A comparison of Performance on WebGPU and WebGL in the Godot game engine

1st Emil Fransson
Department of Computer Science
Blekinge Institute of Technology
Karlskrona, Sweden
emfa17@student.bth.se

2nd Jonatan Hermansson
Department of Computer Science
Blekinge Institute of Technology
Karlskrona, Sweden
johm18@student.bth.se

3rd Yan Hu
Department of Computer Science
Blekinge Institute of Technology
Karlskrona, Sweden
yan.hu@bth.se

***Abstract*—WebGL has been the standard API for rendering graphics on the web 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 about its capabilities when used as a rendering backend in game engines. This paper aims to investigate performance differences between WebGL and WebGPU. It is done in the context of the game engine Godot, and the measured performance is that of the CPU and GPU frame time. 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.**

***Index Terms*—Game Engine, Performance Overhead, Rendering, WebGPU, WebGL**

I. 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) equipped with modern computers. Commonly used APIs include Direct3D [1] for machines running Windows, Metal [2] for Apple products, and Vulkan [3] and OpenGL [4] as a cross-platform alternative. Those APIs all target native platforms, and as is evident, many choices are available to developers. However, when it comes to rendering on the web, the choices narrow significantly. WebGL was the lowest-level alternative for rendering on the web [5]. It is based on the aforementioned OpenGL native API and adopts the same workflow and syntax.

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 [6]. WebGL is a mature API supported by many different hardware products and browsers. It has been applied in many environments and fields, such as rendering backends in the gaming industry and for visualization purposes in medicine and geospatial applications. WebGL currently exists as a possible rendering backend in the Godot engine for rendering graphics on the web platform.

This paper consists of an implementation of a rendering backend for the game engine Godot using the currently latest low-level web graphics API WebGPU, and comparing its performance in various test cases to the performance of the WebGL backend currently implemented in Godot. 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 [7]. 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.

Section 2 lists some related work in the areas of WebGPU. Section 3 details the overall research method including the implementation details and how the experiment and data gathering were conducted. Section 4 presents the results and analysis of the conducted experiment. Section 5 contains a discussion of the performed work, and the final section presents the conclusions and future work.

II. Related Work

A study by 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 [8].

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 that CUDA is 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 [9].

Usher and Pascucci compared the computing capabilities of WebGPU with those of native Vulkan. 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 [10].

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). GraphWaGu is the only GPU-leveraged library that is able to compute iterations of the graph algorithms in parallel. So 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 [11].

There has been quite some research done in the field of WebGPU and its general computing 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 doing this study, no research could be found that places its context inside the environment of a game engine. The work presented in this paper aims to effectively reduce the research gap on WebGPU as a new rasterization technology for the web in the environment of the Godot game engine, grounding the research and results in real usability scenarios.

III. Methods

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 doing this study [12], 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 its open-source nature.

A. Implementation

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.
3) No optimizations are allowed for the WebGPU Rasterizer on the CPU-side or GPU-side, which would put it at an unfair advantage over the WebGL Rasterizer.
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.

To achieve the 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. 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 is a technique used to group similar items and render them together to avoid unnecessary resource binding. For blitting, a separate pipeline was set up with a vertex shader that simply renders a triangle covering the entire back buffer, and a fragment shader that textures this triangle using the main render target texture.

B. 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. The gathered data in the conducted experiment is that of the frame time measured in milliseconds. 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.

The timings are gathered as averages over 2000 frames. 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 used later when enough samples have been gathered to calculate an average elapsed time. 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.

The experiments include two categories: simple 2D games and 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* [13], in which the player must avoid obstacles and gather apples in order for the snake character to grow longer and longer.
2) *Evader* [14], in which the player must avoid incoming shapes on the highway.
3) *Checkers*[1] [15], in which the player plays the checkers game either versus an AI or optionally versus another player locally.
4) *Falling Cats* [16], in which the player must catch cats falling from a tree before they hit the ground.
5) *Deck Before Dawn* [17], 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] [18], in which the player must navigate a duck character in a finite number of sequences in order to collect all ducklings.

The synthetic tests are applied 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.
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.
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.
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. 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.

[1]The version used in testing is v1.0.1-0-g7a4203b
[2]The version used for testing is v1.0.0

C. Hardware and Software Specification

The hardware as well as what versions of relevant graphics drivers were used are presented in Table I.

TABLE I
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

IV. Results

A. Performance Comparison of Game Tests

The GPU frame time is firstly presented and the second the total CPU frame time that measured. 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.

