environments will come with their own set of rules that might impact the results.

Chapter 4

Results and Analysis

This chapter aims to present all the data that has been gathered during the conducted tests in a comparative format between the WebGPU Rasterizer and the WebGL Rasterizer. The chapter starts with a section presenting the results of a performance comparison when profiling the various games detailed in section 3.5, followed by an equivalent section on the conducted synthetic tests. For both cases, the results are split into GPU and CPU frame time comparisons.

The GPU sections contain two graphs and the underlying data in a table for the GPU frame times measured during the relevant test. The first of the graphs presents the mean GPU frame time and the second presents both the highest 1% mean of GPU frame times and the lowest 95% mean, effectively ignoring the 5% worst frame times in an attempt to adjust for unpredictable spikes.

For the CPU sections, what graphs are included varies from test to test, depending on what test results were deemed most noteworthy. However, all graphs are available for the reader in appendix D. For any graphs of the ConstructBatches scope, the appendix also needs to be consulted due to them being completely excluded from this section. The reasoning behind this choice is that the ConstructBatches scope takes place purely on the CPU and has no direct interaction with the respective graphics API. There is, however, still some variation between the two implementations.

The chapter is rounded off with an analysis summary.

4.1 Understanding the Data

When studying the various graphs and tables presented in this chapter (and the additional appendix material) the reader should be informed on what the different functions executed on the CPU pertains. Most details have been discussed already in section 3.5, and so the reader is referred there for more details as to what the functions do.

It is important to realize that both the *RenderBatches* function and the *ConstructBatches* function are contained within the *canvas_render_items* function. The profiling data is gathered per these function scopes in order to gain a greater understanding of how the two Rasterizers more narrowly perform to another; they have been deemed most important for analyzing the relative CPU performance between the Rasterizers. Also, aside from these functions, the whole CPU frame is profiled as well. This is important, as including the full CPU frame timings can produce knowledge relating to e.g. how GPU/CPU synchronization introduces stalling and waiting times that affect the overall performance.

In order to further understand how WebGPU performs compared to WebGL, a speed-up factor is calculated and included for all CPU and GPU mean frame times. Also, the measurement for the likelihood that the observed differences between the groups are due to chance (the *p-value*) is calculated and included for all mean, high 1% mean, and low 95% mean CPU and GPU frame times. The WebGPU speed-up factor is denoted $S_{Latency}$ and the p-value is denoted p. They are both included in all tables presented. In cases where the p-value is greater than 0.05, the data is color-coded red for the reader's convenience.

4.2 Performance Comparison - Game tests

This section presents the measurements gathered from the various game tests. The first section presents the GPU frame time and the second the total CPU frame time as well as the various scopes within it that were also measured.

4.2.1 GPU Frame Time

Below are the graphs and table showing the data gathered on the GPU frame time in the various game tests.

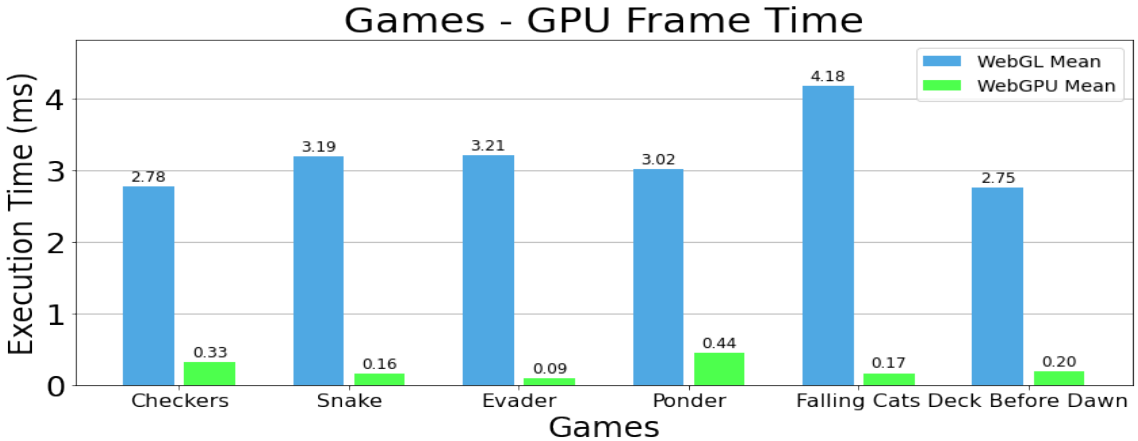

Figure 4.1: Comparison of the mean WebGL and WebGPU GPU frame times, in milliseconds, for the various games. Lower is better.

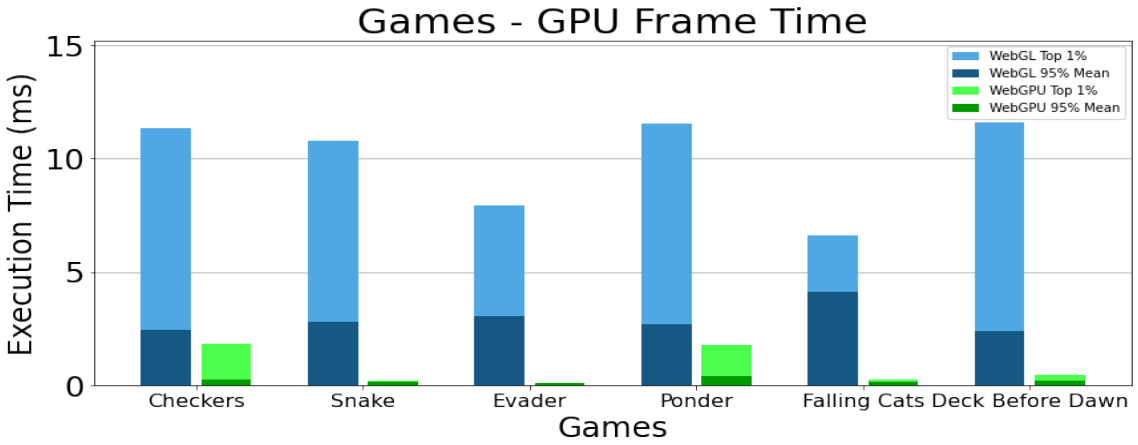

Figure 4.2: Comparison of the highest 1% mean and the lowest 95% mean WebGL and WebGPU GPU frame times, in milliseconds, for the various games. Lower is better.

Table 4.1: Mean, highest 1% mean, and lowest 95% mean GPU frame times, in milliseconds, for the various games. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean GPU times (ms)				High 1% GPU times (ms)			Low 95% GPU times (ms)		
Game	WebGPU	WebGL	$S_{Latency}$	p	WebGPU	WebGL	p	WebGPU	WebGL	p
Checkers	0.326	2.775	8.512	<0.001	1.797	11.343	<0.001	0.256	2.444	<0.001
Snake	0.163	3.191	19.578	<0.001	0.175	10.774	<0.001	0.163	2.814	<0.001
Evader	0.090	3.205	35.611	<0.001	0.095	7.944	<0.001	0.090	3.048	<0.001
Ponder	0.443	3.022	6.822	<0.001	1.790	11.518	<0.001	0.377	2.683	<0.001
Falling Cats	0.165	4.181	25.339	<0.001	0.263	6.604	<0.001	0.163	4.099	<0.001
Deck Before Dawn	0.196	2.753	14.046	<0.001	0.461	11.591	<0.001	0.187	2.362	<0.001

In Figure 4.1, it can be seen that WebGPU on average has much shorter GPU frame times than WebGL in all games that were included in the test. Table 4.1 shows a speed-up factor ranging between 6.822, in the case of Ponder, and 35.611, in the case of Evader. Figure 4.2 shows that the difference between the lowest 95% of frame times and the highest 1% is larger for WebGL. However, for Checkers and Ponder and Falling Cats, the percentage difference is more significant for WebGPU. For checkers, this comes out to a 7.020 times increase for WebGPU compared to a 4.641 times increase for WebGL. For Ponder, the increase is 4.748 times for WebGPU and 4.292 times for WebGL. Lastly, for Falling Cats, WebGPU shows a 1.613 times increase and WebGL shows a 1.611 times increase. For the other games, WebGPU has a smaller spread in absolute and percentage terms.

4.2.2 CPU Frame Time

Below are the graphs for the CPU frame time measured for the various game tests.

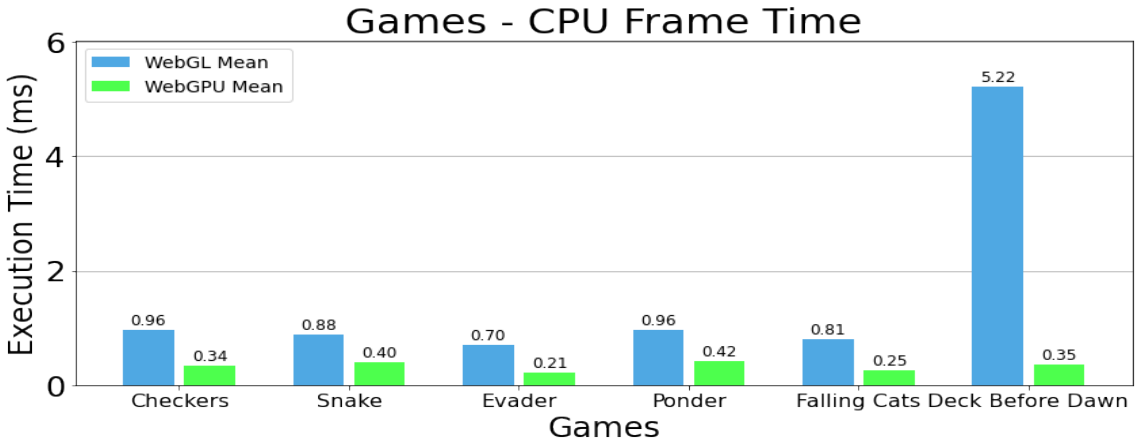

Figure 4.3: Comparison of the mean WebGL and WebGPU CPU frame times, in milliseconds, for the various games. Lower is better.

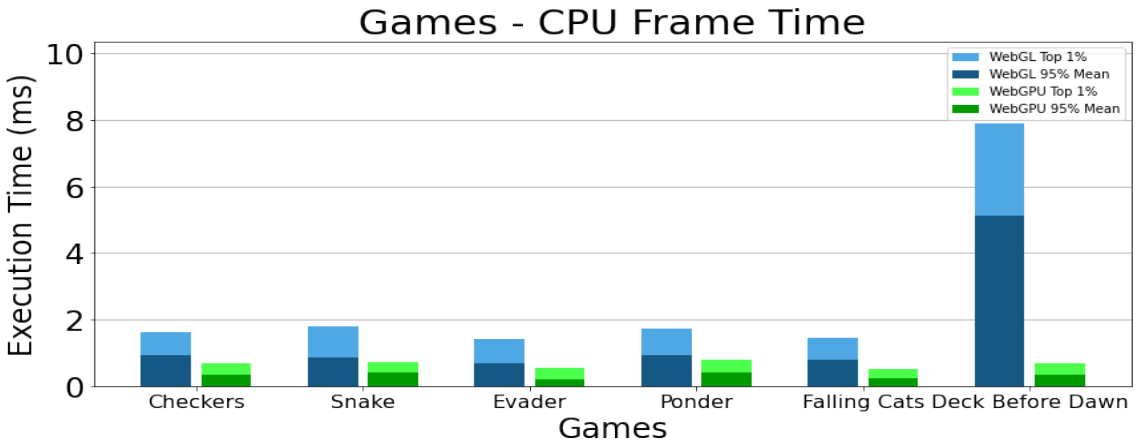

Figure 4.4: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU CPU frame times, in milliseconds, for the various games. Lower is better.

In Figure 4.3, it is shown that WebGPU has shorter mean frame times for all of the game tests compared to WebGL. Deck Before Dawn is a clear outlier in the data set in how much shorter the CPU frame time is with the WebGPU implementation. Figure 4.4 shows that the percentage differences between the lowest 95% and highest 1% of frame times are typically lower compared to the spread documented for GPU frame times in Figure 4.2. This does, however, not hold true for all cases. For instance, Evader shows a larger spread for WebGPU in CPU frame time than it did for the GPU.

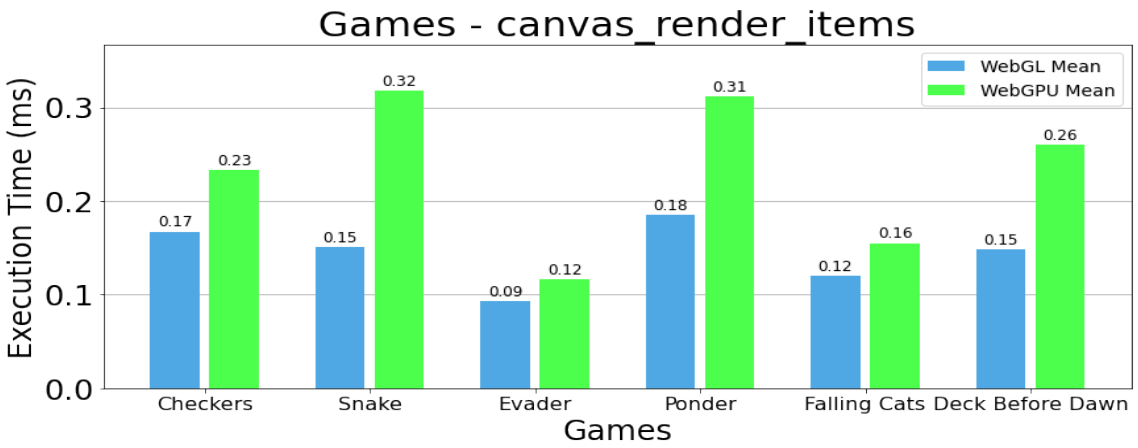

Figure 4.5: Comparison of the mean WebGL and WebGPU canvas_render_items frame times, in milliseconds, for the various games. Lower is better.

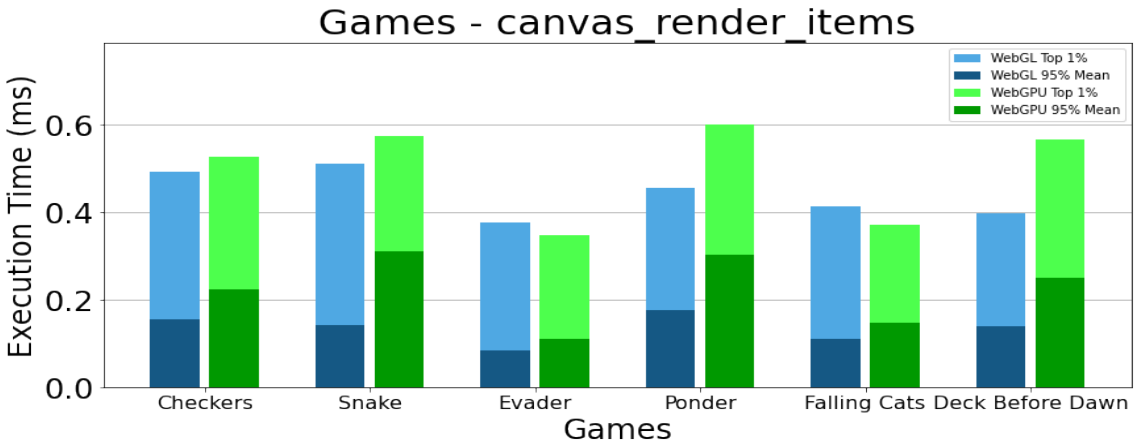

Figure 4.6: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU canvas_render_items frame times, in milliseconds, for the various games. Lower is better.

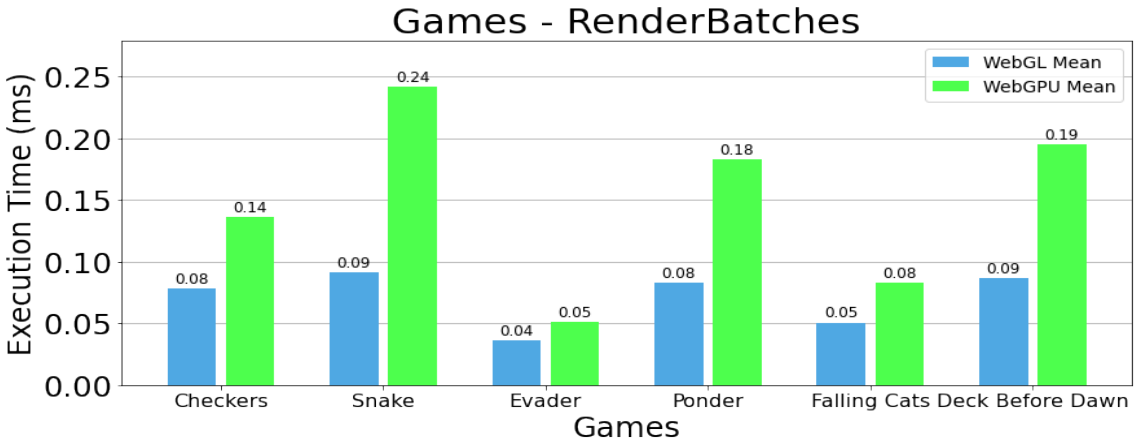

Figure 4.7: Comparison of the mean WebGL and WebGPU RenderBatches frame times, in milliseconds, for the various games. Lower is better.

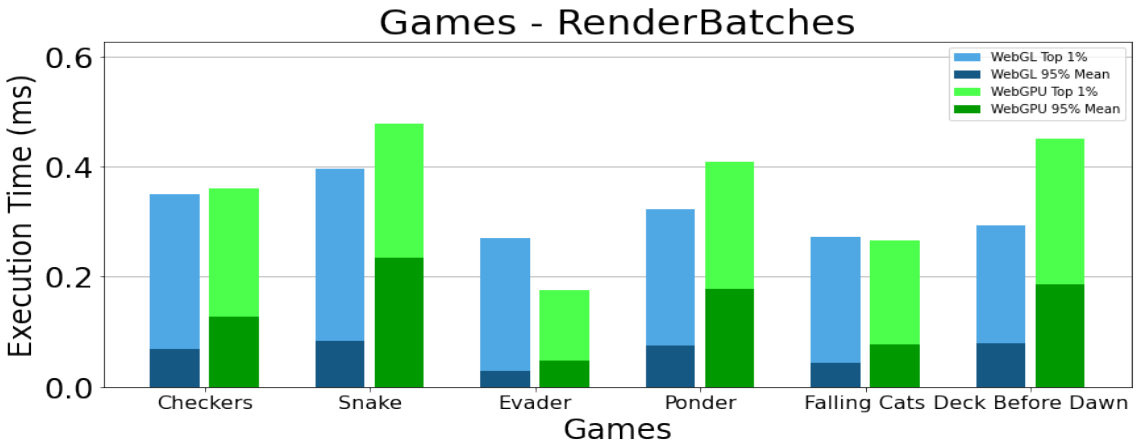

Figure 4.8: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU RenderBatches frame times, in milliseconds, for the various games. Lower is better.

In figures 4.5 and 4.7 a break in the trend of the WebGPU implementation being on average faster than WebGL one can be seen. The times measured are short in both cases, but in the most extreme example, Snake, WebGPU takes 2.133 times longer for `canvas_render_items` and 2.667 times longer for `RenderBatches`.

The difference between the lowest 95% and the highest 1% of measured times is shown in figures 4.6 and 4.8. These can be seen to be similar between the games with Evader and Falling Cats showing the most significant spread for both implementations in both `canvas_render_items` and `RenderBatches`.

Table 4.2: Mean, highest 1% mean, and lowest 95% mean for overall CPU, canvas_render_items, ConstructBatches & RenderBatches CPU times, in milliseconds, for the game Checkers. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean CPU times (ms)				High 1% CPU times (ms)			Low 95% CPU times (ms)		
Checkers	WebGPU	WebGL	p	$S_{Latency}$	WebGPU	WebGL	p	WebGPU	WebGL	p
Overall CPU	0.341	0.959	<0.001	2.812	0.678	1.633	<0.001	0.329	0.936	<0.001
canvas_render_items	0.233	0.167	<0.001		0.526	0.493	0.107	0.223	0.155	<0.001
ConstructBatches	0.018	0.011	<0.001		0.135	0.130	0.722	0.016	0.009	<0.001
RenderBatches	0.136	0.078	<0.001		0.361	0.351	0.474	0.127	0.068	<0.001

Table 4.3: Mean, highest 1% mean, and lowest 95% mean for overall CPU, canvas_render_items, ConstructBatches & RenderBatches CPU times, in milliseconds, for the game Snake. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean CPU times (ms)				High 1% CPU times (ms)			Low 95% CPU times (ms)		
Snake	WebGPU	WebGL	p	$S_{Latency}$	WebGPU	WebGL	p	WebGPU	WebGL	p
Overall CPU	0.404	0.877	<0.001	2.171	0.719	1.805	<0.001	0.395	0.848	<0.001
canvas_render_items	0.318	0.151	<0.001		0.575	0.512	0.419	0.310	0.142	<0.001
ConstructBatches	0.036	0.022	<0.001		0.122	0.074	<0.001	0.034	0.021	<0.001
RenderBatches	0.242	0.091	<0.001		0.478	0.396	0.289	0.235	0.084	<0.001

Table 4.4: Mean, highest 1% mean, and lowest 95% mean for overall CPU, canvas_render_items, ConstructBatches & RenderBatches CPU times, in milliseconds, for the game Evader. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean CPU times (ms)				High 1% CPU times (ms)			Low 95% CPU times (ms)		
Evader	WebGPU	WebGL	p	$S_{Latency}$	WebGPU	WebGL	p	WebGPU	WebGL	p
Overall CPU	0.214	0.703	<0.001	3.285	0.540	1.416	<0.001	0.205	0.677	<0.001
canvas_render_items	0.116	0.093	<0.001		0.348	0.376	0.557	0.110	0.084	<0.001
ConstructBatches	0.006	0.004	<0.001		0.025	0.037	0.051	0.005	0.003	<0.001
RenderBatches	0.051	0.037	<0.001		0.176	0.271	0.040	0.048	0.029	<0.001

Table 4.5: Mean, highest 1% mean, and lowest 95% mean for overall CPU, canvas_render_items, ConstructBatches & RenderBatches CPU times, in milliseconds, for the game Ponder. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean CPU times (ms)				High 1% CPU times (ms)			Low 95% CPU times (ms)		
Ponder	WebGPU	WebGL	p	$S_{Latency}$	WebGPU	WebGL	p	WebGPU	WebGL	p
Overall CPU	0.415	0.956	<0.001	2.304	0.773	1.734	<0.001	0.405	0.926	<0.001
canvas_render_items	0.311	0.185	<0.001		0.600	0.455	<0.001	0.303	0.175	<0.001
ConstructBatches	0.081	0.052	<0.001		0.210	0.216	0.730	0.303	0.175	<0.001
RenderBatches	0.183	0.083	<0.001		0.409	0.322	<0.001	0.177	0.075	<0.001

Table 4.6: Mean, highest 1% mean, and lowest 95% mean for overall CPU, canvas_render_items, ConstructBatches & RenderBatches CPU times, in milliseconds, for the game Falling Cats. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean CPU times (ms)				High 1% CPU times (ms)			Low 95% CPU times (ms)		
Falling Cats	WebGPU	WebGL	p	$S_{Latency}$	WebGPU	WebGL	p	WebGPU	WebGL	p
Overall CPU	0.250	0.807	<0.001	3.228	0.519	1.461	<0.001	0.241	0.783	<0.001
canvas_render_items	0.155	0.120	<0.001		0.372	0.412	0.095	0.147	0.110	<0.001
ConstructBatches	0.014	0.008	<0.001		0.084	0.071	0.277	0.012	0.007	<0.001
RenderBatches	0.083	0.051	<0.001		0.266	0.273	0.585	0.078	0.043	<0.001

Table 4.7: Mean, highest 1% mean, and lowest 95% mean for overall CPU, canvas_render_items, ConstructBatches & RenderBatches CPU times, in milliseconds, for the game Deck Before Dawn. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean CPU times (ms)				High 1% CPU times (ms)			Low 95% CPU times (ms)		
Deck Before Dawn	WebGPU	WebGL	p	$S_{Latency}$	WebGPU	WebGL	p	WebGPU	WebGL	p
Overall CPU	0.350	5.220	<0.001	14.914	0.677	7.893	<0.001	0.340	5.129	<0.001
canvas_render_items	0.260	0.148	<0.001		0.565	0.398	<0.001	0.251	0.139	<0.001
ConstructBatches	0.025	0.018	<0.001		0.121	0.088	0.017	0.023	0.016	<0.001
RenderBatches	0.195	0.087	<0.001		0.451	0.293	<0.001	0.187	0.080	<0.001

Tables 4.2 through 4.7 show the underlying data for figures 4.3 through 4.8. They also show a speed-up factor between WebGL and WebGPU for the mean total CPU frame time in the game tests. This speed-up can be seen to range between 2.171 and 14.583, with WebGPU showing a positive speed-up in all test cases despite its being slower in all the measured scopes it contains.