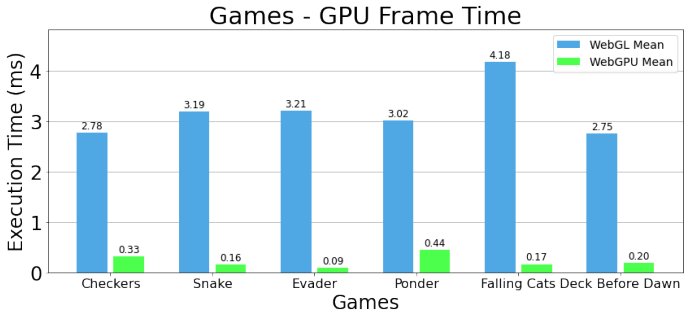

Fig. 1. Comparison of the mean WebGL and WebGPU GPU frame times, in milliseconds, for the various games.

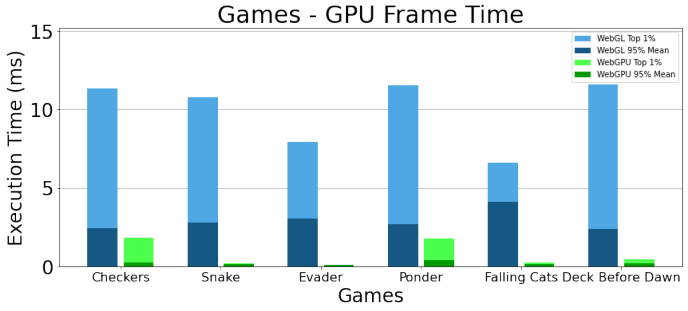

Fig. 2. Comparison of the highest 1% mean and the lowest 95% mean WebGL and WebGPU GPU frame times, in milliseconds, for the various games.

1) GPU Frame Time: In Figure 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. Furthermore, a speed-up of WebGPU to WebGL ranges between 6.822, in the case of Ponder, and 35.611, in the case of Evader. Figure 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.

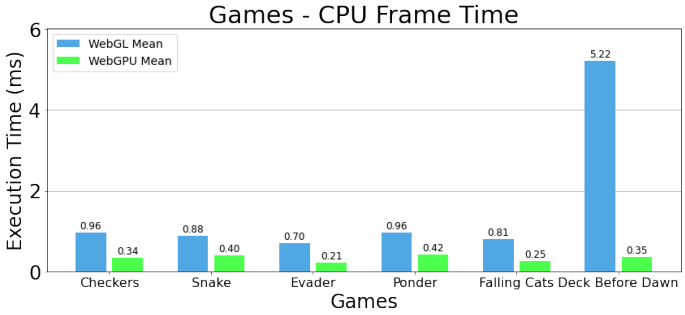

Fig. 3. Comparison of the mean WebGL and WebGPU CPU frame times, in milliseconds, for the various games.

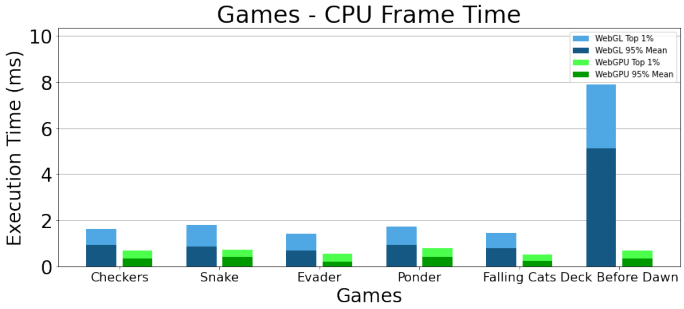

Fig. 4. Comparison of the highest 1% mean and lowest 95% mean WebGL and WebGPU CPU frame times, in milliseconds, for the various games.

2) CPU Frame Time: In Figure 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 terms of how much shorter the CPU frame time is with the WebGPU implementation. Figure 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 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.

B. Performance Comparison of Synthetic Tests

1) GPU Frame Time: 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 5. 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. The results of rendering multiple

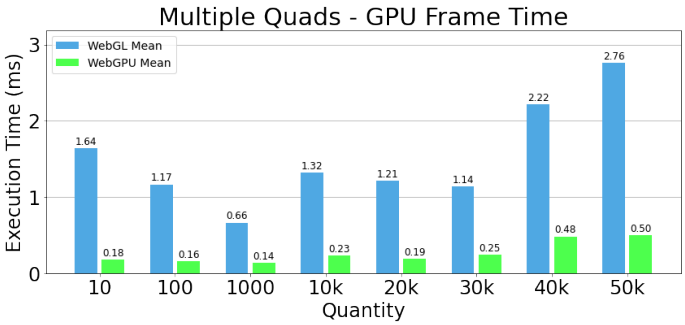

Fig. 5. Comparison of the mean WebGL and WebGPU GPU frame times, in milliseconds, for the Multiple Quads test. The workloads range from 10 to 50.000 quads.

full-screen quads show how considerate pressure was put on both Rasterizers, with long GPU mean frame times for all tests above 1000 quads. In Figure 6, 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.

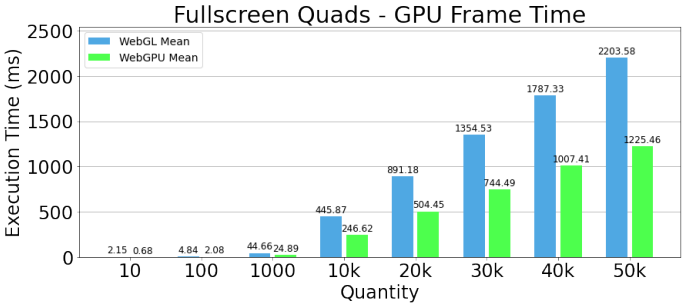

Fig. 6. Comparison of the mean WebGL and WebGPU GPU frame times, in milliseconds, for the Full-screen quads test. The workloads range from 10 to 50.000 full-screen quads.

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, in Figure 7, 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). The GPU frame times for the Large Polygons synthetic test show that WebGPU is roughly 2 - 3 times faster across 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,

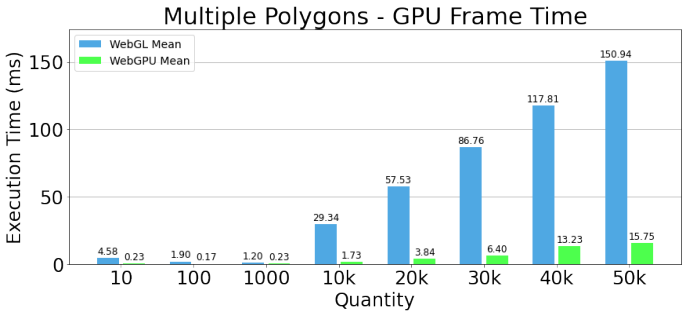

Fig. 7. Comparison of the mean WebGL and WebGPU GPU frame times, in milliseconds, for the Multiple Polygons test. The workloads range from 10 to 50.000 polygons.

WebGL again shows a bigger spread of frame times, with WebGPU remaining fairly stable (see Figure 8).

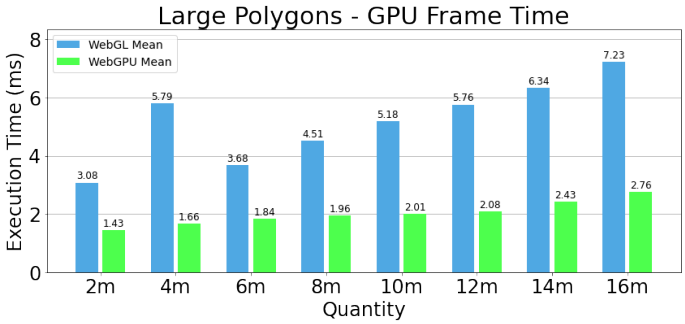

Fig. 8. Comparison of the mean WebGL and WebGPU GPU frame times, in milliseconds, for the Large Polygons test. The workloads range from 2 million to 16 million vertices.

2) CPU Frame Time: For the Quads synthetic test, Figure 9 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.

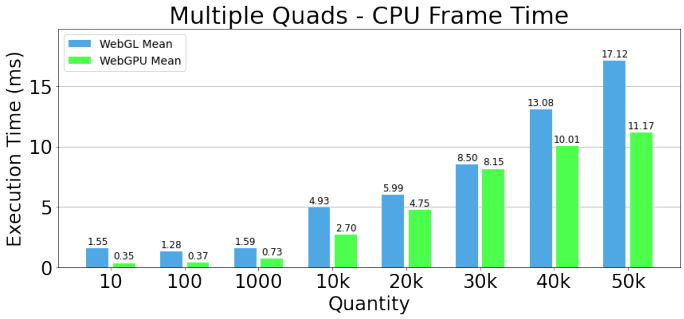

Fig. 9. Comparison of the mean WebGL and WebGPU CPU frame times, in milliseconds, for the Multiple Quads test. The workloads range from 10 to 50.000 quads.

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 10. 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 6.

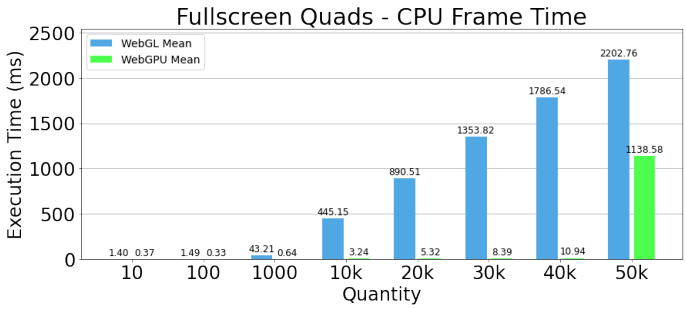

Fig. 10. Comparison of the mean WebGL and WebGPU CPU frame times, in milliseconds, for the Full-screen Quads test. The workloads range from 10 to 50.000 full-screen quads.