4.3 Performance Comparison - Synthetic tests

This section details the profiling results of the synthetic tests, starting with the results for the GPU frame times and being followed up by the CPU frame times. Like with the games section every test type is dealt with in turn and separately.

4.3.1 GPU Frame Time

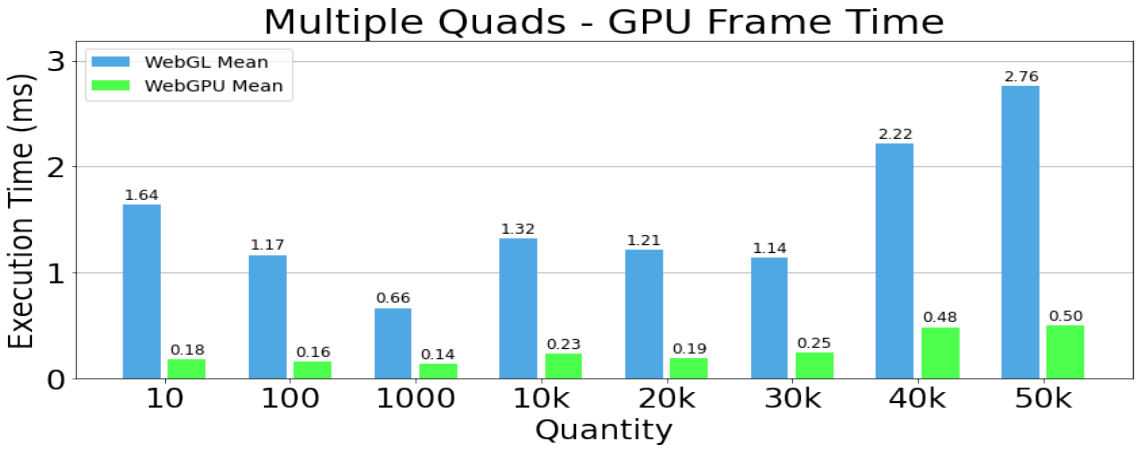

Figure 4.9: Comparison of the mean WebGL and WebGPU GPU frame times, in milliseconds, for the Multiple Quads test. Lower is better. The workloads range from 10 to 50.000 quads.

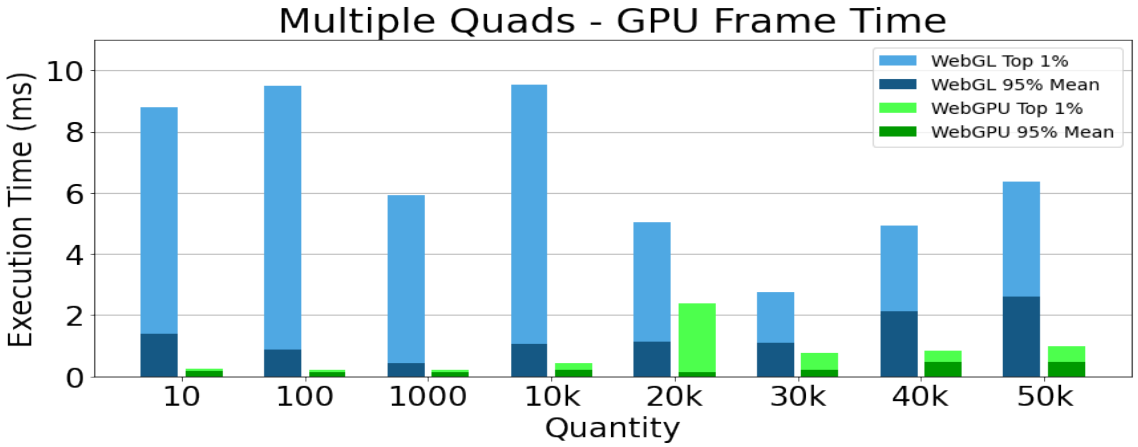

Figure 4.10: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU GPU frame times, in milliseconds, for the Multiple Quads test. Lower is better. The workloads range from 10 to 50.000 quads.

Table 4.8: Resulting mean GPU frame times, in milliseconds, for the WebGPU and WebGL Rasterizers rendering the multiple quads. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean GPU times (ms)				High 1% GPU times (ms)			Low 95% GPU times (ms)		
Multiple Quads	WebGPU	WebGL	$S_{Latency}$	p	WebGPU	WebGL	p	WebGPU	WebGL	p
10	0.181	1.636	9.039	<0.001	0.260	8.790	<0.001	0.180	1.381	<0.001
100	0.155	1.166	7.523	<0.001	0.221	9.495	<0.001	0.153	0.879	<0.001
1000	0.138	0.664	4.812	<0.001	0.218	5.909	<0.001	0.134	0.433	<0.001
10k	0.228	1.325	5.811	<0.001	0.428	9.532	<0.001	0.223	1.044	<0.001
20k	0.190	1.211	6.374	<0.001	2.387	5.053	<0.001	0.130	1.119	<0.001
30k	0.247	1.141	4.619	<0.001	0.773	2.772	<0.001	0.223	1.091	<0.001
40k	0.483	2.216	4.588	<0.001	0.849	4.929	<0.001	0.468	2.117	<0.001
50k	0.501	2.762	5.513	<0.001	0.981	6.356	<0.001	0.481	2.599	<0.001

For the synthetic test involving rendering multiple quads WebGPU outperforms WebGL in all cases in GPU mean frame times, as can be clearly seen in Figure 4.9. The speed-up factor ranges from 4.588, as is the case when rendering 40 000 quads, up to 9.039, as is the case when rendering ten quads (see Table 4.8).

Furthermore, the frame times for WebGPU are also more stable overall than WebGL, which has a vastly higher spread, as is evident in Figure 4.10.

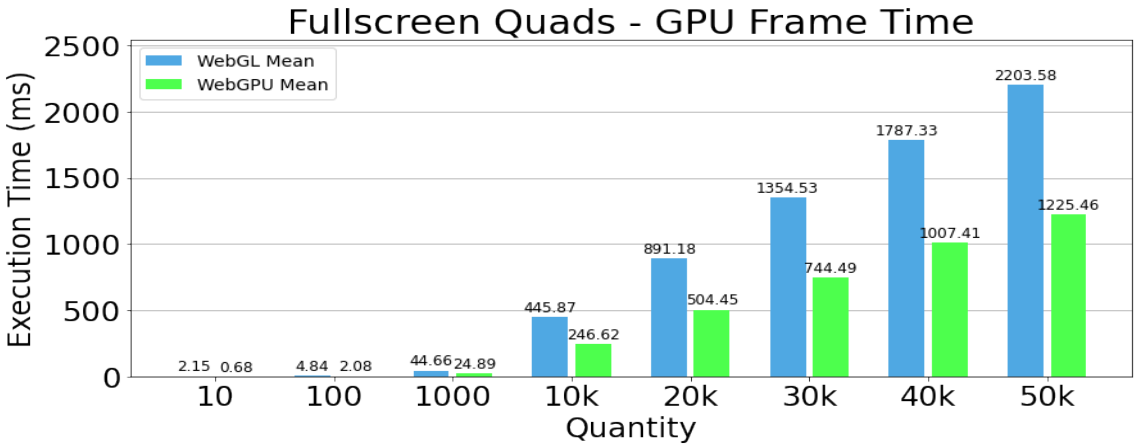

Figure 4.11: Comparison of the mean WebGL and WebGPU GPU frame times, in milliseconds, for the Full-screen quads test. Lower is better. The workloads range from 10 to 50.000 full-screen quads.

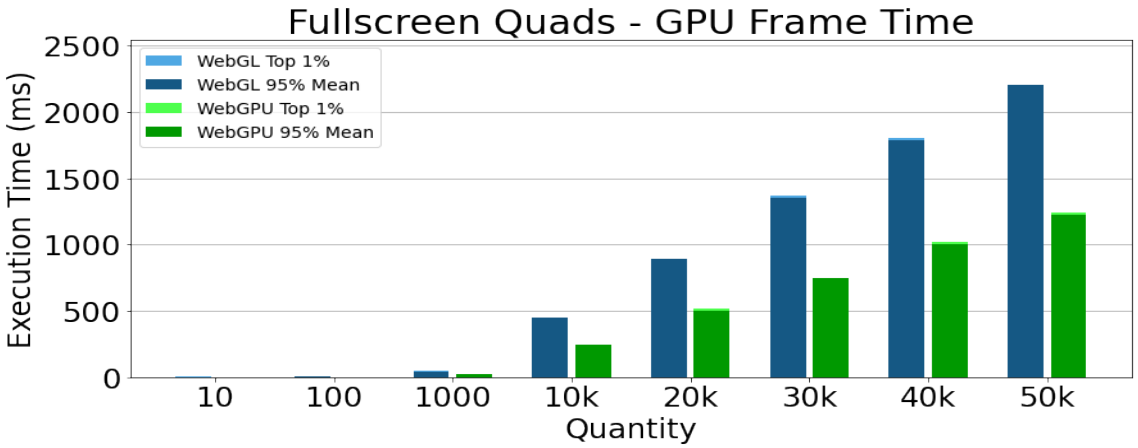

Figure 4.12: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU GPU frame times, in milliseconds, for the Full-screen quads test. Lower is better. The workloads range from 10 to 50.000 full-screen quads.

Table 4.9: Resulting mean GPU frame times for the WebGPU and WebGL Rasterizers rendering the full-screen quads. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean GPU times (ms)				High 1% GPU times (ms)			Low 95% GPU times (ms)		
Full-Screen Quads	WebGPU	WebGL	$S_{Latency}$	p	WebGPU	WebGL	p	WebGPU	WebGL	p
10	0.685	2.151	3.140	<0.001	1.942	10.315	<0.001	0.652	1.893	<0.001
100	2.078	4.837	2.328	<0.001	2.380	7.566	<0.001	2.074	4.735	<0.001
1000	24.891	44.664	1.794	<0.001	25.611	45.373	<0.001	24.865	44.635	<0.001
10k	246.616	445.865	1.808	<0.001	248.262	451.459	<0.001	246.549	445.784	<0.001
20k	504.454	891.181	1.767	<0.001	516.710	892.149	<0.001	504.086	891.156	<0.001
30k	744.488	1354.534	1.819	<0.001	747.311	1366.234	<0.001	744.372	1354.051	<0.001
40k	1007.410	1787.326	1.774	<0.001	1024.746	1802.513	<0.001	1006.823	1786.851	<0.001
50k	1225.461	2203.585	1.798	<0.001	1246.317	2206.178	<0.001	1224.881	2203.508	<0.001

The results of rendering multiple full-screen quads show how considerate pressure was put on both Rasterizers, with long GPU mean frame times for all tests above 1000 quads. Looking at Figure 4.11 both Rasterizers show an approximately linear

increase in frame time as the number of quads increases, with WebGPU being roughly 1.8 - 3.1 times faster than WebGL depending on the profiling context.

The relative spread of frame times is less tangible at these high frame times, as is evident from Figure 4.12, and from Table 4.9, which show results close to that of the mean frame times.

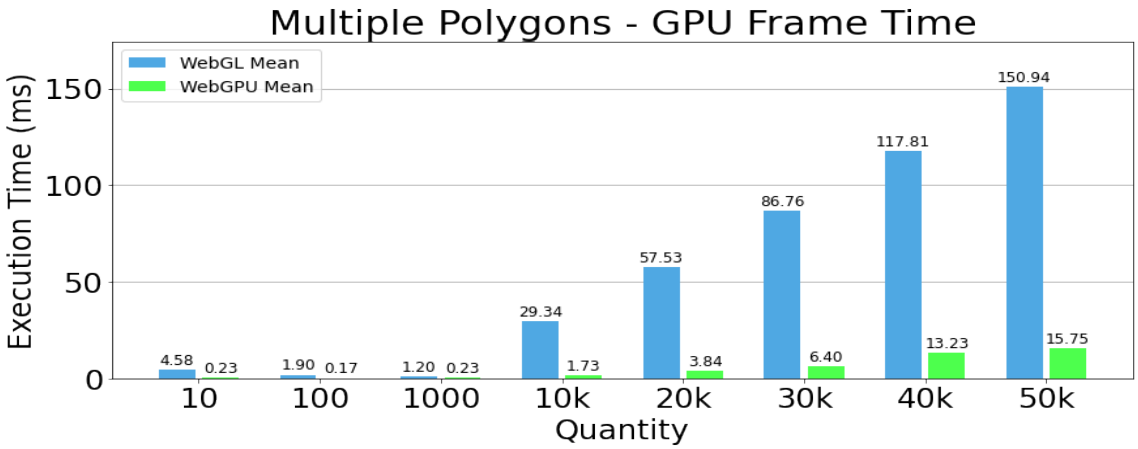

Figure 4.13: Comparison of the mean WebGL and WebGPU GPU frame times, in milliseconds, for the Multiple Polygons test. Lower is better. The workloads range from 10 to 50.000 polygons.

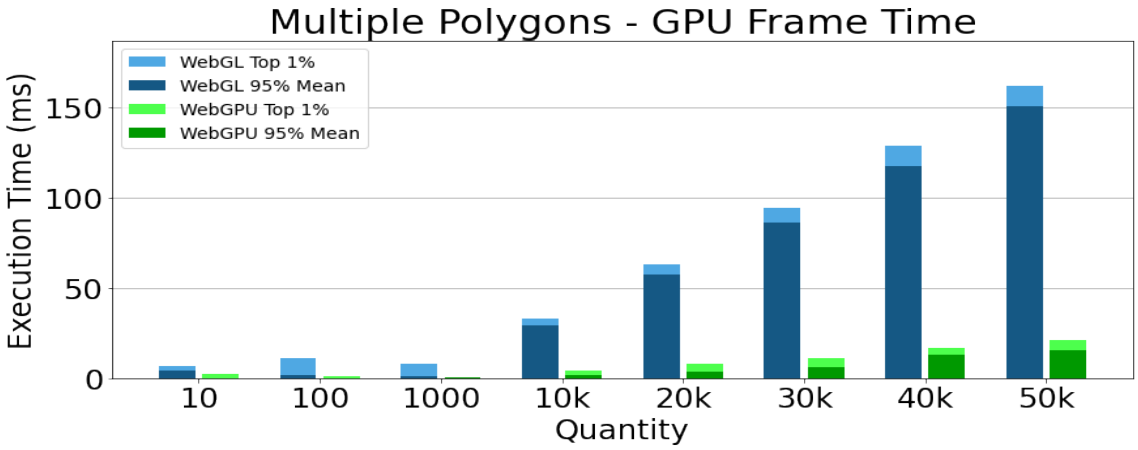

Figure 4.14: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU GPU frame times, in milliseconds, for the Multiple Polygons test. Lower is better. The workloads range from 10 to 50.000 polygons.

Table 4.10: Resulting mean GPU frame times, in milliseconds, for the WebGPU and WebGL Rasterizers rendering the multiple polygons. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean GPU times (ms)				**High 1% GPU times (ms)**			**Low 95% GPU times (ms)**		
Multiple Polygons	WebGPU	WebGL	$S_{Latency}$	p	WebGPU	WebGL	p	WebGPU	WebGL	p
10	0.227	4.579	20.172	<0.001	2.083	6.712	<0.001	0.139	4.505	<0.001
100	0.168	1.896	11.286	<0.001	0.963	10.922	<0.001	0.151	1.480	<0.001
1000	0.226	1.195	5.288	<0.001	0.534	8.172	<0.001	0.218	1.000	<0.001
10k	1.727	29.345	16.992	<0.001	4.123	32.912	<0.001	1.614	29.231	<0.001
20k	3.835	57.533	15.002	<0.001	7.796	63.086	<0.001	3.641	57.317	<0.001
30k	6.404	86.756	13.547	<0.001	11.028	94.496	<0.001	6.182	86.448	<0.001
40k	13.230	117.810	8.905	<0.001	16.700	128.586	<0.001	13.142	117.400	<0.001
50k	15.755	150.939	9.580	<0.001	21.386	161.820	<0.001	15.630	150.512	<0.001

The Polygons synthetic tests show the biggest comparative GPU frame time differences between the two Rasterizers, with WebGPU vastly outperforming WebGL in every case. As an example, looking at Figure 4.13, at the point of rendering 50 000 polygons WebGL manages an average of 150.94 milliseconds per frame while WebGPU is still running at passable real-time speeds (15.75 milliseconds, equivalent to more than 60 frames per second).

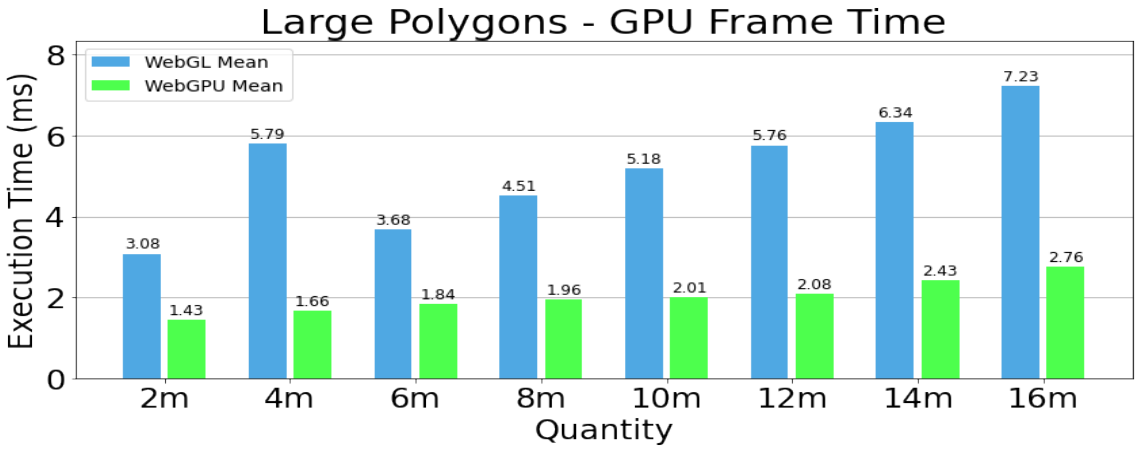

Figure 4.15: Comparison of the mean WebGL and WebGPU GPU frame times, in milliseconds, for the Large Polygons test. Lower is better. The workloads range from 2 million to 16 million vertices.

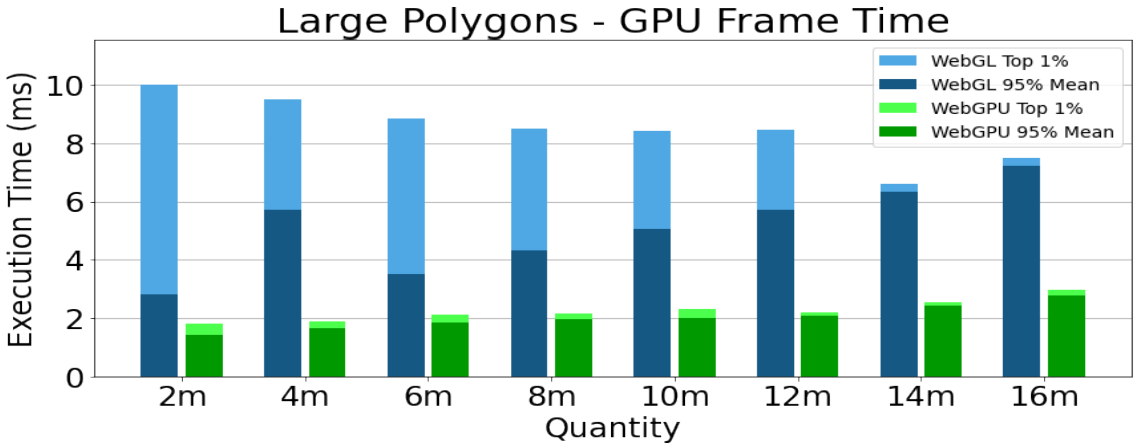

Figure 4.16: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU GPU frame times, in milliseconds, for the Large polygons test. Lower is better. The workloads range from 2 million to 16 million vertices.

Table 4.11: Resulting mean GPU frame times, in milliseconds, for the WebGPU and WebGL Rasterizers rendering the large polygons. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean GPU times (ms)				High 1% GPU times (ms)			Low 95% GPU times (ms)		
Large Polygons	WebGPU	WebGL	$S_{Latency}$	p	WebGPU	WebGL	p	WebGPU	WebGL	p
2m	1.432	3.078	2.149	<0.001	1.828	10.007	<0.001	1.428	2.827	<0.001
4m	1.661	5.788	3.485	<0.001	1.893	9.495	<0.001	1.659	5.719	<0.001
6m	1.840	3.683	2.002	<0.001	2.118	8.856	<0.001	1.836	3.517	<0.001
8m	1.958	4.506	2.301	<0.001	2.142	8.484	<0.001	1.956	4.313	<0.001
10m	2.011	5.175	2.573	<0.001	2.307	8.427	<0.001	2.008	5.051	<0.001
12m	2.085	5.759	2.762	<0.001	2.192	8.469	<0.001	2.083	5.706	<0.001
14m	2.429	6.338	2.609	<0.001	2.560	6.602	<0.001	2.428	6.334	<0.001
16m	2.765	7.228	2.614	<0.001	2.982	7.495	<0.001	2.762	7.223	<0.001

The GPU frame times for the Large Polygons synthetic test show results of WebGPU being roughly 2 - 3 times faster across the various workloads. The frame time increases roughly linearly for the Rasterizers with greater workloads, with a statistical deviation occurring at 4 million polygons for WebGL. Like with the test of rendering multiple quads, WebGL again shows a bigger spread of frame times, with WebGPU remaining fairly stable (see Figure 4.15 and 4.16).

4.3.2 CPU Frame Time

Quads

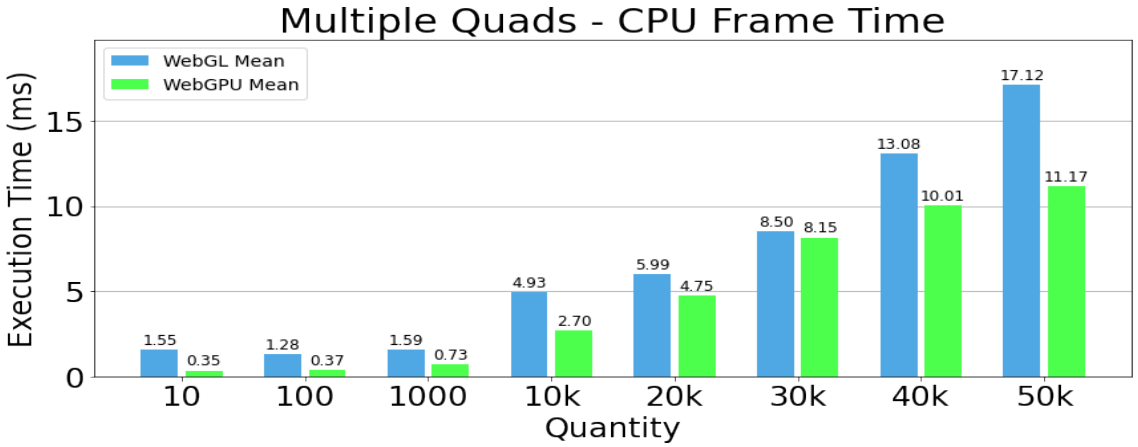

Figure 4.17: Comparison of the mean WebGL and WebGPU CPU frame times, in milliseconds, for the Multiple Quads test. Lower is better. The workloads range from 10 to 50.000 quads.

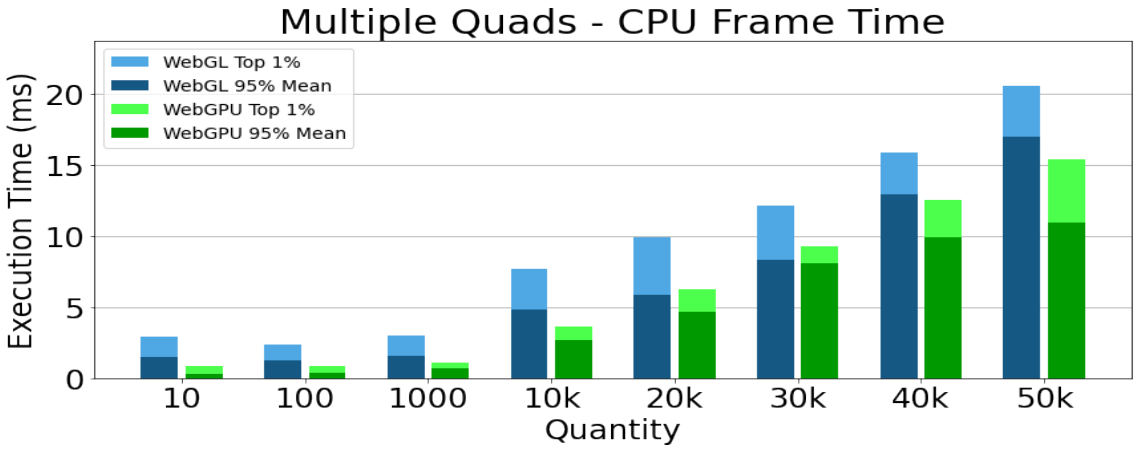

Figure 4.18: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU CPU frame times, in milliseconds, for the Multiple Quads test. Lower is better. The workloads range from 10 to 50.000 quads.

For the Quads synthetic test, Figure 4.17 shows that WebGPU performs better in total CPU frame time compared to WebGL. It is also shown in the graph that the gap between them, with the exception of the 30 000 quads variant, increases the more items are being rendered.

Figure 4.18 shows that, for the workloads of 10 000 quads and up, both WebGL and WebGPU have fairly consistent frame times, with little spread between the lowest 95% and the highest 1% of measured times. The exception to this is for the 50 000 quads test, where WebGPU shows a larger difference than WebGL both in quantity and in proportional increase.

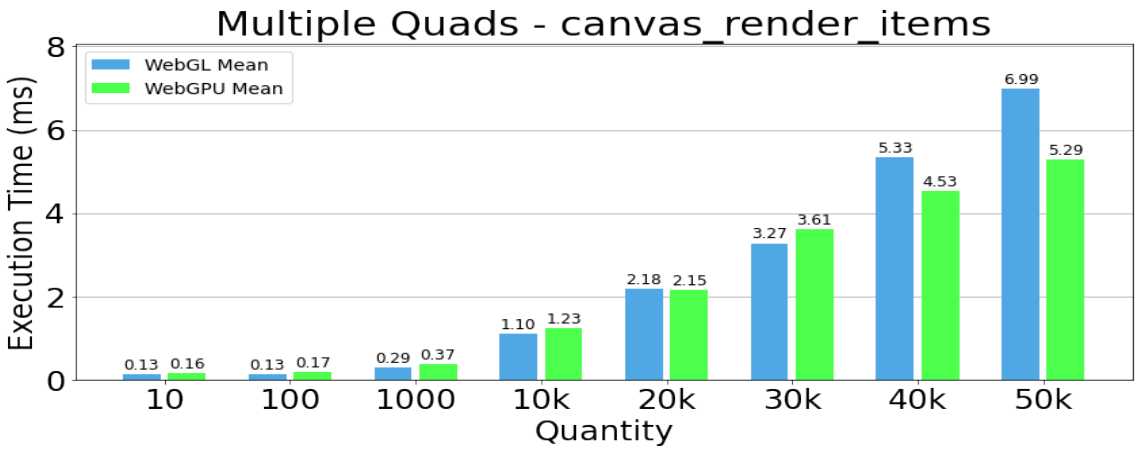

Figure 4.19: Comparison of the mean WebGL and WebGPU canvas_render_items frame times, in milliseconds, for the Multiple Quads test. Lower is better. The workloads range from 10 to 50.000 quads.

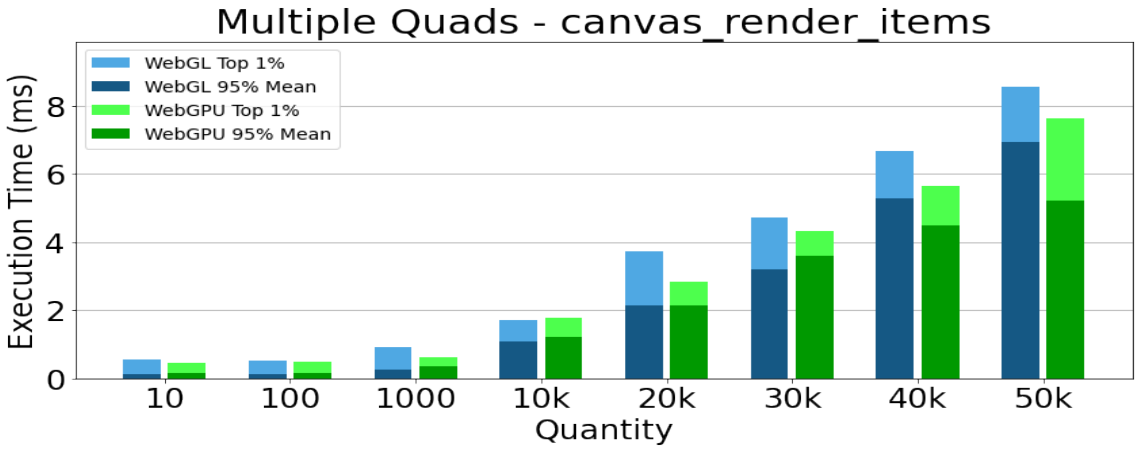

Figure 4.20: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU canvas_render_items frame times for the Multiple Quads test. Lower is better. The workloads range from 10 to 50.000 quads.

In terms of the `canvas_render_items` scope, the quads test shows that the WebGL implementation is faster than the WebGPU one for all workloads below 20 000 quads, see Figure 4.19. Thereafter, WebGPU can be seen to be faster excepting the 30 000 quads test. The gap between the two also appears to grow as the workload increases past 40 000 quads.

In Figure 4.20, it is shown that the proportional spread of the `canvas_render_items` times follows a similar pattern to that shown for total CPU frame time in Figure 4.18. The difference in quantity, however, is different as the time taken for the `canvas_render_items` scope is shorter than that of the total CPU frame.

The `RenderBatches` times measured for the Quads test are very short. However, the WebGPU implementation always performs worse than the WebGL implementation. There is also no clear pattern that depends on the workload variants.

Full-screen Quads

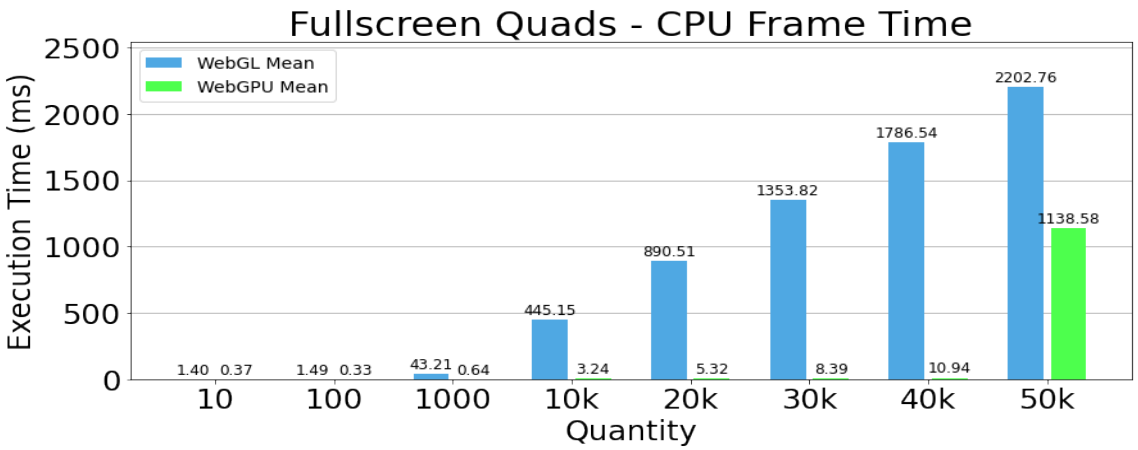

Figure 4.21: Comparison of the mean WebGL and WebGPU CPU frame times, in milliseconds, for the Full-screen Quads test. Lower is better. The workloads range from 10 to 50.000 full-screen quads.

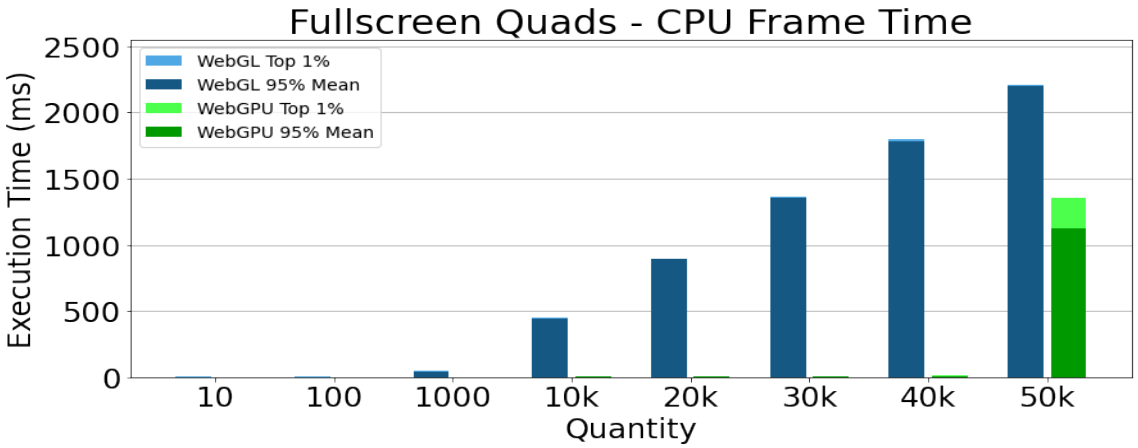

Figure 4.22: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU CPU frame times, in milliseconds, for the Full-screen Quads test. Lower is better. The workloads range from 10 to 50.000 full-screen quads.

For the Full-screen Quads synthetic tests WebGL can be seen to have a mostly linear increase in mean CPU frame time following from the number of quads that need to be rendered, see Figure 4.21. However, for WebGPU, while there is a fairly steady increase for all workloads below 50 000 quads, the 50 000 quads variant has a much larger frame time than the 40 000 quads variant. The frame time for this variant looks very similar to the GPU frame time presented in Figure 4.11.

Regarding the differences between the 95% lowest and 1% highest frame times, Figure 4.22 shows that the spread is insignificant in almost all cases. This is due to the frame time already being very long and the difference being less than 100 milliseconds. The outlier here is, again, the 50 000 quads variant for WebGPU where this difference is clearly shown to be much larger. It should, however, be

noted that despite the spread for the other variants looking minuscule in the above graph, it still amounts to multiple tens of milliseconds, which in a case with shorter overall times would be devastating for the frame rate.

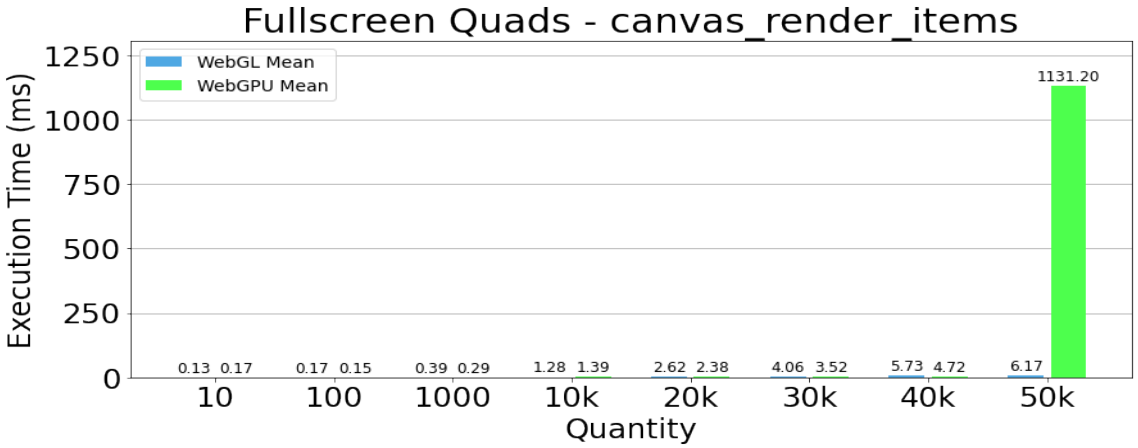

Figure 4.23: Comparison of the mean WebGL and WebGPU canvas_render_items frame times, in milliseconds, for the Full-screen Quads test. Lower is better. The workloads range from 10 to 50.000 full-screen quads.

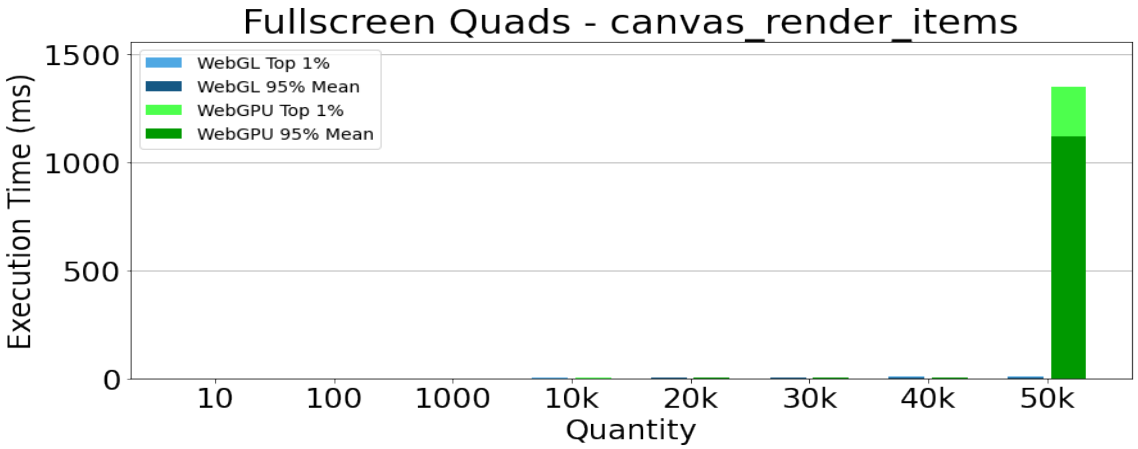

Figure 4.24: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU canvas_render_items frame times, in milliseconds, for the Full-screen Quads test. The workloads range from 10 to 50.000 full-screen quads.

The graphs associated with `canvas_render_items` seen in figures 4.23 and 4.24 there is a steady increase in frame time for both WebGL and WebGPU. However, WebGPU for the 50 000 quads variant gets a much longer time that is close to the mean CPU frame time presented in Figure 4.21 and the mean GPU frame time presented in Figure 4.11. Again, the spread is large for the WebGPU 50 000 quads variant. The differences for the other variants are difficult to discern in the graph but come out to less than 5 milliseconds in all cases.

As with the aforementioned Quads test, the `RenderBatches` times measured for the Full-screen Quads test are very short and show similar times as the Quads test

does. However, again, there is no apparent connection between workload size and time taken for the `RenderBatches` scope.

Polygons

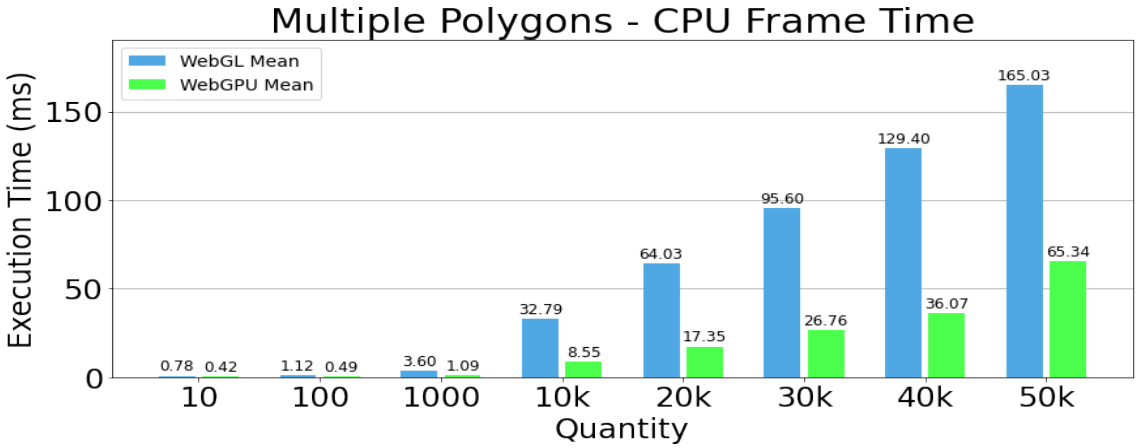

Figure 4.25: Comparison of the mean WebGL and WebGPU CPU frame times, in milliseconds, for the Multiple Polygons test. The workloads range from 10 to 50.000 polygons.

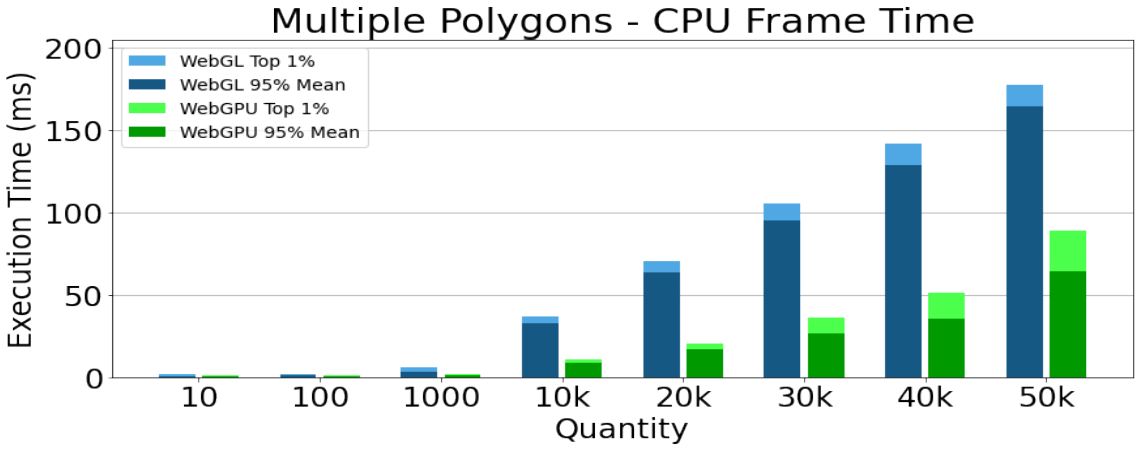

Figure 4.26: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU CPU frame times, in milliseconds, for the Multiple Polygons test. Lower is better. The workloads range from 10 to 50.000 polygons.

The total CPU frame time, Figure 4.25, of the Polygons synthetic tests shows that, outside of the 10 polygon test case, the WebGPU implementation is faster than the WebGL one. It also shows that an increase in rendered polygons causes a nearly linear increase in frame time, where the increase for the WebGL Rasterizer is steeper than that of the WebGPU one. Notably, for WebGPU, the step from 40 000 polygons to 50 000 polygons breaks the previous pseudo-linearity and causes the frame time to increase more significantly.

For the spread of frame times, Figure 4.26 shows that the proportional difference between the lowest 95% and the highest 1% of frame times is rather small. However, it should be noted that in terms of absolute quantity, this spread is still significant due to the longer frame times of the test. Note that the spread is larger for the WebGPU implementation both in terms of the absolute difference and the proportional difference.

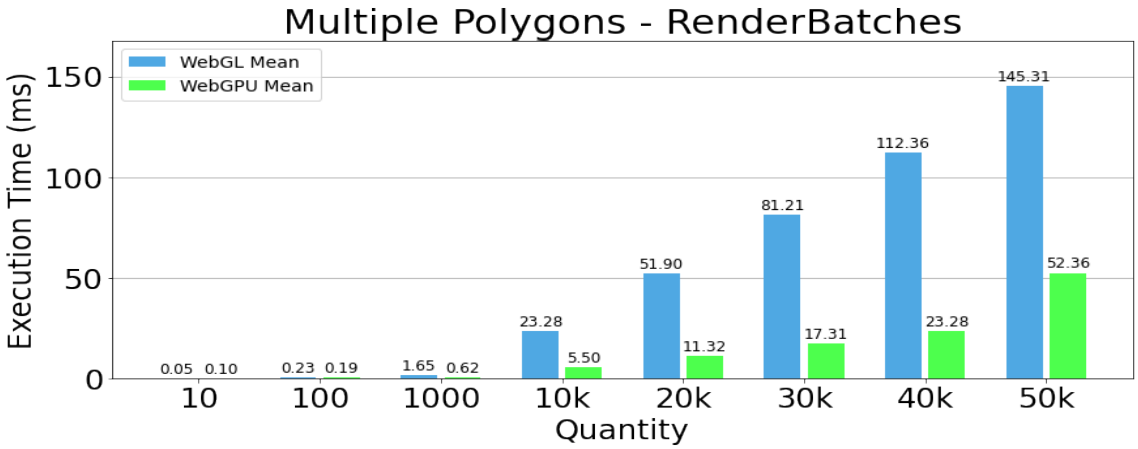

Figure 4.27: Comparison of the mean WebGL and WebGPU RenderBatches frame times for the Multiple Polygons test. Lower is better. The workloads range from 10 to 50.000 polygons.

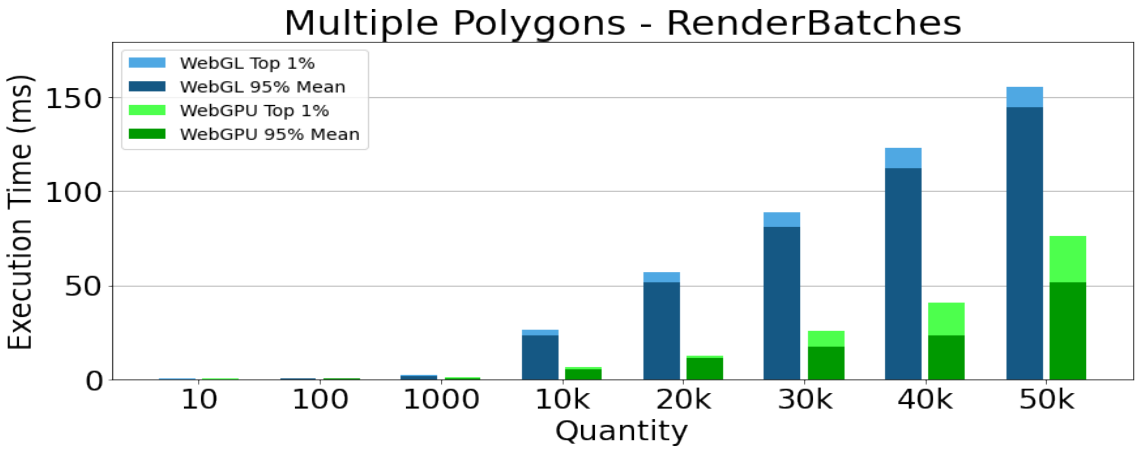

Figure 4.28: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU RenderBatches frame times, in milliseconds, for the Multiple Polygons test. Lower is better. The workloads range from 10 to 50.000 polygons.