The total CPU frame time of the Polygons synthetic tests shows that outside of the 10 polygon test case, the WebGPU implementation is faster than the WebGL one (Figure 11). 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.

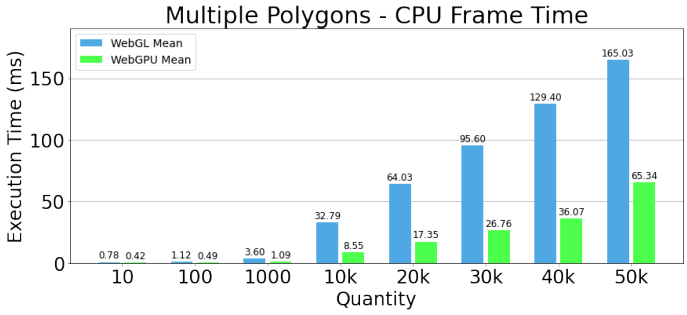

Fig. 11. Comparison of the mean WebGL and WebGPU CPU frame times, in milliseconds, for the Multiple Polygons test. The workloads range from 10 to 50.000 polygons.

In the Large Polygons synthetic test, the total CPU time recorded for the two Rasterizers in Figure 12 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).

We used paired T-tests as the main statistical significance tests in our study. The p-value of all the comparisons of overall mean values of GPU and CPU frame times are less than 0.05, which means it is significantly different. In all games, WebGPU outperforms WebGL both in terms of overall CPU and GPU mean frame times. The synthetic tests show similar results, with WebGPU outperforming WebGL in both CPU and

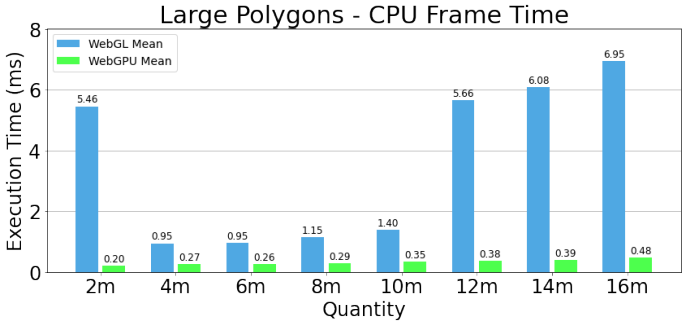

Fig. 12. Comparison of the mean WebGL and WebGPU CPU frame times, in milliseconds, for the Large Polygons test. The workloads range from 2 million to 16 million vertices.

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.

V. Discussion

There are several explanations as to why WebGPU performs better than WebGL at the rendering tasks presented in the study. One is due to the use of modern graphics drivers and bundled state. These provide optimizations that WebGL or OpenGL cannot achieve, and explain the superior performance of WebGPU. Despite WebGPU showing consistently better frame times than WebGL, there are still times when it struggles. For instance, the CPU frame time for the 50,000 full-screen quads synthetic test increases significantly compared to the 40 000 full-screen quads due to data uploading to the instance buffer. This function call may force CPU and GPU synchronization, leading to longer CPU times and longer GPU frame times. The reason for synchronization not being necessary for other variants is unknown and requires further study.

Aside from the already discussed performance benefits inherent to WebGPU as a modern technology, there exist other possible explanations as to why the performance of WebGL falls behind WebGPU in the experiments conducted. One of the reasons is related to stalling. WebGL experiences different types of stalling at varying workloads, while WebGPU does not. In the case of larger workloads, WebGL is several hundred milliseconds slower than WebGPU in mean GPU frame times. On the other hand, the Polygons tests show that the GPU is stalled instead as the workload increases. The CPU stalls as it waits for WebGL GPU instructions to complete, which is reflected in the exceptionally high mean CPU frame times in WebGL.

VI. 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 grounded in both game examples with realistic workloads and raw stress tests of varying workloads through synthetic experiments. 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. Furthermore, and in general, the presented results are statistically significant. The WebGPU renderer implementation is relatively naive, the better results could be achieved with a more modern graphics API workflow. 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. Another suggestion for future research is to build upon the work in this study 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.

References

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

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

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

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

[5] 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.

[6] 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

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

[8] 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.

[9] 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

[10] 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.

[11] 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

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

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

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

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

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

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

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