For the Polygons test, the `RenderBatches` scope is the main contributor to the long frame times. A similar pattern to that of the aforementioned total CPU frame time can be found in the `RenderBatches` times. This is the case for both mean time and spread between the lowest 95% and highest 1% of measured times.

Large Polygons

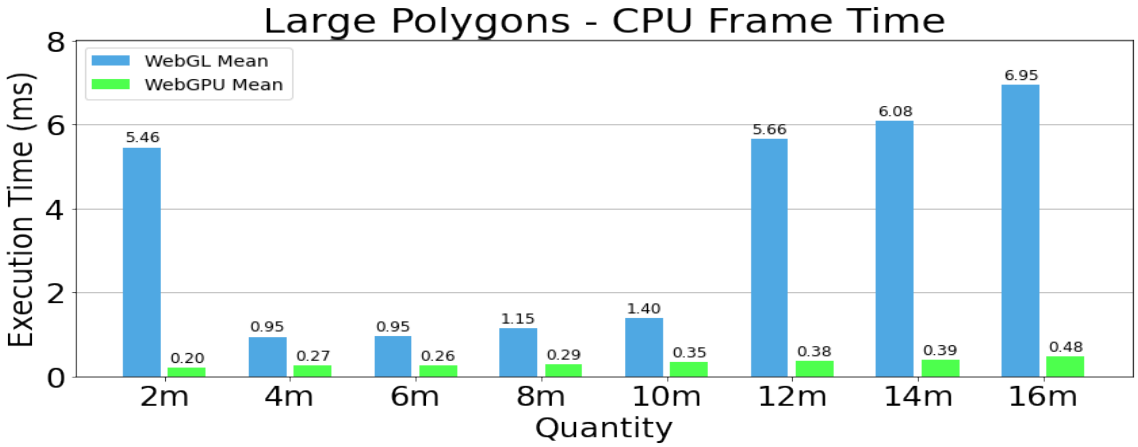

Figure 4.29: Comparison of the mean WebGL and WebGPU CPU frame times, in milliseconds, for the Large Polygons test. Lower is better. The workloads range from 2 million to 16 million vertices.

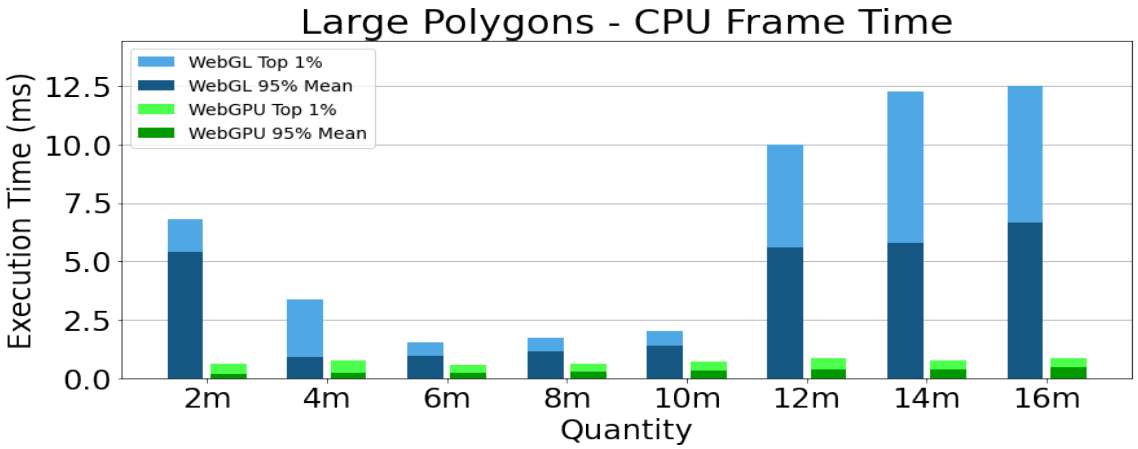

Figure 4.30: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU CPU frame times, in milliseconds, for the Large Polygons test. Lower is better. The workloads range from 2 million to 16 million vertices.

In the Large Polygons synthetic test, the total CPU time recorded for the two Rasterizers in Figure 4.29 shows that WebGPU has a very small steady increase following an increase in workload, whereas the WebGL implementation varies seemingly settling into a steady increase after 12 million vertices (240 polygons).

In terms of the difference between the lowest 95% and the highest 1%, Figure 4.30 shows that the WebGPU implementation, in line with the mean times, has a very similar spread notwithstanding the workload of the test. WebGL, however, shows relatively small differences for the smaller test sizes, excepting the 4 million vertices variant, and relatively large differences for the higher vertex count variants.

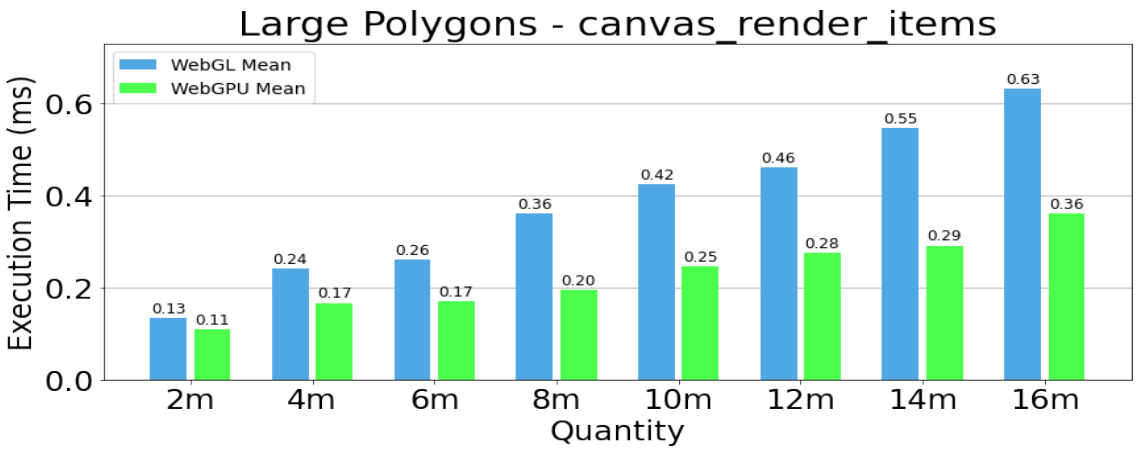

Figure 4.31: Comparison of the mean WebGL and WebGPU canvas_render_items frame times, in milliseconds, for the Large Polygons test. Lower is better. The workloads range from 2 million to 16 million vertices.

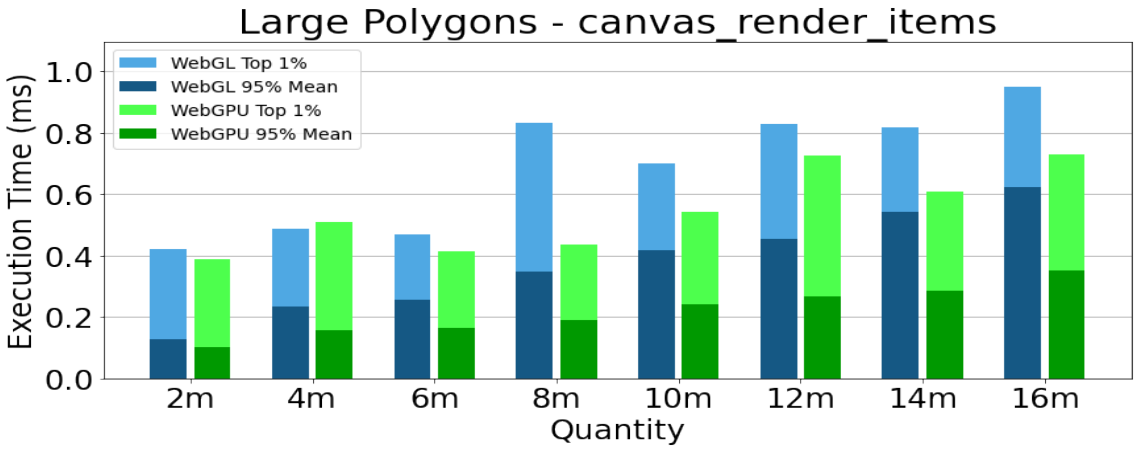

Figure 4.32: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU canvas_render_items frame times, in milliseconds, for the Large Polygons test. Lower is better. The workloads range from 2 million to 16 million vertices.

The mean times for the `canvas_render_items` scope, as shown in Figure 4.31, display a fairly linear increase in time taken for both implementations, with WebGPU having a less steep increase than WebGL. The increase isn't strictly linear, however, as the WebGPU 4 million and 6 million vertex variants both show the exact same time.

Interestingly, the proportional differences, as can be seen in Figure 4.32, between the two implementations are quite similar and relatively large. They also vary a lot more than the steady increase that was seen in the mean times for the same tests.

Lastly, it should be mentioned that for all tests except for the 50 000 quads variant of the Quads test, the WebGPU implementation is slower than the WebGL implementation when it comes to `ConstructBatches`. The graphs and tables for this data, as well as the tables for all other CPU side measurements of the synthetic tests, can be found in appendix D.

4.4 Statistical Significance

In studying the calculated p-values for the various means presented in this chapter as well as in appendix D, it is discovered that most are significantly different. However, in the case of the means of the full sample collections, the `RenderBatches` times for the 40 000 quads synthetic test shows a p-value of 0.116 which is greater than 0.05. Thus, WebGL and WebGPU are not significantly different in this test. This is also supported by the fact that the graph in Figure D.11 shows the two being almost equal.

For the means calculated for the highest 1% of measured times, p-values exceeding 0.05 are often presented for the times measured on the CPU. However, in the case of times measured on the GPU, the p-values show that the two APIs are always significantly different.

Lastly, for the lowest 95% of measured times, the `canvas_render_items` time for the 20 000 quads synthetic test is the only time the calculated p-value exceeds 0.05. Thereby, in accordance with the similarity shown in Figure 4.20, WebGL and WebGPU are not significantly different in this regard for the chosen sample count.

4.5 Analysis Summary

In all games, WebGPU outperforms WebGL both in terms of overall CPU and GPU mean frame times. WebGPU performs worse in the `ConstructBatches` and `RenderBatches` functions, especially for the game Snake, however, the margins are very small in general. WebGPU also remains more stable overall with the lowest mean 95% frame times being closer to the mean frame times, as well as having fewer frame time spikes. This holds especially true for the GPU timings, as the Rasterizers perform more equally on the CPU side.

The synthetic tests show similar results, with WebGPU outperforming WebGL in both CPU and GPU frame time performance. This is especially prominent in the case of rendering multiple polygons and multiple quads. WebGL also shows more fluctuating frame times, as heavily evident by the multiple quads and the large polygons tests. WebGPU is always slower in the `ConstructBatches` function, with the exception of rendering 50 000 quads, which is the only statistical anomaly to the case. The `RenderBatches` function performs better on the WebGPU Rasterizer when the tests involve many batches (Polygons and Large Polygons tests).

As evident by the previous section, *Statistical Significance*, the gathered data pairs are by and large significantly different from each other, with the majority of samples falling below a p-value of 0.001. This holds especially true for the overall mean CPU and GPU values, as well as the 95% low mean CPU and GPU values. For the data on the high 1%, this is also always significantly different on GPU, however many cases exist where this is not the case on CPU, as is evident by the Full-Screen Quads and Multiple Quads synthetic tests (Table E.3 and E.2).

Chapter 5

Discussion

This chapter will discuss the results presented in chapter 4, Results and Analysis. First, the research question stated in section 3.1 will be answered. Subsequently, the outcome of the results will be discussed, as well as existing limitations regarding their analysis.

5.1 Research Question & Answers

RQ: What is the difference in performance regarding CPU and GPU frame time between WebGPU and WebGL when used as a rendering backend in the Godot game engine?

The data gathered and presented in the previous chapter, *Results and Analysis*, as well as the additional material in the equivalent appendix section, has clearly answered the Research Question. Additional details follow below:

5.1.1 Mean CPU & GPU Frame Time

For both the synthetic tests and the games tests, WebGPU outperforms WebGL in terms of mean GPU frame time for every experiment made. For the games, this is evident from looking at Figure 4.1 and Table 4.1. The speed-up of using the WebGPU Rasterizer ranges from roughly 6.8 - 35.6, a significant result. This trend continues for the synthetic experiments made. The smallest speed-up measured occurred at the point of the Full-Screen Quads test, of rendering 20 000 quads, where a speed-up of 1.767 was reached (see Table 4.9). In contrast, the biggest comparative difference in terms of performance reached for the synthetic tests was found upon performing the Polygons experiment, where a speedup of 20.172 was reached, which the reader may verify in Table 4.10. See section 5.2 for a discussion on possible explanations as to the nature of these results.

The mean CPU frame times are also faster across every experiment made over the two test types. For the games, this is evident from looking at tables 4.2 - 4.7. The CPU speed-ups for the games are more modest than the equivalent GPU speed-ups, with the majority being around 2 - 3 times faster on WebGPU. *Deck Before Dawn* is a noteworthy exception to this as it performs particularly worse on WebGL than WebGPU (Roughly 15 times slower). However, the cause of this discrepancy has not been determined and remains unknown. The game was also remeasured which yielded similar results.

Studying the mean synthetic tests CPU results found in tables E.1 - E.4, it can be seen that WebGPU performs better than WebGL. For the Full-Screen Quads test, the difference in mean CPU frame times is very big for the tests of rendering 1000 - 40 000 quads, with speedups in the range of 160 - 170. A possible explanation for this is due to how WebGL is stalling on the GPU on those workloads, whereas WebGPU is not, resulting in longer synchronization/waiting times on the CPU. While the CPU frame times are even worse on WebGL for rendering 50 000 quads, the speed up has dropped significantly, down to 1.935, as WebGPU suddenly struggles with vastly longer CPU frame times than normal. This sudden increase in frame time may be due to WebGPU being forced to synchronize the GPU and CPU when copying all batch data contents to the GPU. See section 5.2 for more discussion.

Another discrepancy between the two Rasterizers can be found by looking at the mean CPU times for the cases where the workloads are very light. For rendering 2 million Large Polygons, the WebGL Rasterizer takes an average of 5.459 ms to finish a frame, with WebGPU taking 0.198 ms, producing a speed-up factor of 27.571. This gap is diminished in the following test of rendering 4 million Large Polygons, as the WebGL Rasterizer then only takes 0.947 ms to produce a frame on the CPU. This phenomenon, while not as pronounced, is seen again on the test of rendering multiple quads, as the test switches from rendering ten quads, to rendering 100 quads (Table E.2). This may be yet again due to stalling. However, it is not evident from looking at the equivalent mean GPU frame time graphs (figures 4.15 and 4.9). It is the case that rendering 100 quads is faster than rendering ten. However, it is not the case that rendering 4 million large polygons is faster than rendering 2 million. So the results are not as consistent as to be able to draw any certain conclusions.

5.1.2 Main CPU Function Performance

While the mean CPU frame times are decidedly faster on the WebGPU Rasterizer across all tests, it is often not the case that the main Rasterizer functions (`canvas_render_items`, `ConstructBatches` & `RenderBatches`) implemented are. For the games, the WebGL Rasterizer consistently produces a faster result, as verified in Tables 4.2–4.7.

For the synthetic tests `ConstructBatches` is always faster on the WebGL Rasterizer, except for rendering 50 000 full-screen quads, which seems to be a statistical anomaly. However, the times measured for both implementations are always fairly similar. This result is expected considering the `ConstructBatches` scope performs no direct communication with the graphics API and therefore should remain mostly unaffected by what API is being used. This fact has led to the conclusion that the implementations of the `ConstructBatches` scope are mostly equivalent with minor differences caused by architectural mismatches.

Furthermore, WebGL performs better in certain cases when it comes to the `RenderBatches` scope. From what the data shows, it can be gleaned that these cases are when only a few batches need to be rendered. A good example of this can be seen in the Polygons synthetic test, wherein WebGL outperforms WebGPU in terms of the `RenderBatches` measurements for ten polygons. After that, however, WebGPU is always faster. The cause of this phenomenon is unknown, although it is assumed to be caused by WebGL being more lightweight compared to WebGPU.

Thus, requiring less binding of resources by default for one frame, whereas WebGPU requires an entire pipeline state to be bound. It stands to reason, therefore, that WebGPU has a larger upfront cost when rendering, causing smaller scenes with very few draw calls to be relatively inefficient compared to how WebGL does it. On the other hand, as soon as the scenes get more complex, WebGPU seems to benefit greatly from this greater upfront cost as it has much shorter CPU rendering times for the higher batch-count tests.

5.1.3 Validity and Reliability of Data

From the conducted paired t-tests done on the gathered data, it is valid to declare with high confidence that the results, by and large, are statistically significant. There are a few exceptions to this, most prevalent in the high 1% mean CPU values in the games experiments, or the Full-Screen Quads and Multiple Quads high 1% mean CPU synthetic tests. This finding is considered reasonable, as the number of samples for calculating the 1% high mean values is a fraction of the original sample count (20 in total per test). This inherently reduces the degree of certainty possible when conducting the t-tests. Despite this, on the GPU side, there is still always a significant difference in the results between the measured WebGPU and WebGL values.

One theory as to why the results are less significant on the CPU is that often very limited work happens per frame, with the majority of work being put on the GPU. As both Rasterizers are fast enough in these cases, the data yielded is close enough to not be of significant difference. An example where this reasoning does not hold well, however, can be seen in the case of rendering 100 multiple polygons in Table E.1. The resulting p-value for the `RenderBatches` function is 0.361 and the equivalent p-value for the `canvas_render_items` function is 0.161, both significantly higher than 0.005. However, as seen in the experiment of rendering 10 polygons, there is a significant difference. In that case, less work is put on the CPU than the 100 polygons test, which is a contradicting finding.

5.2 WebGPU Performance

There are several explanations as to why WebGPU performs better than WebGL at the rendering tasks presented in the thesis. Some are difficult to ascertain as they depend on both WebGPU implementation details and the particular graphics drivers being used on the machine it runs on. However, due to the drivers for modern graphics APIs being able to work closer to the hardware, it stands to reason that they can reach degrees of optimization unachievable by the WebGL or OpenGL ones.

The use of bundled state is seemingly also of great value to the WebGPU implementation as mentioned in section 5.1.2. It is, again, difficult to estimate exactly what is gained for WebGPU through this architectural choice, due to WebGL being, to an extent, opaque in its setting of GPU state. However, this difference between the two APIs provides a reasonable explanation for the results that have previously been presented.

Despite WebGPU showing consistently better frame times than WebGL, there are still times when it struggles. Most notably, the CPU frame time for the 50

000 full-screen quads synthetic test, where WebGPU suddenly shows a 104 times increase in frame time compared to the 40 000 full-screen quads variant. This can be further traced to the `canvas_render_items` function, where the same pattern shows up. However, none of the two measured scopes contained within the `canvas_render_items` function display the same behavior. Because the CPU frame time for this case is very similar to the GPU frame time, this suggests that within the `canvas_render_items` function some synchronization between the GPU and CPU takes place. The seemingly most reasonable explanation is the uploading of data to the instance buffer that takes place in between the `ConstructBatches` and `RenderBatches` scopes. It involves a call to the `WriteBuffer` function on the GPU queue and depending on how the WebGPU implementation handles this call, it is possible that this could force synchronization between the CPU and GPU, which consequently causes the CPU times to increase drastically due to the very long GPU frame times of the test.

On the other hand, GPU frame times are very long for other variants of the Full-screen Quads test. The reason synchronization would not need to take place for the other variants remains unknown and can most likely only be answered by studying the WebGPU implementation.

5.3 WebGL Performance

Aside from the already discussed performance benefits that are inherent to WebGPU as a modern technology, there exist other possible explanations as to why the performance of WebGL falls behind WebGPU in the conducted experiments.

WebGL seems to experience different kinds of stalling at different workloads, whereas WebGPU does not. Looking at Figure 4.21, WebGL experiences exceptionally high mean CPU frame times. As this particular test is very heavy on the GPU a plausible reason is that the CPU stalls as it is waiting on the WebGL GPU instructions. The claim is reinforced by the fact that WebGL is *several hundreds* of milliseconds slower than WebGPU in mean GPU frame times in the case of the larger workloads, as evident in Figure 4.11.

A possible example of the GPU being stalled instead can be seen in the Polygons tests. (Figure 4.13). As the workload increases from ten to 100, to 1000 polygons the GPU mean frame times shorten. This could be the case of the GPU being less and less bottlenecked by the CPU as it gets more work to perform. Looking at Figure 4.25, the claim is supported by the fact that the CPU mean frame times increase a bit up to and including 1000 polygons, at which point the work is more evenly spread across the CPU and GPU. From that point, the jump from 1000 to 10 000 polygons is big enough for the GPU to start struggling more.

5.4 Limitations

This section will present some limitations pertaining to the analysis of the results previously presented.

The first limitation concerns the method used for measuring elapsed time on the GPU. As stated in section 3.5 Experiment and Data Gathering, the two graphics

APIs differ in how elapsed time on the GPU is measured. Most importantly, due to how API calls are issued in WebGL—namely, there being no command queue structure available to the user—there is no guarantee that the GPU time elapsed will be affected by what takes place on the CPU between the start and end of a time elapsed query. This stands in stark contrast to how measurement is done in WebGPU. There, a timestamp is queued as a command within a packed command buffer that has been fully prepared before its submission to the GPU. Thereby making sure the only elapsed time being measured is the execution of the commands queued between the start and end timestamp.

Such a difference in timing would barely matter when the CPU only issues a few WebGL calls during a frame. However, for the test cases presented where there are many thousands of draw calls, there is no way of knowing when exactly the elapsed time query starts due to not being able to know when exactly the first draw call is submitted to the GPU.

This discrepancy in the measurement of elapsed GPU time is a direct consequence of WebGL forcing time elapsed to be measured in this exact way. There is no way of getting around the fact that GPU command submission is opaque to the user of the API.

Secondly, the Web limits its frame rate to that of the refresh rate of the monitor. This results in measurements of elapsed CPU time having to be done within a singular frame. Without this frame rate limit, measurements of frame time could be done across two frames to measure the exact time it takes to render an entire frame. However, due to the forced synchronization between frames, such a measurement would be meaningless. Instead, the rendering part of the engine code is measured. With this trade-off, there are some possible issues regarding what is being done outside of the rendering code of the engine. For instance, the Emscripten submission of frames to the browser takes place outside of the user's control and outside of the rendering code. Therefore, if WebGPU and WebGL are handled differently by Emscripten in some regards, the measurements presented in this thesis would not fairly represent this difference.

Chapter 6

Conclusions and Future Work

This paper has investigated the relative performance of two Rasterizers based on two different rendering APIs; WebGL and WebGPU. This was done by implementing a WebGPU Rasterizer backend and comparing it with the existing WebGL Rasterizer backend in the context of the Godot game engine. The work was chosen to highlight the performance capabilities of a modern web rendering technology through WebGPU when compared to the existing WebGL alternative. The work was grounded in both game examples with realistic workloads and raw stress tests of varying workloads through synthetic experiments.

The aim of the paper, as stated in the research question, was to determine the difference in CPU and GPU frame times between using a WebGPU and WebGL Rasterizer. To account for occasional spikes in frame time, aside from just presenting the overall mean CPU and GPU frame time results, the 1% high and 95% low average frame timings were also calculated and presented. Additionally, in order to gain a greater understanding of the CPU frame times, measurements were taken for a selection of the most important render functions and were presented as well.

For the implementation, careful consideration was put into making a backend WebGPU Rasterizer with an emphasis on minimizing unfairness with regard to architecture, utilized rendering techniques, and shader complexity. This was done to ensure the measurements gathered were relevant to the stated research question.

The results presented show how the WebGPU implementation, in its current state, consistently performs better than the WebGL equivalent. It does so across all conducted experiments in terms of total mean CPU and GPU frame time. The 1% high and 95% low averages show that the WebGPU implementation is also overall more stable in its frame times compared to WebGL, although there are some exceptions. Furthermore, and in general, the presented results are statistically significant.

For the profiled Rasterizer functions WebGPU is consistently slower for all conducted game experiments. In general, for tests consisting of a few batches with few draw calls, WebGPU performs worse than WebGL, only overtaking it as the number of batches increases. A possible cause of this behavior could be that WebGL is more lightweight in initial frame setup, but as complexity increases the bundled state approach of WebGPU works in its favor.

Overall, it can be concluded that the aim of determining the differences in CPU and GPU frame times between a WebGPU and WebGL Rasterizer was achieved. The results and findings of the thesis have broad implications for the future of web rendering technology, highlighting the potential advantages of WebGPU over the existing WebGL alternative.

6.1 Future Work

During the process of working on this thesis, a number of discoveries were made with regard to what could be further improved upon and researched. These discoveries are detailed in their own subsections, with *Optimizations* discussing how the implementation, in its current form, could be improved, and *Future Research* detailing possible endeavors that could be explored to expand upon the existing work.

6.1.1 Optimizations

The implementation of the WebGPU renderer is relatively naive in the sense that it blindly follows an architecture specifically made for APIs designed like OpenGL. Therefore, it stands to reason that even better results could be achieved if an architecture more suitable to the modern graphics API workflow was used for the implementation. An example of this would be to implement the renderer using the Godot Engine RenderDevice backend instead of the Rasterizer backend, introduced with Godot 4.0. However, with the current state of WebGPU and functionality like push constants being unsupported, an entirely separate rendering backend might be preferred.

Further optimizations come down to minimizing structures being constructed at runtime, such as the instance buffer, and trying to keep the rebinding of resources to a minimum. For this thesis, this has been attempted. However, it is possible that it could be further realized within a more accommodating architecture.

6.1.2 Future Research

A notable suggestion for future research is to investigate the GPU VRAM usage by both WebGL and WebGPU, if and when this feature eventually becomes available for WebGPU. Memory usage could play a large part in how effective an application runs as general memory pressure and cache misses has a certain cost associated with them. As WebGL and WebGPU both manage the memory "behind the scenes" in order to effectively handle multiple frames in flight, an analysis of the memory usage would add new insights into how the two technologies perform comparatively.

Another suggestion for future research is to build upon the work in this thesis in order to have the WebGPU Rasterizer more feature rich. This would mainly involve adding support for additional render item types and complementing the 2D Canvas Renderer with the 3D Scene Renderer. This would open up new performance comparisons to be made and new insights as to how the APIs:s compare. One such comparison could be to perform 3D shadow passes where only the vertex shader and depth buffer are bound to the render pipeline (omitting the fragment shader), which in modern rendering architectures tends to be well-optimized. Yet another comparison could be to compare anti-alias resolve queries of the two APIs:s, which are bound to be different in terms of performance. These are but two of many possible comparisons that could be made if more features are added to the WebGPU Rasterizer. A feature-complete WebGPU Rasterizer equivalent would "tell the whole story" and make it possible to generalize how the two APIs:s compare across all domains of the Godot rendering backend.

As this work compares WebGPU and WebGL in the environment of the Godot game engine, future research would benefit from replicating the experiment in other engines featuring a WebGL backend (Unity being a notable example). This would lead to new knowledge on how WebGPU as a modern web rendering solution scales over multiple products, each with its own set of rules and standards.

The work presented in this thesis is open source and free to replicate and improve upon.

References

[1] Aezart, "Snake," Itch.io. [Online]. Available: https://aezart.itch.io/checkers

[2] A. Aldahir, "Evaluation of the performance of webGPU in a cluster of web-browsers for scientific computing," Bachelor's thesis, Umeå University, 2022. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-197058

[3] angelchama333, "Falling Cats," Itch.io. [Online]. Available: https://angelchama333.itch.io/falling-cats

[4] "Metal Overview," Apple Inc. [Online]. Available: https://developer.apple.com/metal/

[5] ceruleancerise, "Ponder," Itch.io. [Online]. Available: https://ceruleancerise.itch.io/ponder

[6] R. Dunlop, "FPS Versus Frame Time." [Online]. Available: http://www.mvps.org/directx/articles/fps_versus_frame_time.htm

[7] L. Dyken, P. Poudel, W. Usher, S. Petruzza, J. Y. Chen, and S. Kumar, "Graphwagu: Gpu powered large scale graph layout computation and rendering for the web," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2022. [Online]. Available: https://diglib.eg.org/xmlui/bitstream/handle/10.2312/pgv20221067/073-083.pdf?sequence=1

[8] "Emscripten documentation," Emscripten. [Online]. Available: https://emscripten.org/

[9] 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, 2014.

[10] O. Ferraz, P. Menezes, V. Silva, and G. Falcao, "Benchmarking Vulkan vs OpenGL Rendering on Low-Power Edge GPUs," in *2021 International Conference on Graphics and Interaction (ICGI)*, 2021, pp. 1–8.

[11] "Godot 4.0 sets sail: All aboard for new horizons," Godot. [Online]. Available: https://godotengine.org/article/godot-4-0-sets-sail/

[12] P. Hex, "Snake in Godot4," Itch.io. [Online]. Available: https://hexblit.itch.io/snake-in-godot4

[13] M. Hidaka, Y. Kikura, Y. Ushiku, and T. Harada, "WebDNN: Fastest DNN execution framework on web browser," in *MM 2017 - Proceedings of the 2017 ACM Multimedia Conference*, 2017, pp. 1213–1216.

[14] R. K. and W. C., "WebGL + WebGPU Meetup July 12, 2022," The Khronos Group, 2022. [Online]. Available: https://www.khronos.org/assets/uploads/developers/presentations/WebGL__WebGPU_Updates_Jul_22.pdf

[15] "OpenGL - The Industry Standard for High Performance Graphics," Khronos Group. [Online]. Available: https://www.opengl.org/

[16] D. Liu, J. Peng, Y. Wang, M. Huang, Q. He, Y. Yan, B. Ma, C. Yue, and Y. Xie, "Implementation of interactive three-dimensional visualization of air pollutants using WebGL," *Environmental Modelling & Software*, vol. 114, pp. 188–194, Apr. 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1364815218304195

[17] M. Lujan, M. Baum, D. Chen, and Z. Zong, "Evaluating the Performance and Energy Efficiency of OpenGL and Vulkan on a Graphics Rendering Server," in *2019 International Conference on Computing, Networking and Communications (ICNC)*, 2019, pp. 777–781.

[18] "Direct3D - Win32 apps," Microsoft, Sep. 2021. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3d

[19] MohamedA.G, "Evader," Itch.io. [Online]. Available: https://mohamedag.itch.io/evader

[20] A. Rossberg, "WebAssembly Core Specification," Apr. 2022. [Online]. Available: https://www.w3.org/TR/wasm-core-2/

[21] J. Shiraef, "An exploratory study of high performance graphics application programming interfaces," Master's thesis, University of Chattanoga, 2016. [Online]. Available: https://scholar.utc.edu/theses/446

[22] ShoeFisherGames, "Deck Before Dawn," Itch.io. [Online]. Available: https://shoefishergames.itch.io/deck-before-dawn

[23] U. Technologies, "Best practices for profiling game performance | Unity." [Online]. Available: https://unity.com/how-to/best-practices-for-profiling-game-performance

[24] W. Usher and V. Pascucci, "Interactive Visualization of Terascale Data in the Browser: Fact or Fiction?" in *2020 IEEE 10th Symposium on Large Data Analysis and Visualization (LDAV)*, 2020, pp. 27–36.

[25] Vulkan, "Vulkan Cross platform 3D Graphics," Khronos Group. [Online]. Available: https://www.vulkan.org/

[26] "High Resolution Time Level 2," W3C, 2019. [Online]. Available: https://www.w3.org/TR/hr-time-2/#clock-resolution

[27] "WebGPU," W3C, 2021. [Online]. Available: https://www.w3.org/TR/2021/WD-webgpu-20210518/

Appendix A

Shader Listings

```
#include "stdlib_inc.glsl"

layout(location = 6) in highp vec4 attrib_A;
layout(location = 7) in highp vec4 attrib_B;
layout(location = 8) in highp vec4 attrib_C;
layout(location = 9) in highp vec4 attrib_D;
layout(location = 10) in highp vec4 attrib_E;
layout(location = 11) in highp vec4 attrib_F;
layout(location = 12) in highp uvec4 attrib_G;
layout(location = 13) in highp uvec4 attrib_H;

#define read_draw_data_world_x attrib_A.xy
#define read_draw_data_world_y attrib_A.zw
#define read_draw_data_world_ofs attrib_B.xy
#define read_draw_data_color_texture_pixel_size attrib_B.zw

#define read_draw_data_modulation attrib_C
#define read_draw_data_ninepatch_margins attrib_D
#define read_draw_data_dst_rect attrib_E
#define read_draw_data_src_rect attrib_F
#define read_draw_data_flags attrib_G.z
#define read_draw_data_specular_shininess attrib_G.w
#define read_draw_data_lights attrib_H

flat out vec4 varying_A;
flat out vec2 varying_B;
flat out vec4 varying_C;
flat out vec4 varying_E;
flat out uvec2 varying_F;
flat out uvec4 varying_G;

#include "canvas_uniforms_inc.glsl"

out vec2 uv_interp;
out vec4 color_interp;
out vec2 vertex_interp;

void main() {
  varying_A = vec4(read_draw_data_world_x, read_draw_data_world_y);
  varying_B = read_draw_data_color_texture_pixel_size;
  varying_C = read_draw_data_ninepatch_margins;
  varying_E = read_draw_data_src_rect;
  varying_F = uvec2(read_draw_data_flags, read_draw_data_specular_shininess);
  varying_G = read_draw_data_lights;

  vec2 vertex;
  vec4 color;
  vec2 uv;

  vec2 vertex_base_arr[6] = vec2[](vec2(0.0, 0.0), vec2(0.0, 1.0), vec2(1.0, 1.0), vec2(1.0, 0.0), vec2(0.0,
     0.0), vec2(1.0, 1.0));
  vec2 vertex_base = vertex_base_arr[gl_VertexID % 6];

  uv = read_draw_data_src_rect.xy + abs(read_draw_data_src_rect.zw) * ((read_draw_data_flags &
     FLAGS_TRANSPOSE_RECT) != uint(0) ? vertex_base.yx : vertex_base.xy);
  color = read_draw_data_modulation;
  vertex = read_draw_data_dst_rect.xy + abs(read_draw_data_dst_rect.zw) * mix(vertex_base, vec2(1.0, 1.0) -
     vertex_base, lessThan(read_draw_data_src_rect.zw, vec2(0.0, 0.0)));

  mat4 model_matrix = mat4(vec4(read_draw_data_world_x, 0.0, 0.0), vec4(read_draw_data_world_y, 0.0, 0.0), vec4
     (0.0, 0.0, 1.0, 0.0), vec4(read_draw_data_world_ofs, 0.0, 1.0));

  vertex = (model_matrix * vec4(vertex, 0.0, 1.0)).xy;
  color_interp = color;
  vertex = (canvas_transform * vec4(vertex, 0.0, 1.0)).xy;
  vertex_interp = vertex;
  uv_interp = uv;
  gl_Position = screen_transform * vec4(vertex, 0.0, 1.0);
}
```

Figure A.1: Listing of the canvas glsl-vertex shader used by the WebGL Rasterizer for rendering quads.

```
#[fragment]

#include "canvas_uniforms_inc.glsl"
#include "stdlib_inc.glsl"

in vec2 uv_interp;
in vec2 vertex_interp;
in vec4 color_interp;

// Can all be flat as they are the same for the whole batched instance
flat in vec4 varying_A;
flat in vec2 varying_B;
#define read_draw_data_world_x varying_A.xy
#define read_draw_data_world_y varying_A.zw
#define read_draw_data_color_texture_pixel_size varying_B

flat in vec4 varying_C;
#define read_draw_data_ninepatch_margins varying_C

flat in vec4 varying_E;
#define read_draw_data_src_rect varying_E

flat in uvec2 varying_F;
flat in uvec4 varying_G;
#define read_draw_data_flags varying_F.x
#define read_draw_data_specular_shininess varying_F.y
#define read_draw_data_lights varying_G

uniform sampler2D color_buffer; //texunit:-4
uniform sampler2D sdf_texture; //texunit:-5
uniform sampler2D normal_texture; //texunit:-6
uniform sampler2D specular_texture; //texunit:-7

uniform sampler2D color_texture; //texunit:0

layout(location = 0) out vec4 frag_color;

void main() {
  vec4 color = color_interp;
  vec2 uv = uv_interp;
  color *= texture(color_texture, uv);
  frag_color = color;
}
```

Figure A.2: Listing of the canvas glsl-fragment shader used by the WebGL Rasterizer for rendering quads.

```
#[vertex]

layout(location = 0) in vec2 vertex_attrib;
layout(location = 3) in vec4 color_attrib;
layout(location = 4) in vec2 uv_attrib;

#include "stdlib_inc.glsl"

layout(location = 6) in highp vec4 attrib_A;
layout(location = 7) in highp vec4 attrib_B;
layout(location = 8) in highp vec4 attrib_C;
layout(location = 9) in highp vec4 attrib_D;
layout(location = 10) in highp vec4 attrib_E;
layout(location = 11) in highp vec4 attrib_F;
layout(location = 12) in highp uvec4 attrib_G;
layout(location = 13) in highp uvec4 attrib_H;

#define read_draw_data_world_x attrib_A.xy
#define read_draw_data_world_y attrib_A.zw
#define read_draw_data_world_ofs attrib_B.xy
#define read_draw_data_color_texture_pixel_size attrib_B.zw
#define read_draw_data_modulation attrib_C
#define read_draw_data_ninepatch_margins attrib_D
#define read_draw_data_dst_rect attrib_E
#define read_draw_data_src_rect attrib_F
#define read_draw_data_flags attrib_G.z
#define read_draw_data_specular_shininess attrib_G.w
#define read_draw_data_lights attrib_H

flat out vec4 varying_A;
flat out vec2 varying_B;
flat out vec4 varying_C;
flat out uvec2 varying_F;
flat out uvec4 varying_G;

#include "canvas_uniforms_inc.glsl"

out vec2 uv_interp;
out vec4 color_interp;
out vec2 vertex_interp;

void main() {
  varying_A = vec4(read_draw_data_world_x, read_draw_data_world_y);
  varying_B = read_draw_data_color_texture_pixel_size;
  varying_C = read_draw_data_ninepatch_margins;
  varying_F = uvec2(read_draw_data_flags, read_draw_data_specular_shininess);
  varying_G = read_draw_data_lights;

  vec2 vertex;
  vec4 color;
  vec2 uv;

  vertex = vertex_attrib;
  color = color_attrib * read_draw_data_modulation;
  uv = uv_attrib;

  mat4 model_matrix = mat4(vec4(read_draw_data_world_x, 0.0, 0.0), vec4(read_draw_data_world_y, 0.0, 0.0), vec4
     (0.0, 0.0, 1.0, 0.0), vec4(read_draw_data_world_ofs, 0.0, 1.0));

  vertex = (model_matrix * vec4(vertex, 0.0, 1.0)).xy;
  color_interp = color;
  vertex = (canvas_transform * vec4(vertex, 0.0, 1.0)).xy;
  vertex_interp = vertex;
  uv_interp = uv;
  gl_Position = screen_transform * vec4(vertex, 0.0, 1.0);
}
```

Figure A.3: Listing of the canvas glsl-vertex shader used by the WebGL Rasterizer for rendering polygons.

```
#[fragment]

#include "canvas_uniforms_inc.glsl"
#include "stdlib_inc.glsl"

in vec2 uv_interp;
in vec2 vertex_interp;
in vec4 color_interp;

flat in vec4 varying_A;
flat in vec2 varying_B;
#define read_draw_data_world_x varying_A.xy
#define read_draw_data_world_y varying_A.zw
#define read_draw_data_color_texture_pixel_size varying_B
flat in vec4 varying_C;
#define read_draw_data_ninepatch_margins varying_C

flat in uvec2 varying_F;
flat in uvec4 varying_G;
#define read_draw_data_flags varying_F.x
#define read_draw_data_specular_shininess varying_F.y
#define read_draw_data_lights varying_G

uniform sampler2D color_buffer; //texunit:-4
uniform sampler2D sdf_texture; //texunit:-5
uniform sampler2D normal_texture; //texunit:-6
uniform sampler2D specular_texture; //texunit:-7
uniform sampler2D color_texture; //texunit:0

layout(location = 0) out vec4 frag_color;

void main() {
  vec4 color = color_interp;
  vec2 uv = uv_interp;
  color *= texture(color_texture, uv);
  frag_color = color;
}
```

Figure A.4: Listing of the canvas glsl-fragment shader used by the WebGL Rasterizer for rendering polygons.

```
const FLAGS_TRANSPOSE_RECT: u32 = (1 << 10);
const FLAGS_DEFAULT_NORMAL_MAP_USED: u32 = (1 << 26);
const FLAGS_DEFAULT_SPECULAR_MAP_USED: u32 = (1 << 27);
struct CanvasData {
  CanvasTransform: mat4x4<f32>,
  ScreenTransform: mat4x4<f32>,
  CanvasNormalTransform: mat4x4<f32>,
  CanvasModulation: vec4<f32>,
  ScreenPixelSize: vec2<f32>,
  Time: f32,
  UsePixelSnap: u32,
  SdfToTex: vec4<f32>,
  ScreenToSdf: vec2<f32>,
  SdfToScreen: vec2<f32>,
  DirectionalLightCount: u32,
  TexToSdf: f32,
  Padding1: u32,
  Padding2: u32
  }
struct Miscellaneous {
  PlaceHolder1: u32,
  PlaceHolder2: u32,
  DataFlags: u32,
  SpecularShininess: u32
  }
struct PerInstanceData {
  DstRect: vec4<f32>,
  SrcRect: vec4<f32>,
  ModulationColor: vec4<f32>,
  Misc: Miscellaneous,
  WorldPos: vec4<f32>,
  WorldOffset: vec2<f32>,
  TexelSize: vec2<f32> ,
  Lights: vec4<u32>,
  MsdfOrNinepatchMargins: vec4<f32>
  }
struct VertexOutput {
  @builtin(position) outPositionCS: vec4<f32>,
  @location(0) color: vec4<f32>,
  @location(1) tex: vec2<f32>,
  @location(2) @interpolate(flat) WorldXY: vec4<f32>,
  @location(3) @interpolate(flat) TexelSize: vec2<f32>,
  @location(4) @interpolate(flat) MsdfOrNinepatchMargins: vec4<f32>,
  @location(5) @interpolate(flat) SrcRect: vec4<f32>,
  @location(6) @interpolate(flat) FlagsAndSpecularShininess: vec2<u32>,
  @location(7) @interpolate(flat) Lights: vec4<u32>
}
@group(0) @binding(2) var<uniform> canvasData : CanvasData;
@group(0) @binding(3) var<storage,read> perInstanceData : array<PerInstanceData>;

@vertex
fn main(@builtin(vertex_index) vertexID : u32, @builtin(instance_index) instanceID : u32) -> VertexOutput {
  var out: VertexOutput;
  out.WorldXY = perInstanceData[instanceID].WorldPos;
  out.TexelSize = perInstanceData[instanceID].TexelSize;
  out.MsdfOrNinepatchMargins = perInstanceData[instanceID].MsdfOrNinepatchMargins;
  out.SrcRect = perInstanceData[instanceID].SrcRect;
  out.FlagsAndSpecularShininess = vec2<u32>(perInstanceData[instanceID].Misc.DataFlags, perInstanceData[
      instanceID].Misc.SpecularShininess);
  out.Lights = perInstanceData[instanceID].Lights;

  var vertex  : vec2<f32>;
  var color : vec4<f32>;
  var uv    : vec2<f32>;

  var vertexBaseArray = array<vec2<f32>, 6>(vec2<f32>(0.0, 0.0), vec2<f32>(0.0, 1.0), vec2<f32>(1.0, 1.0), vec2<
      f32>(1.0, 0.0), vec2<f32>(0.0, 0.0), vec2<f32>(1.0, 1.0));
  var vertexBase = vertexBaseArray[vertexID % 6];
  var mixValue = vec2<f32>(0.0, 0.0);
  if (perInstanceData[instanceID].SrcRect.z < 0.0)
      mixValue.x = 1;
  if (perInstanceData[instanceID].SrcRect.w < 0.0)
      mixValue.y = 1;

  vertex = perInstanceData[instanceID].DstRect.xy + abs(perInstanceData[instanceID].DstRect.zw) * mix(vertexBase
      , vec2<f32>(1.0, 1.0) - vertexBase, mixValue);

  if ((perInstanceData[instanceID].Misc.DataFlags & FLAGS_TRANSPOSE_RECT) != 0) {
    uv = perInstanceData[instanceID].SrcRect.xy + abs(perInstanceData[instanceID].SrcRect.zw) * vertexBase.yx;
  }
  else{
    uv = perInstanceData[instanceID].SrcRect.xy + abs(perInstanceData[instanceID].SrcRect.zw) * (vertexBase.xy *
       vec2<f32>(1.0, -1.0) + vec2<f32>(0.0, 1.0));
  }
  var modelMatrix = mat4x4<f32>(vec4<f32>(perInstanceData[instanceID].WorldPos.xy, 0.0, 0.0), vec4<f32>(
      perInstanceData[instanceID].WorldPos.zw, 0.0, 0.0), vec4<f32>(0.0, 0.0, 1.0, 0.0), vec4<f32>(
      perInstanceData[instanceID].WorldOffset.xy, 0.0, 1.0));
  vertex = (modelMatrix * vec4<f32>(vertex, 0.0, 1.0)).xy;
  vertex = (canvasData.CanvasTransform * vec4<f32>(vertex, 0.0, 1.0)).xy;
  vertex = (canvasData.ScreenTransform * vec4<f32>(vertex, 0.0, 1.0)).xy;
  color = perInstanceData[instanceID].ModulationColor;
  out.outPositionCS = vec4<f32>(vertex, 0.0, 1.0);
  out.color = color;
  out.tex = uv;
  return out;
}
```

Figure A.5: Listing of the canvas wgsl-vertex shader used by the WebGPU Rasterizer for rendering quads.

```
const FLAGS_TRANSPOSE_RECT: u32 = (1 << 10);
const FLAGS_DEFAULT_NORMAL_MAP_USED: u32 = (1 << 26);
const FLAGS_DEFAULT_SPECULAR_MAP_USED: u32 = (1 << 27);

struct FragInput {
  @location(0) inColor: vec4<f32>,
  @location(1) tex: vec2<f32>,
  @location(2) @interpolate(flat) WorldXY: vec4<f32>,
  @location(3) @interpolate(flat) TexelSize: vec2<f32>,
  @location(4) @interpolate(flat) MsdfOrNinepatchMargins: vec4<f32>,
  @location(5) @interpolate(flat) SrcRect: vec4<f32>,
  @location(6) @interpolate(flat) FlagsAndSpecularShininess: vec2<u32>,
  @location(7) @interpolate(flat) Lights: vec4<u32>
}

@group(0) @binding(0) var bilinearSampler : sampler;
@group(0) @binding(1) var diffuseTexture : texture_2d<f32>;
@group(0) @binding(4) var normalMapTexture : texture_2d<f32>;
@group(0) @binding(5) var specularMapTexture : texture_2d<f32>;

@fragment
fn main(input: FragInput) -> @location(0) vec4<f32> {
  var color = input.inColor;
  var uv = input.tex;
  color *= textureSample(diffuseTexture, bilinearSampler, vec2f(input.tex.x, input.tex.y));
  return color;
}
```

Figure A.6: Listing of the canvas wgsl-fragment shader used by the WebGPU Rasterizer for rendering quads.

```
const FLAGS_TRANSPOSE_RECT: u32 = (1 << 10);
const FLAGS_DEFAULT_NORMAL_MAP_USED: u32 = (1 << 26);
const FLAGS_DEFAULT_SPECULAR_MAP_USED: u32 = (1 << 27);
struct CanvasData {
  CanvasTransform: mat4x4<f32>,
  ScreenTransform: mat4x4<f32>,
  CanvasNormalTransform: mat4x4<f32>,
  CanvasModulation: vec4<f32>,
  ScreenPixelSize: vec2<f32>,
  Time: f32,
  UsePixelSnap: u32,
  SdfToTex: vec4<f32>,
  ScreenToSdf: vec2<f32>,
  SdfToScreen: vec2<f32>,
  DirectionalLightCount: u32,
  TexToSdf: f32,
  Padding1: u32,
  Padding2: u32
 }
struct Miscellaneous {
  PlaceHolder1: u32,
  PlaceHolder2: u32,
  DataFlags: u32,
  SpecularShininess: u32
}
struct PerInstanceData {
  DstRect: vec4<f32>,
  SrcRect: vec4<f32>,
  ModulationColor: vec4<f32>,
  Misc: Miscellaneous,
  WorldPos: vec4<f32>,
  WorldOffset: vec2<f32>,
  TexelSize: vec2<f32>,
  Lights: vec4<u32>,
  MsdfOrNinepatchMargins: vec4<f32>
}
struct VertexInput {
  @location(0) inPositionLS:  vec2<f32>,
  @location(1) inColor:   vec4<f32>,
  @location(2) inTexCoords: vec2<f32>
}
struct VertexOutput {
  @builtin(position) outPositionCS: vec4<f32>,
  @location(0) color: vec4<f32>,
  @location(1) tex: vec2<f32>,
  @location(2) @interpolate(flat) WorldXY: vec4<f32>,
  @location(3) @interpolate(flat) TexelSize: vec2<f32>,
  @location(4) @interpolate(flat) MsdfOrNinepatchMargins: vec4<f32>,
  @location(5) @interpolate(flat) FlagsAndSpecularShininess: vec2<u32>,
  @location(6) @interpolate(flat) Lights: vec4<u32>
}

@group(0) @binding(2) var<uniform> canvasData : CanvasData;
@group(0) @binding(3) var<storage, read> perInstanceData : array<PerInstanceData>;

@vertex
fn main(vsIn : VertexInput, @builtin(instance_index) instanceID : u32) -> VertexOutput {
  var out: VertexOutput;

  out.WorldXY = perInstanceData[instanceID].WorldPos;
  out.TexelSize = perInstanceData[instanceID].TexelSize;
  out.MsdfOrNinepatchMargins = perInstanceData[instanceID].MsdfOrNinepatchMargins;
  out.FlagsAndSpecularShininess = vec2<u32>(perInstanceData[instanceID].Misc.DataFlags, perInstanceData[
      instanceID].Misc.SpecularShininess);
  out.Lights = perInstanceData[instanceID].Lights;

  var vertex  : vec2<f32>;
  var color : vec4<f32>;
  var uv    : vec2<f32>;
  vertex = vsIn.inPositionLS;
  uv = vsIn.inTexCoords;
  color = vsIn.inColor * perInstanceData[instanceID].ModulationColor;

  var modelMatrix = mat4x4<f32>(vec4<f32>(perInstanceData[instanceID].WorldPos.xy, 0.0, 0.0), vec4<f32>(
      perInstanceData[instanceID].WorldPos.zw, 0.0, 0.0), vec4<f32>(0.0, 0.0, 1.0, 0.0), vec4<f32>(
      perInstanceData[instanceID].WorldOffset.xy, 0.0, 1.0));

  vertex = (modelMatrix * vec4<f32>(vertex, 0.0, 1.0)).xy;
  vertex = (canvasData.CanvasTransform * vec4<f32>(vertex, 0.0, 1.0)).xy;
  vertex = (canvasData.ScreenTransform * vec4<f32>(vertex, 0.0, 1.0)).xy;
  out.outPositionCS = vec4<f32>(vertex, 0.0, 1.0);
  out.color = color;
  out.tex = uv;
  return out;
}
```

Figure A.7: Listing of the canvas wgsl-vertex shader used by the WebGPU Rasterizer for rendering polygons.

```
const FLAGS_TRANSPOSE_RECT: u32 = (1 << 10);
const FLAGS_DEFAULT_NORMAL_MAP_USED: u32 = (1 << 26);
const FLAGS_DEFAULT_SPECULAR_MAP_USED: u32 = (1 << 27);

struct FragInput {
  @location(0) inColor: vec4<f32>,
  @location(1) tex: vec2<f32>,
  @location(2) @interpolate(flat) WorldXY: vec4<f32>,
  @location(3) @interpolate(flat) TexelSize: vec2<f32>,
  @location(4) @interpolate(flat) MsdfOrNinepatchMargins: vec4<f32>,
  @location(5) @interpolate(flat) FlagsAndSpecularShininess: vec2<u32>,
  @location(6) @interpolate(flat) Lights: vec4<u32>
}

@group(0) @binding(0) var bilinearSampler : sampler;
@group(0) @binding(1) var diffuseTexture : texture_2d<f32>;
@group(0) @binding(4) var normalMapTexture : texture_2d<f32>;
@group(0) @binding(5) var specularMapTexture : texture_2d<f32>;

@fragment
fn main(input: FragInput) -> @location(0) vec4<f32> {
  var color = input.inColor;
  var uv = input.tex;
  color *= textureSample(diffuseTexture, bilinearSampler, vec2f(input.tex.x, input.tex.y));
  return color;
}
```

Figure A.8: Listing of the canvas wgsl-fragment shader used by the WebGPU Rasterizer for rendering polygons.

Appendix B

Game Footage

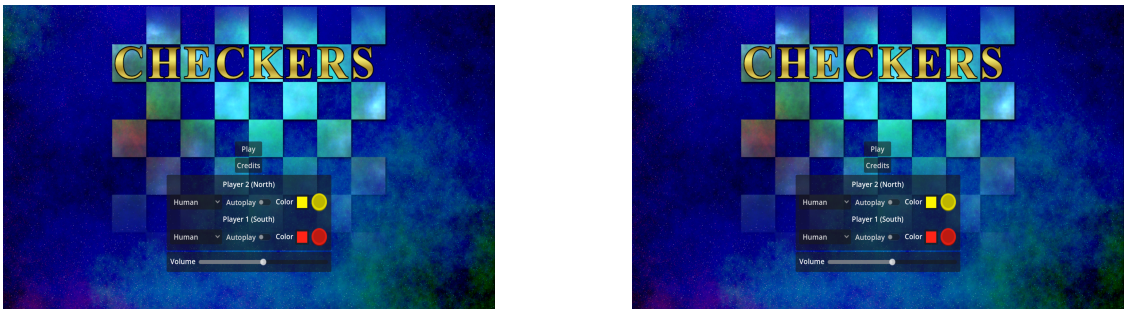

Figure B.1: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features the main menu from the game Checkers.

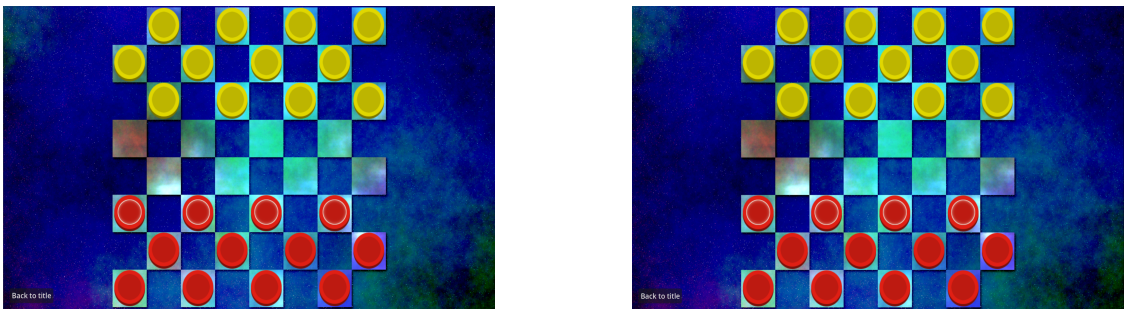

Figure B.2: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features gameplay from the game Checkers.

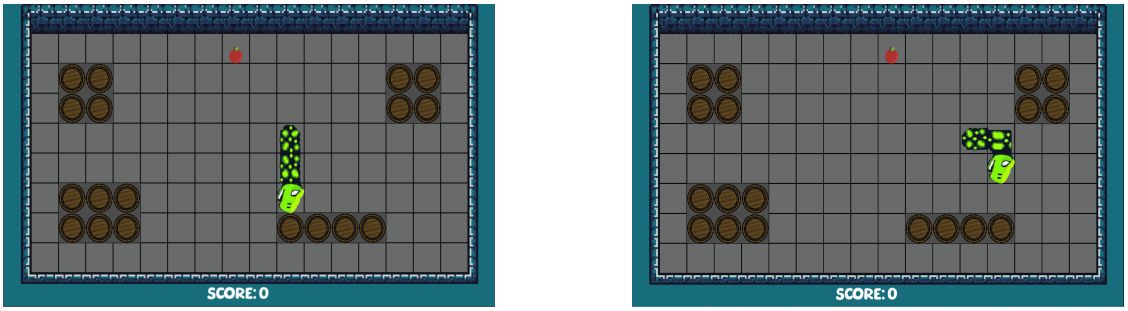

Figure B.3: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features gameplay from the game Snake.

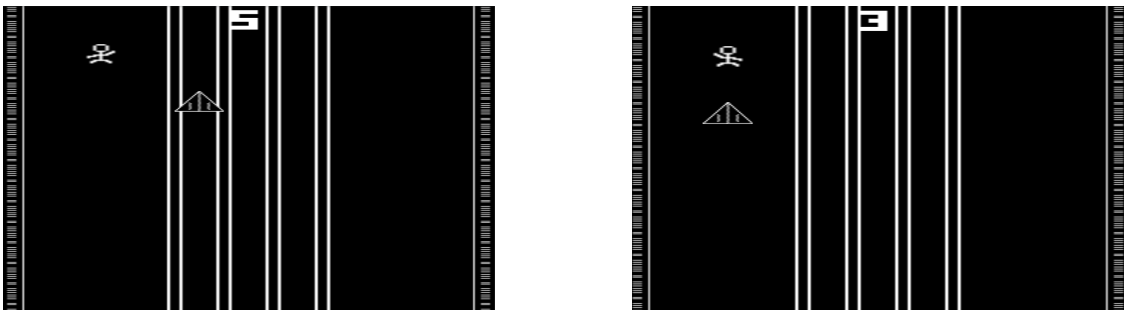

Figure B.4: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features gameplay from the game Evader.

Figure B.5: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features gameplay from the game Falling Cats.

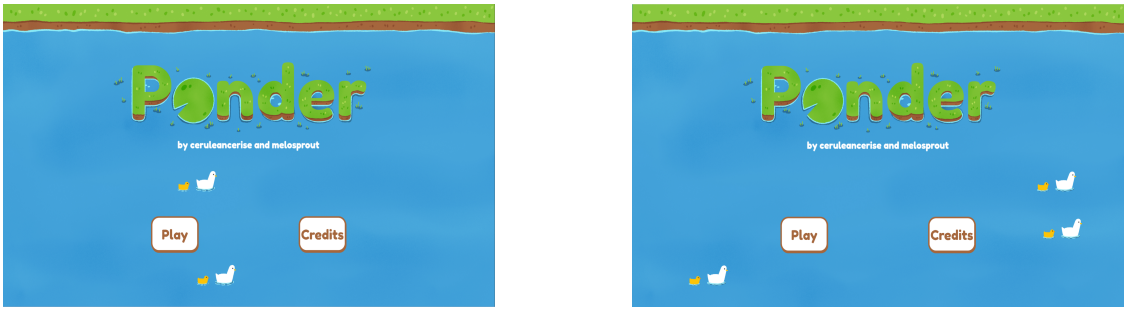

Figure B.6: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features the main menu from the game Ponder.

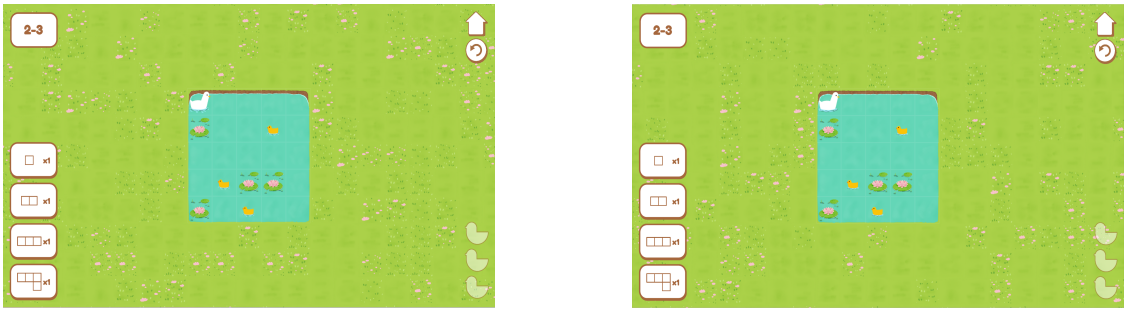

Figure B.7: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features gameplay from the game Ponder.

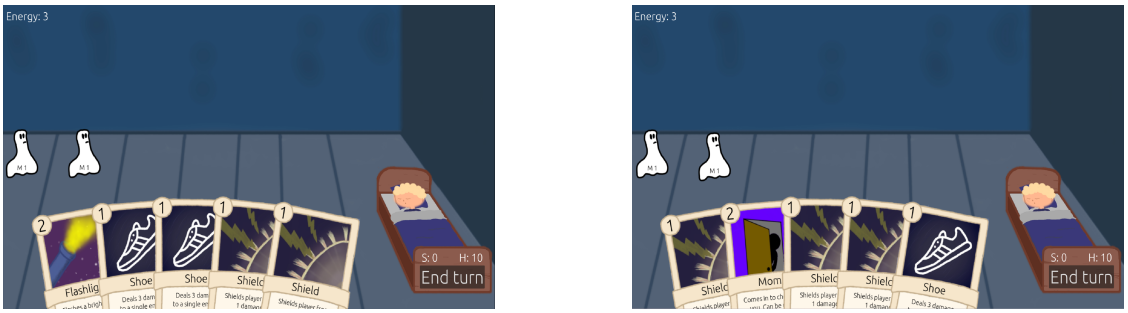

Figure B.8: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features gameplay from the game Deck Before Dawn.

Appendix C

Synthetic Tests Footage

Figure C.1: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features a synthetic test of rendering 30000 polygons, each composed of 360 vertices.

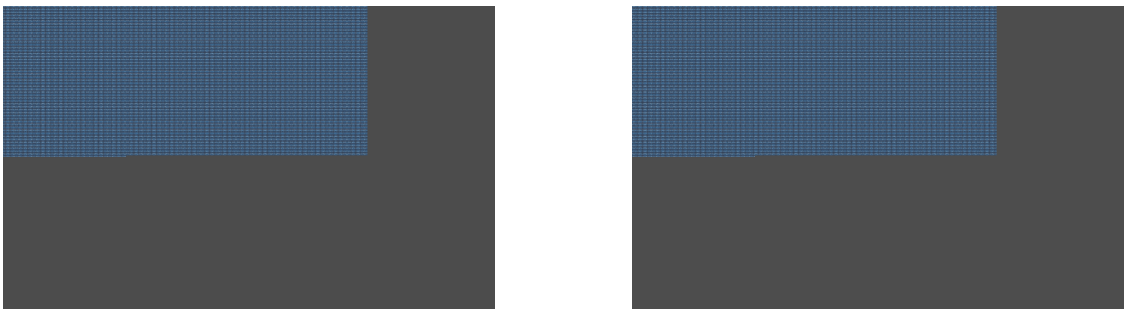

Figure C.2: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features a synthetic test of rendering 30000 textured sprites (quads).

Figure C.3: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features a synthetic test of rendering 30000 layered full-screen textured sprites (quads).

Figure C.4: A side-by-side comparison of the rasterized output between the WebGPU Rasterizer (left) and the WebGL Rasterizer (right). The footage features a synthetic test of rendering 20 layered polygons, each with 50000 vertices (1 million in total).

Appendix D

Graphs For Measurements

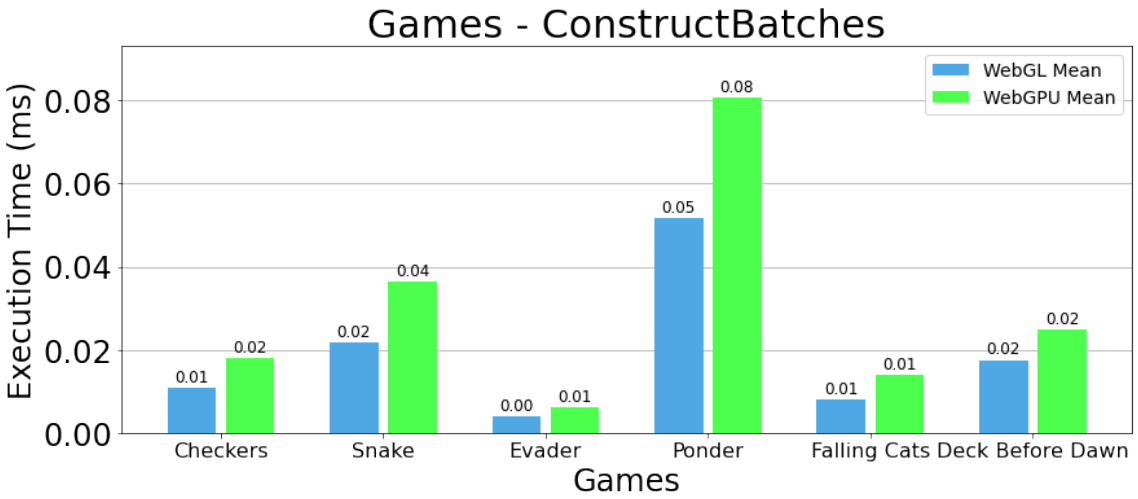

Figure D.1: Comparison of the mean WebGL and WebGPU ConstructBatches frame times, in milliseconds, for the various games. Lower is better.

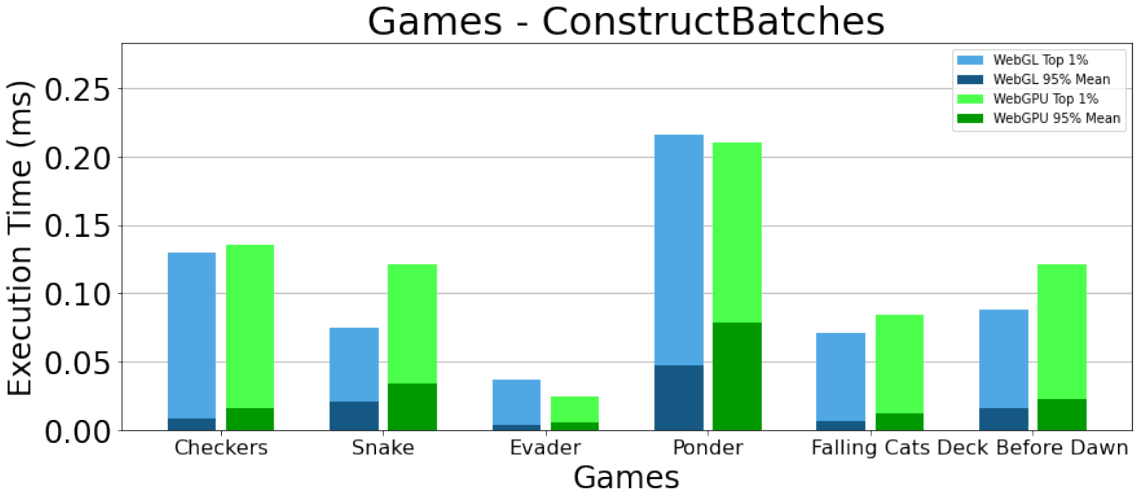

Figure D.2: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU ConstructBatches frame times, in milliseconds, for the various games. Lower is better.

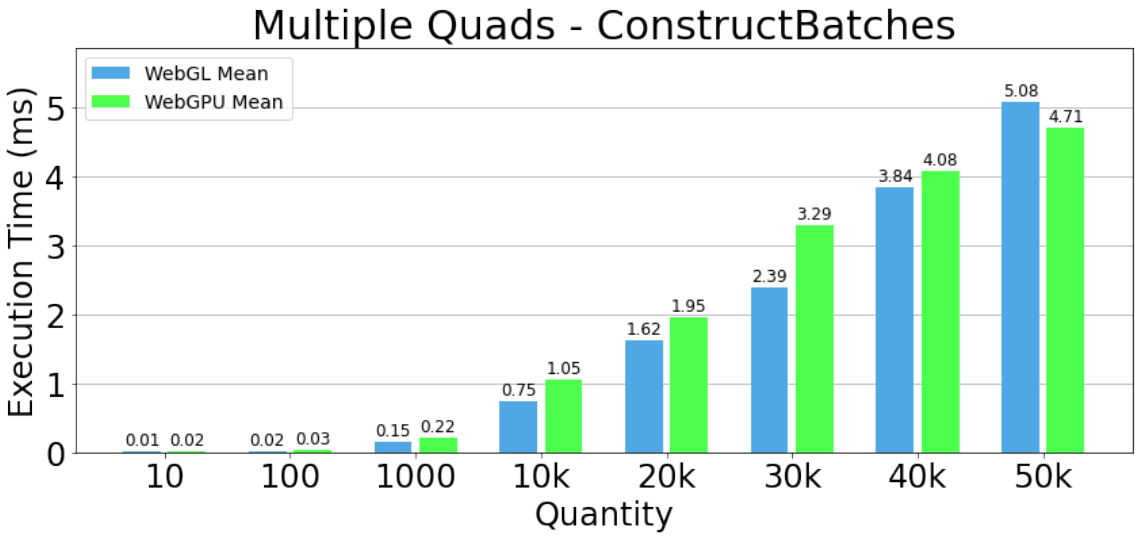

Figure D.3: Comparison of the mean WebGL and WebGPU ConstructBatches frame times, in milliseconds, for the Multiple Quads test. Lower is better. The workloads range from 10 to 50.000 quads.

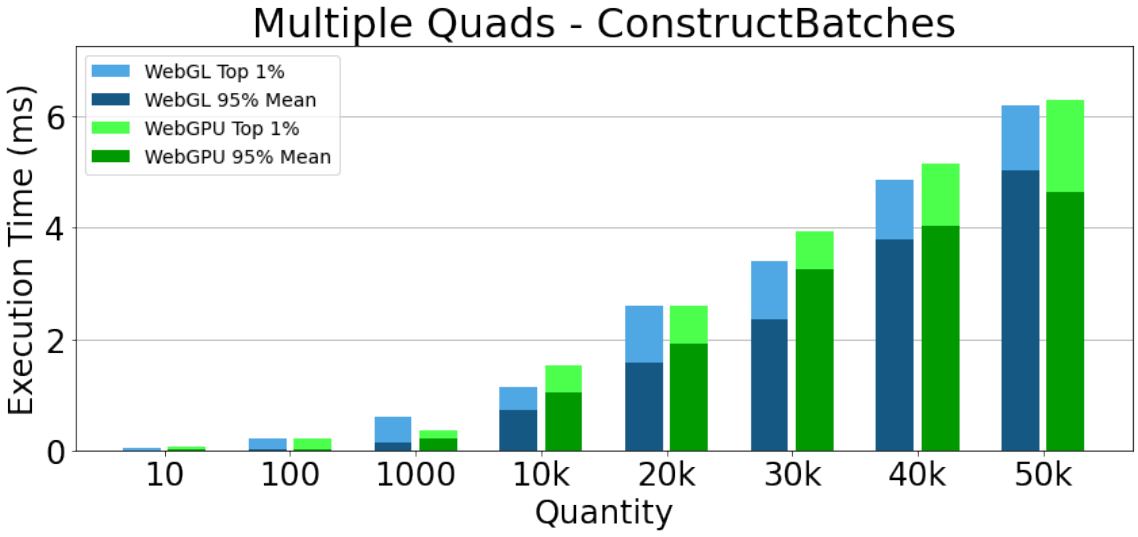

Figure D.4: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU ConstructBatches frame times, in milliseconds, for the Multiple Quads test. Lower is better. The workloads range from 10 to 50.000 quads.

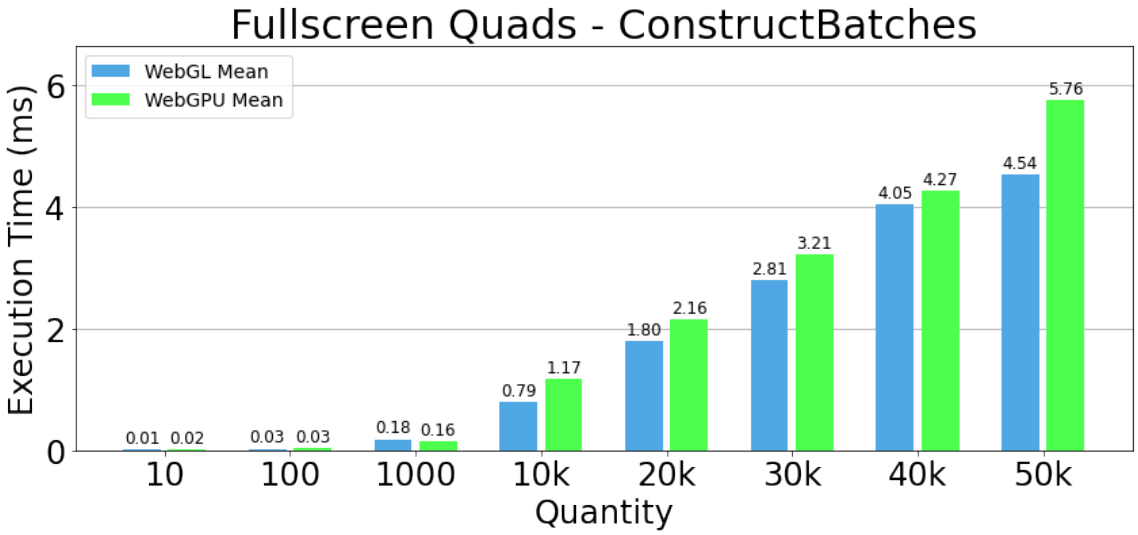

Figure D.5: Comparison of the mean WebGL and WebGPU ConstructBatches frame times, in milliseconds, for the Full-screen Quads test. Lower is better. The workloads range from 10 to 50.000 full-screen quads.

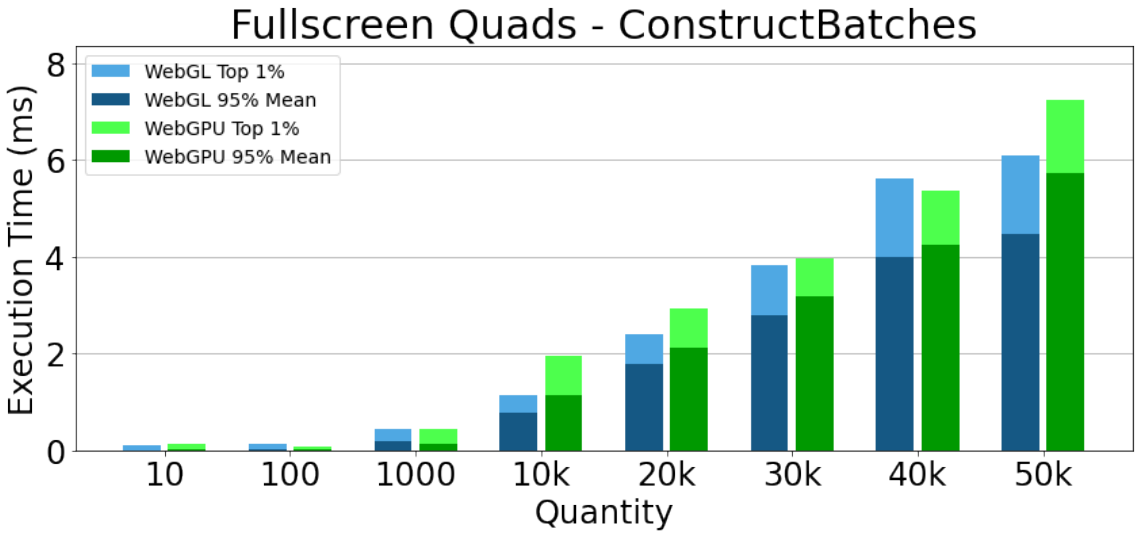

Figure D.6: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU ConstructBatches frame times, in milliseconds, for the Full-screen Quads test. Lower is better. The workloads range from 10 to 50.000 full-screen quads.

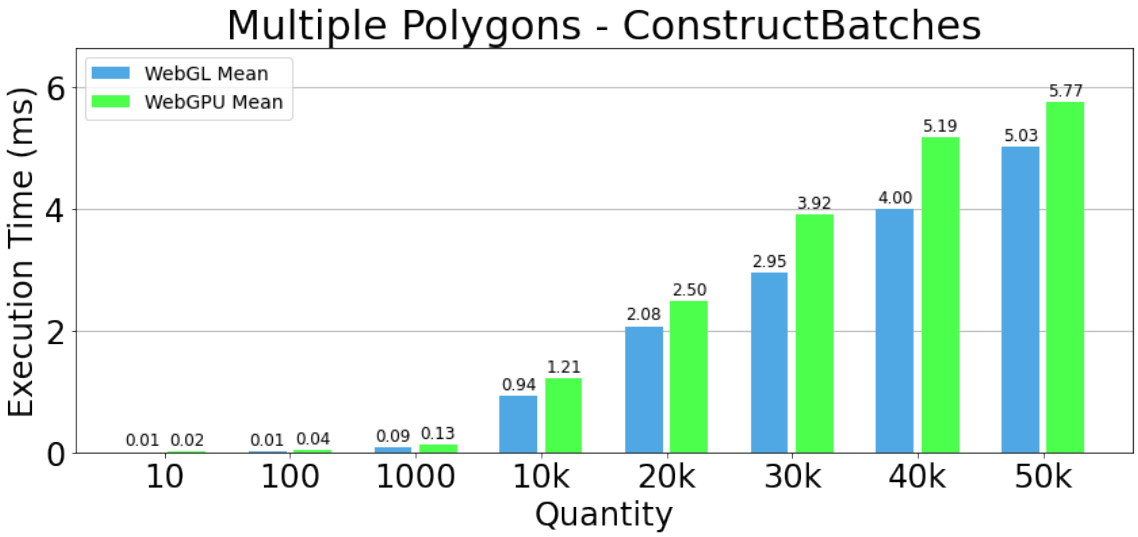

Figure D.7: Comparison of the mean WebGL and WebGPU ConstructBatches frame times, in milliseconds, for the Multiple Polygons test. Lower is better. The workloads range from 10 to 50.000 polygons.

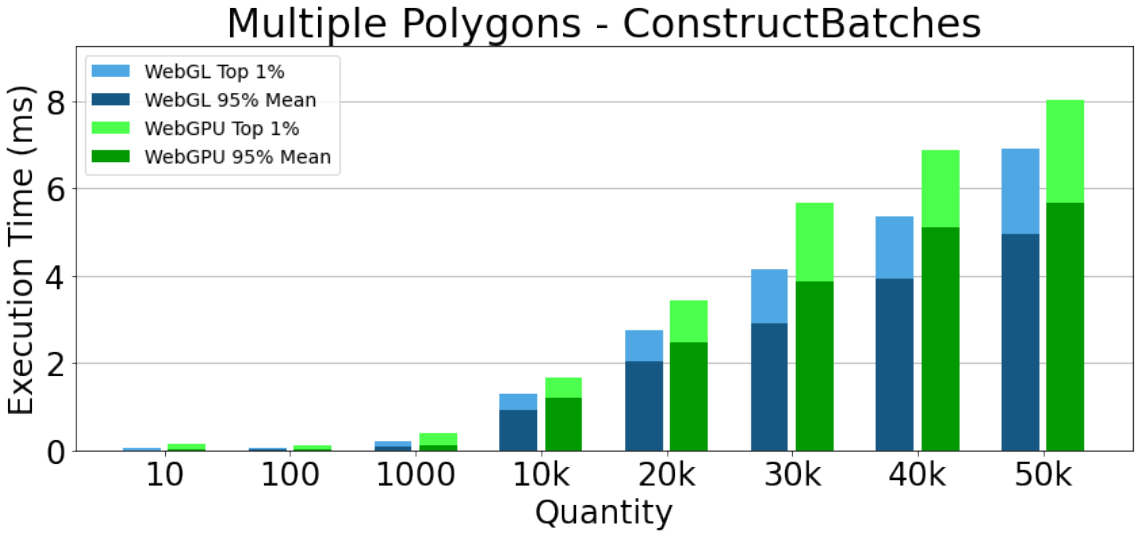

Figure D.8: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU ConstructBatches frame times, in milliseconds, for the Multiple Polygons test. Lower is better. The workloads range from 10 to 50.000 polygons.

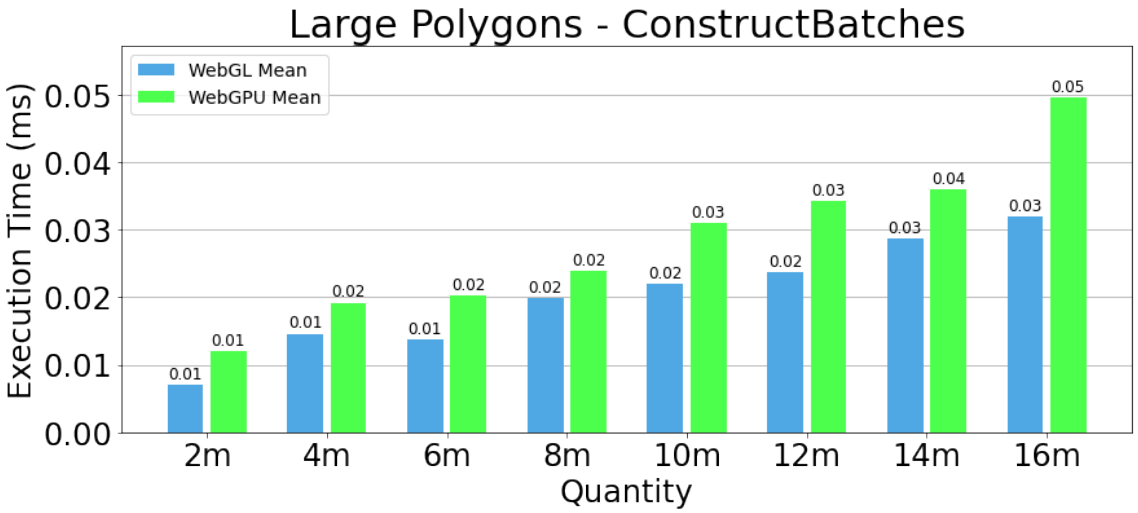

Figure D.9: Comparison of the mean WebGL and WebGPU ConstructBatches frame times, in milliseconds, for the Large Polygons test. Lower is better. The workloads range from 2 million to 16 million vertices.

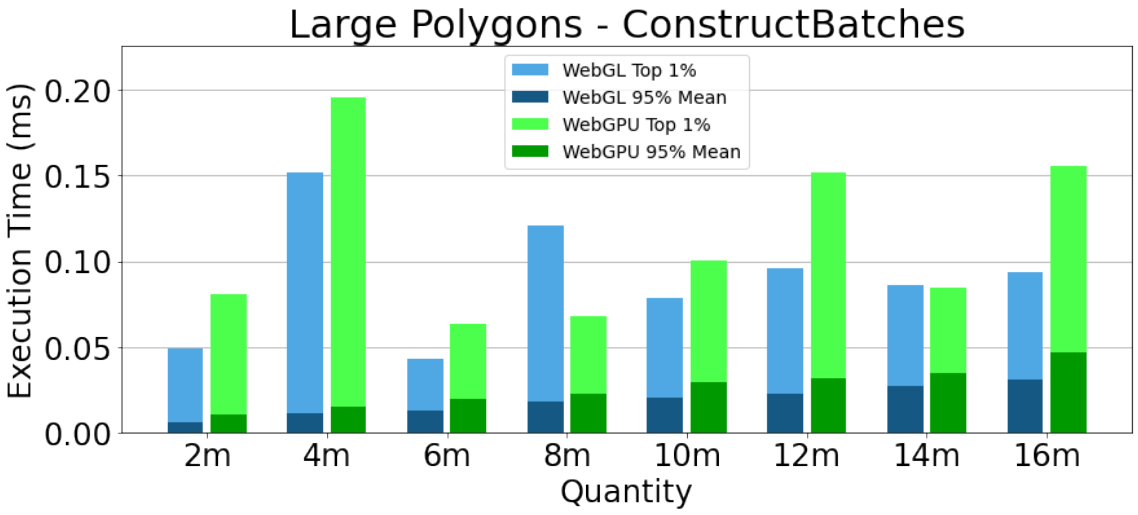

Figure D.10: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU ConstructBatches frame times, in milliseconds, for the Large Polygons test. Lower is better. The workloads range from 2 million to 16 million vertices.

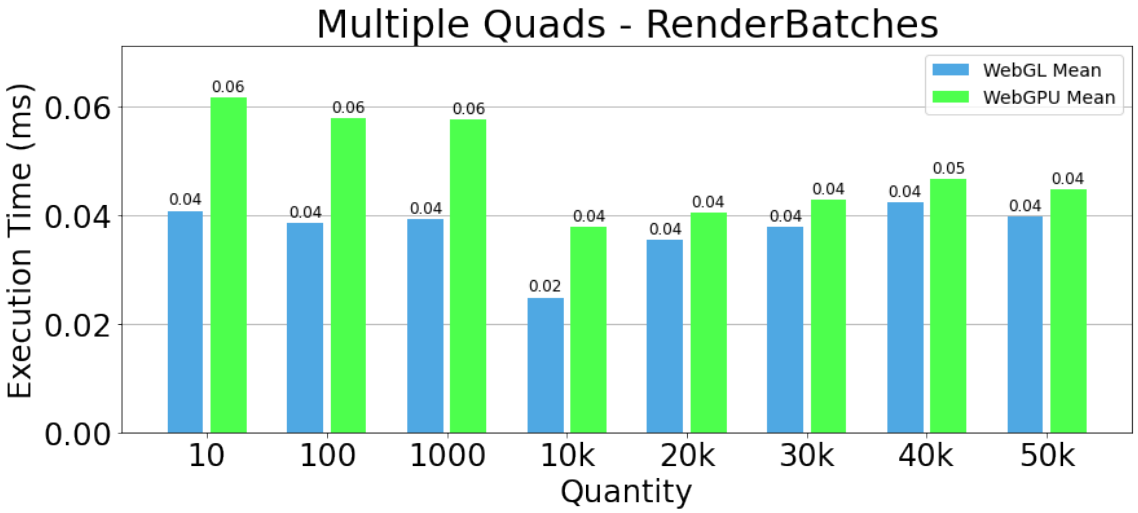

Figure D.11: Comparison of the mean WebGL and WebGPU RenderBatches frame times, in milliseconds, for the Multiple Quads test. Lower is better. The workloads range from 10 to 50.000 quads.

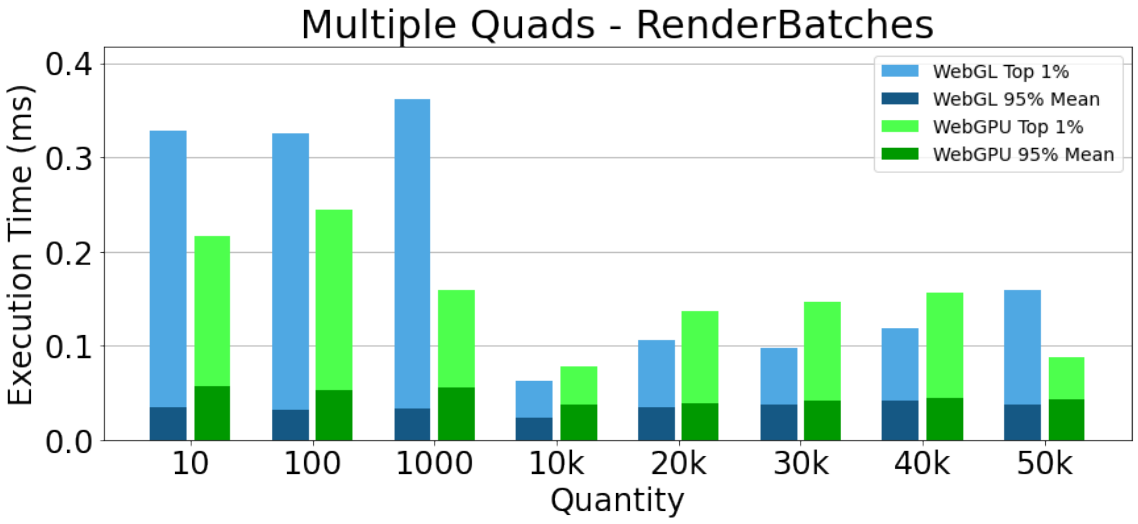

Figure D.12: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU RenderBatches frame times, in milliseconds, for the Multiple Quads test. Lower is better. The workloads range from 10 to 50.000 quads.

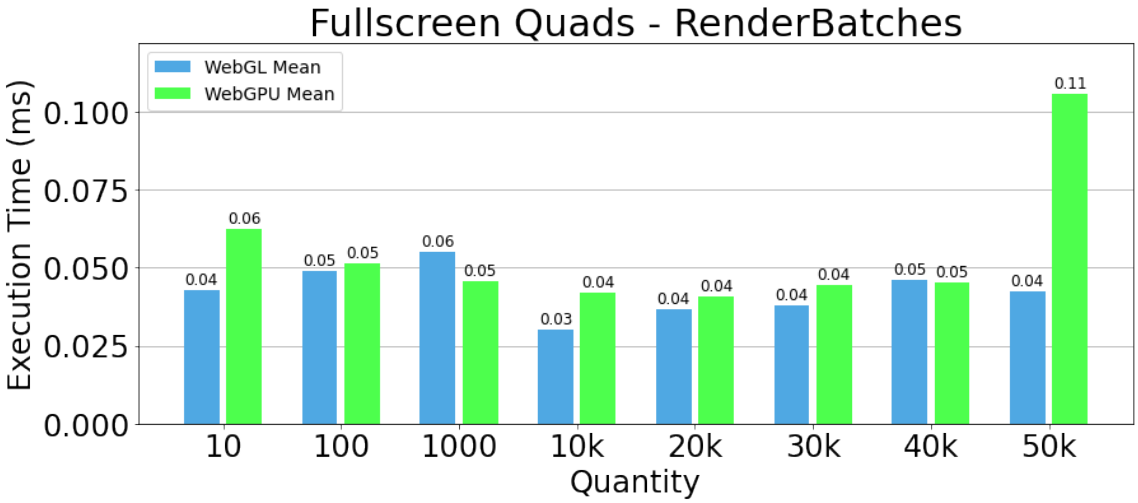

Figure D.13: Comparison of the mean WebGL and WebGPU RenderBatches frame times, in milliseconds, for the Full-screen Quads test. Lower is better. The workloads range from 10 to 50.000 full-screen quads.

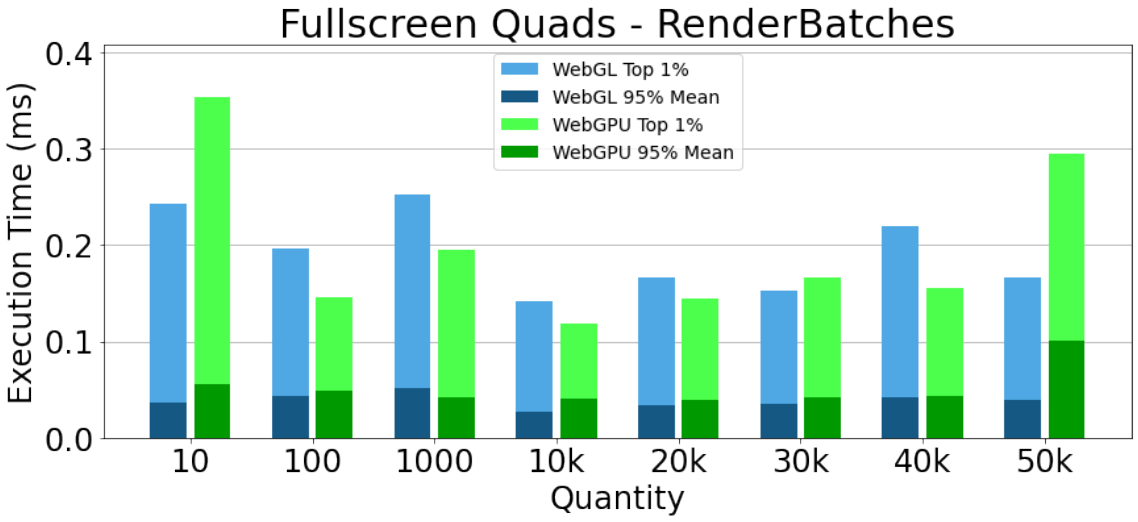

Figure D.14: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU RenderBatches frame times, in milliseconds, for the Full-screen Quads test. Lower is better. The workloads range from 10 to 50.000 full-screen quads.

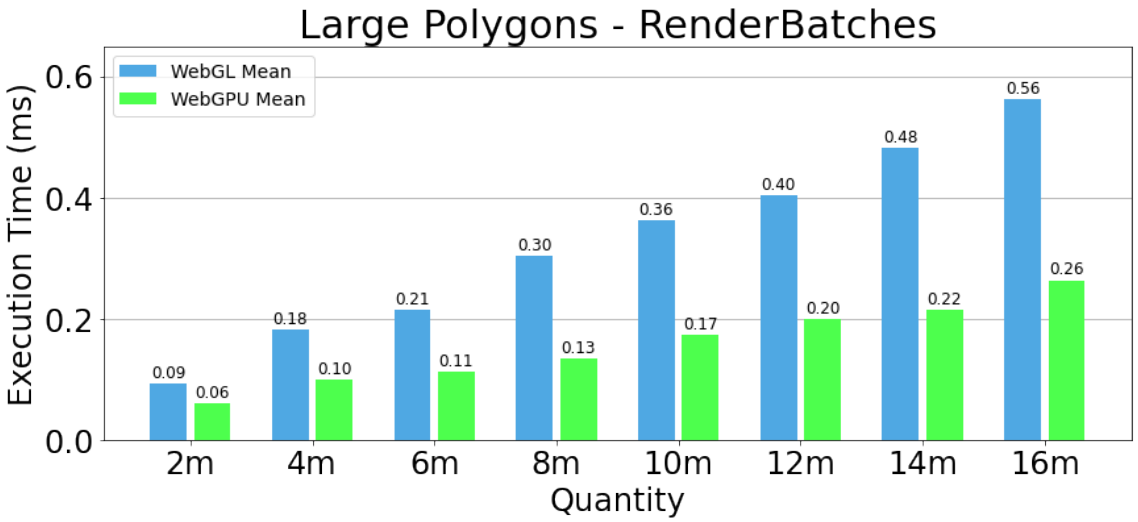

Figure D.15: Comparison of the mean WebGL and WebGPU RenderBatches frame times, in milliseconds, for the Large Polygons test. Lower is better. The workloads range from 2 million to 16 million vertices.

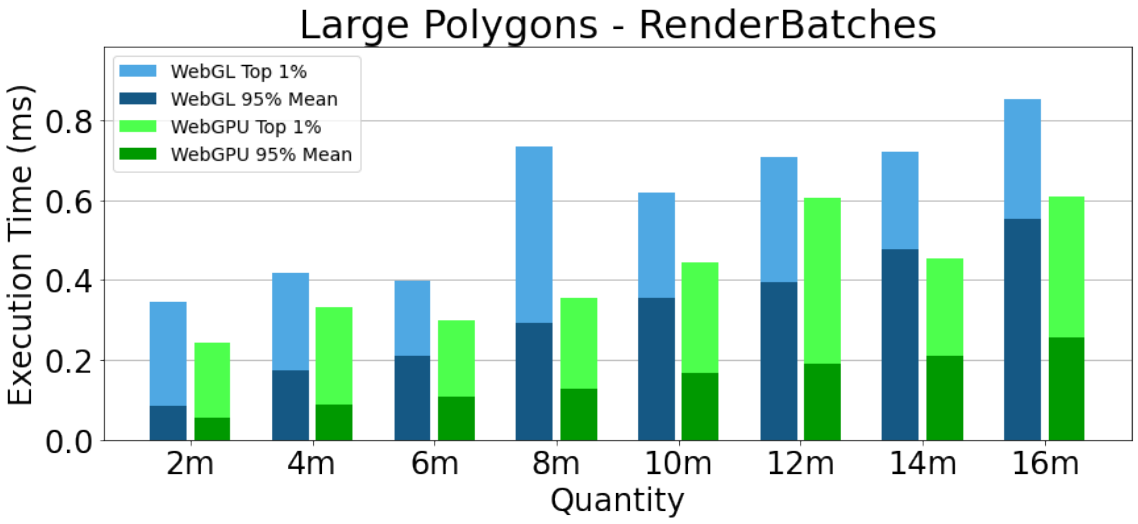

Figure D.16: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU RenderBatches frame times, in milliseconds, for the Large Polygons test. Lower is better. The workloads range from 2 million to 16 million vertices.

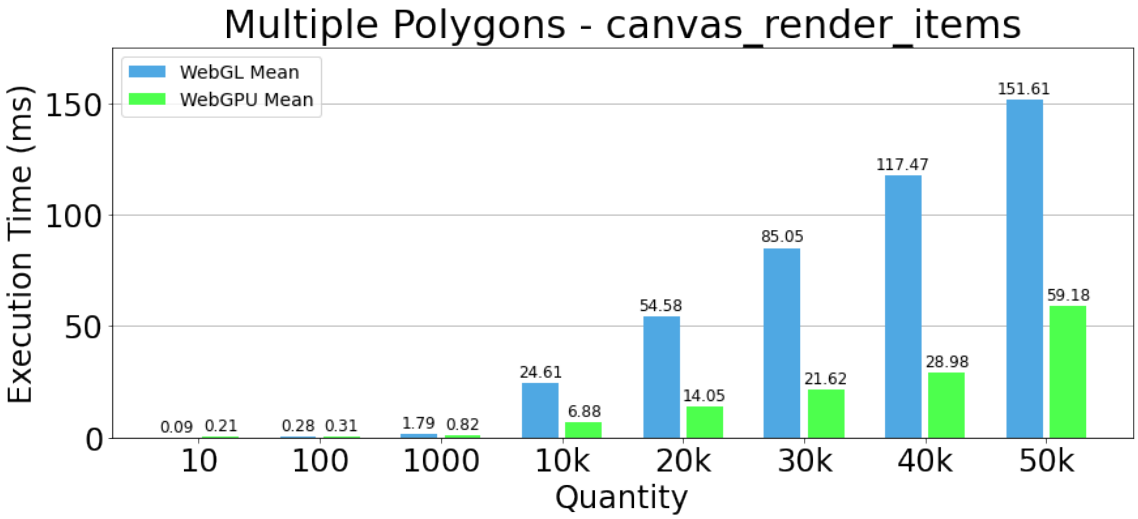

Figure D.17: Comparison of the mean WebGL and WebGPU canvas_render_items frame times, in milliseconds, for the Multiple Polygons test. Lower is better. The workloads range from 10 to 50.000 polygons.

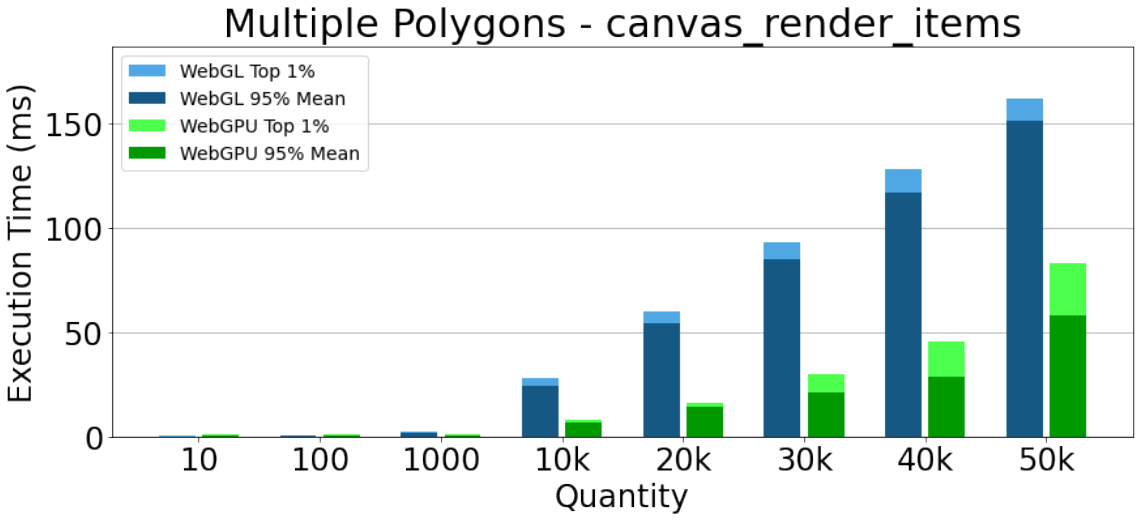

Figure D.18: Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU canvas_render_items frame times, in milliseconds, for the Multiple Polygons test. Lower is better. The workloads range from 10 to 50.000 polygons.

Appendix E

CPU Time Tables

Table E.1: Resulting mean, high 1% mean and low 95 % mean CPU frame times, in milliseconds, for the WebGPU and WebGL Rasterizers rendering the variable numbers of multiple polygons. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean CPU times (ms)				High 1% CPU times (ms)			Low 95% CPU times (ms)		
Multiple Polygons	WebGPU	WebGL	p	$S_{Latency}$	WebGPU	WebGL	p	WebGPU	WebGL	p
10, Overall CPU	0.421	0.776	<0.001	1.843	1.232	1.649	0.007	0.401	0.744	<0.001
10, canvas_render_items	0.208	0.095	<0.001		0.857	0.337	0.001	0.193	0.088	<0.001
10, ConstructBatches	0.020	0.005	<0.001		0.162	0.044	<0.001	0.018	0.004	<0.001
10, RenderBatches	0.096	0.049	<0.001		0.570	0.232	0.005	0.087	0.044	<0.001
100, Overall CPU	0.493	1.117	<0.001	2.266	1.162	2.004	<0.001	0.476	1.086	<0.001
100, canvas_render_items	0.305	0.281	<0.001		0.858	0.631	0.161	0.293	0.271	<0.001
100, ConstructBatches	0.037	0.014	<0.001		0.107	0.071	0.027	0.035	0.013	<0.001
100, RenderBatches	0.193	0.228	<0.001		0.699	0.549	0.361	0.183	0.219	<0.001
1000, Overall CPU	1.086	3.604	<0.001	3.319	1.749	6.191	<0.001	1.064	3.551	<0.001
1000, canvas_render_items	0.819	1.795	<0.001		1.378	2.474	<0.001	0.802	1.774	<0.001
1000, ConstructBatches	0.129	0.089	<0.001		0.384	0.213	<0.001	0.124	0.086	<0.001
1000, RenderBatches	0.622	1.653	<0.001		1.064	2.333	<0.001	0.610	1.633	<0.001
10k, Overall CPU	8.548	32.793	<0.001	3.836	10.623	36.779	<0.001	8.473	32.659	<0.001
10k, canvas_render_items	6.880	24.608	<0.001		7.929	27.898	<0.001	6.842	24.508	<0.001
10k, ConstructBatches	1.214	0.936	<0.001		1.678	1.290	<0.001	1.196	0.922	<0.001
10k, RenderBatches	5.499	23.285	<0.001		6.317	26.484	<0.001	5.472	23.187	<0.001
20k, Overall CPU	17.349	64.028	<0.001	3.691	20.578	70.267	<0.001	17.211	63.776	<0.001
20k, canvas_render_items	14.055	54.579	<0.001		15.952	59.737	<0.001	13.980	54.378	<0.001
20k, ConstructBatches	2.499	2.077	<0.001		3.449	2.755	<0.001	2.462	2.047	<0.001
20k, RenderBatches	11.322	51.897	<0.001		12.592	57.005	<0.001	11.275	51.703	<0.001
30k, Overall CPU	26.756	95.596	<0.001	3.573	35.890	104.974	<0.001	26.506	95.224	<0.001
30k, canvas_render_items	21.621	85.055	<0.001		30.212	93.032	<0.001	21.427	84.749	<0.001
30k, ConstructBatches	3.917	2.954	<0.001		5.663	4.147	<0.001	3.861	2.904	<0.001
30k, RenderBatches	17.305	81.206	<0.001		25.458	88.582	<0.001	17.143	80.914	<0.001
40k, Overall CPU	36.066	129.399	<0.001	3.588	51.556	141.631	<0.001	35.738	128.920	<0.001
40k, canvas_render_items	28.978	117.470	<0.001		45.869	128.163	<0.001	28.699	117.058	<0.001
40k, ConstructBatches	5.186	3.996	<0.001		6.887	5.365	<0.001	5.118	3.933	<0.001
40k, RenderBatches	23.277	112.357	<0.001		40.860	122.806	<0.001	23.021	111.959	<0.001
50k, Overall CPU	65.336	165.028	<0.001	2.526	88.754	177.235	<0.001	64.473	164.519	<0.001
50k, canvas_render_items	59.179	151.610	<0.001		82.886	162.062	<0.001	58.296	151.189	<0.001
50k, ConstructBatches	5.767	5.030	<0.001		8.026	6.912	<0.001	5.669	4.951	<0.001
50k, RenderBatches	52.361	145.309	<0.001		76.367	155.460	<0.001	51.456	144.903	<0.001

Table E.2: Resulting mean, high 1% mean and low 95 % mean CPU frame times, in milliseconds, for the WebGPU and WebGL Rasterizers rendering the variable numbers of multiple quads. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean CPU times (ms)				**High 1% CPU times (ms)**			**Low 95% CPU times (ms)**		
Multiple Quads	WebGPU	WebGL	p	$S_{Latency}$	WebGPU	WebGL	p	WebGPU	WebGL	p
10, Overall CPU	0.347	1.551	<0.001	4.470	0.847	2.912	<0.001	0.333	1.497	<0.001
10, canvas_render_items	0.159	0.126	<0.001		0.469	0.572	0.330	0.151	0.117	<0.001
10, ConstructBatches	0.015	0.008	<0.001		0.071	0.052	0.006	0.014	0.007	<0.001
10, RenderBatches	0.062	0.041	<0.001		0.216	0.329	0.212	0.057	0.035	<0.001
100, Overall CPU	0.368	1.283	<0.001	3.486	0.835	2.397	<0.001	0.351	1.241	<0.001
100, canvas_render_items	0.174	0.134	<0.001		0.484	0.537	0.438	0.163	0.125	<0.001
100, ConstructBatches	0.035	0.023	<0.001		0.205	0.202	0.936	0.031	0.019	<0.001
100, RenderBatches	0.058	0.039	<0.001		0.245	0.325	0.317	0.052	0.032	<0.001
1000, Overall CPU	0.734	1.591	<0.001	2.168	1.117	3.025	<0.001	0.720	1.540	<0.001
1000, canvas_render_items	0.372	0.294	<0.001		0.606	0.906	<0.001	0.364	0.273	<0.001
1000, ConstructBatches	0.218	0.151	<0.001		0.367	0.596	<0.001	0.214	0.139	<0.001
1000, RenderBatches	0.058	0.039	<0.001		0.159	0.362	<0.001	0.055	0.033	<0.001
10k, Overall CPU	2.696	4.934	<0.001	1.830	3.677	7.679	<0.001	2.660	4.847	<0.001
10k, canvas_render_items	1.234	1.095	<0.001		1.770	1.711	0.585	1.215	1.077	<0.001
10k, ConstructBatches	1.053	0.748	<0.001		1.533	1.149	<0.001	1.037	0.736	<0.001
10k, RenderBatches	0.038	0.025	<0.001		0.078	0.063	0.109	0.037	0.024	<0.001
20k, Overall CPU	4.752	5.992	<0.001	1.261	6.279	9.932	<0.001	4.696	5.873	<0.001
20k, canvas_render_items	2.154	2.184	0.007		12.854	3.726	<0.001	2.127	2.132	0.635
20k, ConstructBatches	1.953	1.617	<0.001		2.590	2.592	0.979	1.928	1.583	<0.001
20k, RenderBatches	0.040	0.035	<0.001		10.136	0.106	0.032	0.038	0.034	<0.001
30k, Overall CPU	8.152	8.501	<0.001	1.043	9.277	12.141	<0.001	8.111	8.350	<0.001
30k, canvas_render_items	3.612	3.272	<0.001		4.323	4.734	<0.001	3.587	3.205	<0.001
30k, ConstructBatches	3.290	2.391	<0.001		3.944	3.398	<0.001	3.268	2.344	<0.001
30k, RenderBatches	0.043	0.038	<0.001		0.147	0.098	0.072	0.041	0.037	<0.001
40k, Overall CPU	10.013	13.085	<0.001	1.307	12.542	15.864	<0.001	9.909	12.961	<0.001
40k, canvas_render_items	4.532	5.330	<0.001		5.648	6.687	<0.001	4.483	5.271	<0.001
40k, ConstructBatches	4.080	3.842	<0.001		5.153	4.875	<0.001	4.032	3.799	<0.001
40k, RenderBatches	0.047	0.042	<0.001		0.156	0.119	0.502	0.045	0.041	<0.001
50k, Overall CPU	11.170	17.122	<0.001	1.533	15.442	20.581	<0.001	10.978	16.993	<0.001
50k, canvas_render_items	5.288	6.987	<0.001		7.630	8.560	0.003	5.208	6.927	<0.001
50k, ConstructBatches	4.707	5.083	<0.001		6.300	6.210	0.152	4.636	5.039	<0.001
50k, RenderBatches	0.045	0.040	<0.001		0.088	0.159	0.003	0.044	0.038	<0.001

Table E.3: Resulting mean, high 1% mean and low 95 % mean CPU frame times, in milliseconds, for the WebGPU and WebGL Rasterizers rendering the variable numbers of full-screen quads. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean CPU times (ms)				**High 1% CPU times (ms)**			**Low 95% CPU times (ms)**		
Full-Screen Quads	WebGPU	WebGL	p	$S_{Latency}$	WebGPU	WebGL	p	WebGPU	WebGL	p
10, Overall CPU	0.375	1.398	<0.001	3.728	0.902	2.719	<0.001	0.358	1.352	<0.001
10, canvas_render_items	0.168	0.134	<0.001		0.598	0.393	0.005	0.156	0.125	<0.001
10, ConstructBatches	0.016	0.009	<0.001		0.133	0.096	0.142	0.013	0.007	<0.001
10, RenderBatches	0.062	0.043	<0.001		0.354	0.242	0.066	0.055	0.037	<0.001
100, Overall CPU	0.334	1.493	<0.001	4.470	0.617	2.735	<0.001	0.325	1.449	<0.001
100, canvas_render_items	0.155	0.172	<0.001		0.302	0.395	<0.001	0.150	0.164	<0.001
100, ConstructBatches	0.030	0.028	<0.001		0.080	0.147	<0.001	0.029	0.025	<0.001
100, RenderBatches	0.051	0.049	0.004		0.146	0.197	0.004	0.049	0.044	<0.001
1000, Overall CPU	0.637	43.208	<0.001	67.830	1.319	52.918	<0.001	0.611	42.909	<0.001
1000, canvas_render_items	0.293	0.385	<0.001		0.749	0.734	0.720	0.277	0.373	<0.001
1000, ConstructBatches	0.156	0.184	<0.001		0.445	0.442	0.884	0.146	0.177	<0.001
1000, RenderBatches	0.045	0.055	<0.001		0.194	0.253	0.209	0.042	0.051	<0.001
10k, Overall CPU	3.235	445.146	<0.001	137.603	4.781	452.812	<0.001	3.182	444.953	<0.001
10k, canvas_render_items	1.394	1.277	<0.001		2.254	1.757	<0.001	1.361	1.260	<0.001
10k, ConstructBatches	1.174	0.793	<0.001		1.946	1.129	<0.001	1.144	0.782	<0.001
10k, RenderBatches	0.042	0.030	<0.001		0.119	0.141	0.177	0.040	0.028	<0.001
20k, Overall CPU	5.316	890.508	<0.001	167.515	6.631	896.947	<0.001	5.273	890.302	<0.001
20k, canvas_render_items	2.379	2.620	<0.001		3.238	3.302	0.277	2.352	2.594	<0.001
20k, ConstructBatches	2.156	1.804	<0.001		2.923	2.391	<0.001	2.132	1.784	<0.001
20k, RenderBatches	0.041	0.037	<0.001		0.144	0.167	0.215	0.039	0.034	<0.001
30k, Overall CPU	8.390	1353.816	<0.001	161.361	10.062	1367.305	<0.001	8.326	1353.269	<0.001
30k, canvas_render_items	3.523	4.060	<0.001		4.336	5.635	0.025	3.494	4.017	<0.001
30k, ConstructBatches	3.213	2.808	<0.001		3.970	3.829	0.436	3.187	2.778	<0.001
30k, RenderBatches	0.045	0.038	<0.001		0.167	0.153	0.548	0.043	0.036	<0.001
40k, Overall CPU	10.939	1786.538	<0.001	163.344	12.872	1802.173	<0.001	10.863	1786.011	<0.001
40k, canvas_render_items	4.722	5.726	<0.001		5.893	7.724	<0.001	4.679	5.666	<0.001
40k, ConstructBatches	4.274	4.047	<0.001		5.359	5.611	0.236	4.234	4.002	<0.001
40k, RenderBatches	0.045	0.046	0.116		0.156	0.219	0.548	0.043	0.042	<0.001
50k, Overall CPU	1138.584	2202.755	<0.001	1.935	1355.518	2209.514	<0.001	1127.640	2202.478	<0.001
50k, canvas_render_items	1131.196	6.167	<0.001		1348.941	8.416	<0.001	1120.228	6.066	<0.001
50k, ConstructBatches	5.763	4.536	<0.001		7.239	6.078	<0.001	5.714	4.469	<0.001
50k, RenderBatches	0.106	0.042	<0.001		0.295	0.166	<0.001	0.101	0.040	<0.001

Table E.4: Resulting mean, high 1% mean and low 95 % mean CPU frame times, in milliseconds, for the WebGPU and WebGL Rasterizers rendering the variable numbers of large polygons. Lower is better. $S_{Latency}$ denotes WebGPU speed-up. p is the t-test yielded p-value.

	Mean CPU times (ms)				**High 1% CPU times (ms)**			**Low 95% CPU times (ms)**		
Large Polygons	WebGPU	WebGL	p	$S_{Latency}$	WebGPU	WebGL	p	WebGPU	WebGL	p
2m, Overall CPU	0.198	5.459	<0.001	27,571	0.621	6.789	<0.001	0.186	5.401	<0.001
2m, canvas_render_items	0.110	0.135	<0.001		0.389	0.423	0.628	0.102	0.127	<0.001
2m, ConstructBatches	0.012	0.007	<0.001		0.080	0.049	0.048	0.011	0.006	<0.001
2m, RenderBatches	0.060	0.093	<0.001		0.243	0.344	<0.001	0.054	0.087	<0.001
4m, Overall CPU	0.267	0.947	<0.001	3.547	0.752	3.358	<0.001	0.254	0.904	<0.001
4m, canvas_render_items	0.167	0.242	<0.001		0.510	0.487	0.725	0.156	0.234	<0.001
4m, ConstructBatches	0.019	0.015	<0.001		0.196	0.152	<0.001	0.015	0.011	<0.001
4m, RenderBatches	0.099	0.182	<0.001		0.333	0.419	<0.001	0.090	0.174	<0.001
6m, Overall CPU	0.256	0.953	<0.001	3.723	0.578	1.515	<0.001	0.249	0.934	<0.001
6m, canvas_render_items	0.170	0.262	<0.001		0.413	0.469	0.342	0.165	0.255	<0.001
6m, ConstructBatches	0.020	0.014	<0.001		0.063	0.043	0.022	0.019	0.013	<0.001
6m, RenderBatches	0.112	0.214	<0.001		0.300	0.399	0.021	0.108	0.209	<0.001
8m, Overall CPU	0.285	1.150	<0.001	4.035	0.631	1.752	<0.001	0.277	1.128	<0.001
8m, canvas_render_items	0.196	0.362	<0.001		0.436	0.831	<0.001	0.190	0.348	<0.001
8m, ConstructBatches	0.024	0.020	<0.001		0.068	0.121	<0.001	0.023	0.018	<0.001
8m, RenderBatches	0.134	0.303	<0.001		0.356	0.735	<0.001	0.129	0.291	<0.001
10m, Overall CPU	0.348	1.401	<0.001	4.026	0.692	2.026	<0.001	0.339	1.378	<0.001
10m, canvas_render_items	0.247	0.424	<0.001		0.543	0.699	0.055	0.240	0.416	<0.001
10m, ConstructBatches	0.031	0.022	<0.001		0.100	0.079	0.040	0.029	0.021	<0.001
10m, RenderBatches	0.173	0.362	<0.001		0.444	0.619	0.037	0.167	0.355	<0.001
12m, Overall CPU	0.379	5.663	<0.001	14.942	0.866	9.990	<0.001	0.368	5.573	<0.001
12m, canvas_render_items	0.276	0.462	<0.001		0.726	0.830	0.231	0.266	0.453	<0.001
12m, ConstructBatches	0.034	0.024	<0.001		0.152	0.096	0.002	0.032	0.022	<0.001
12m, RenderBatches	0.199	0.403	<0.001		0.605	0.707	0.231	0.191	0.395	<0.001
14m, Overall CPU	0.390	6.078	<0.001	15.585	0.743	12.250	<0.001	0.381	5.769	<0.001
14m, canvas_render_items	0.292	0.547	<0.001		0.610	0.819	0.008	0.285	0.540	<0.001
14m, ConstructBatches	0.036	0.029	<0.001		0.085	0.086	0.899	0.035	0.027	<0.001
14m, RenderBatches	0.215	0.482	<0.001		0.455	0.721	<0.001	0.210	0.476	<0.001
16m, Overall CPU	0.477	6.952	<0.001	14.574	0.876	12.503	<0.001	0.467	6.676	<0.001
16m, canvas_render_items	0.360	0.633	<0.001		0.728	0.949	0.013	0.352	0.624	<0.001
16m, ConstructBatches	0.050	0.032	<0.001		0.155	0.094	<0.001	0.047	0.031	<0.001
16m, RenderBatches	0.263	0.562	<0.001		0.609	0.853	0.007	0.256	0.554	<0.001

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden