

Enhancing the Performance of Web-based Extended Reality with WebAssembly and WebGPU

T.M.R.D. Rodrigo
2025



Enhancing the Performance of Web-based Extended Reality with WebAssembly and WebGPU

T.M.R.D. Rodrigo
Index No: 20001509

Supervisor: Dr. K.D. Sandaruwan

May 2025

Submitted in partial fulfillment of the requirements of the
B.Sc. (Honours) in Computer Science Final Year Project



Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, to be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name: T.M.R.D. Rodrigo

Signature of Candidate:



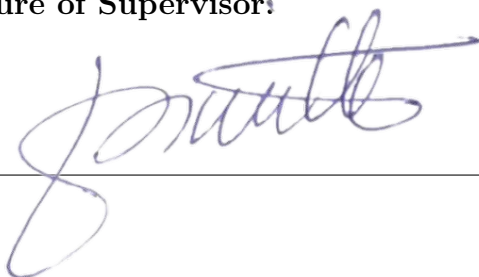
Date:

14/06/2025

This is to certify that this dissertation is based on the work of Mr. **T.M.R.D. Rodrigo** under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Supervisor's Name: Dr. K.D. Sandaruwan

Signature of Supervisor:



Date:

14/06/2025

Abstract

The increasing demand for immersive web-based experiences has driven the development of advanced browser technologies capable of delivering rich, interactive content. Among these, WebAssembly and WebGPU have emerged as powerful tools that promise near-native performance and high-fidelity rendering capabilities, respectively. This research explores the practical integration of these two technologies within the context of web-based Extended Reality (XR) applications, focusing on performance and implementation challenges.

The primary objective of this study is to investigate how WebAssembly and WebGPU can be jointly leveraged to implement performant XR applications within the browser, and to determine whether they offer tangible advantages over traditional WebGL and JavaScript-based approaches. To explore this, a prototype system supporting both Virtual Reality (VR) and Augmented Reality (AR) was developed. WebAssembly was utilized for handling CPU-related tasks, such as processing scene data, managing XR session data, and preparing GPU commands. Meanwhile, WebGPU was employed for rendering complex 3D scenes and executing large-scale matrix computations. Due to the lack of official WebGPU support in the WebXR Device API, the prototype adopted WebGL as a bridging layer for XR rendering layer handling, resulting in a hybrid rendering pipeline that was critically evaluated throughout the study.

The study was driven by a series of research questions aimed at identifying suitable implementation approaches, demonstrating a practical integration strategy, and assessing performance outcomes. A comprehensive performance evaluation was conducted across multiple dimensions, including CPU usage, GPU workload, frame rate stability, and draw call behavior under varying scene complexities. The results showed that the integration of WebAssembly and WebGPU led to notable improvements in CPU efficiency—especially in scenarios involving high-volume matrix computations. However, it did not consistently outperform the traditional WebGL-based implementation in terms of overall performance. This limitation may have caused by the overhead introduced by the interoperability

layer required to bridge WebGPU with WebGL in the absence of native WebXR support.

In the course of conducting these experiments, the relationship between scene complexity and the number of draw calls was thoroughly examined. This provided valuable insights into CPU-GPU communication bottlenecks and validated the use of draw call merging as a viable optimization technique in WebGPU-based rendering pipelines.

Despite certain limitations—such as incomplete browser support, device compatibility constraints, and the necessity for prototype-specific optimizations—the study provides a valuable foundation for future research. It offers practical guidance for developers navigating the challenges of next-generation XR development on the web and lays the groundwork for more seamless and efficient integration of WebGPU within the WebXR ecosystem.

As browser vendors continue to evolve their support for WebGPU and improve XR APIs, the insights and implementation strategies presented in this thesis will serve as a stepping stone for more performant, portable, and interactive web-based XR applications.

Keywords— Computer Graphics, Extended Reality, WebXR, Performance Optimization, WebAssembly, WebGPU, WebGL, Texture Blitting, Interoperability

Preface

This dissertation, titled "Enhancing the Performance of Web-based Extended Reality with WebAssembly and WebGPU", has been completed as partial fulfillment of the requirements for the B.Sc. in Computer Science Final Year Project (SCS4224) at the University of Colombo School of Computing, Sri Lanka. The research was conducted from June 2024 to April 2025.

The representation of computer graphics in video games and immersive experiences plays a significant role in determining real-time performance. However, as these graphical representations grow more advanced, some hardware and devices struggle to maintain adequate performance. This raises the question of whether such performance limitations can be overcome—particularly through software-level optimizations. This curiosity inspired an exploration of the domains where performance bottlenecks exist and where targeted improvements may be possible.

The development of immersive technologies on the web has reached a pivotal point, where performance and accessibility are converging to shape the next generation of digital experiences. This thesis investigates a key aspect of that evolution—the integration of WebAssembly and WebGPU in web-based Extended Reality (XR) applications. Although both technologies are powerful individually, combining them in an XR context introduces distinct challenges, especially considering the current limitations of browser APIs and inconsistent platform support.

At the heart of this research is an attempt to move beyond the theoretical promise of WebAssembly and WebGPU by implementing and evaluating a functional system that tests their practical viability in immersive web-based applications. Through design, development, and empirical evaluation, this study aims to provide insights into how these emerging technologies can be effectively leveraged for XR, and where their performance boundaries currently lie.

This document presents the outcomes of the inquiry along with the technical insights and reasoning that shaped the project, aiming to guide others working on web-based immersive systems.

Acknowledgement

First and foremost, I would like to express my sincere gratitude to my supervisor, Dr. K.D. Sandaruwan, for his continuous support, insightful guidance, and encouragement throughout the duration of this project. His expertise and constructive feedback were instrumental in shaping the direction and outcome of this research.

I am also grateful to the academic and administrative staff at the University of Colombo School of Computing for providing the necessary resources and an intellectually stimulating environment that enabled the successful completion of this work.

A special thanks goes to my fellow colleagues and peers, whose helpful discussions and collaboration have greatly contributed to my learning experience during this project. This is including to the people who lent their devices to conduct the quantitative experiments and the participants who engaged in the qualitative experiment, in which the results contributed to formulating the conclusion of the study.

I would also like to acknowledge the developers and maintainers of the open-source tools, libraries, and documentation that were crucial in the implementation and testing phases of this research.

Finally, I extend my heartfelt appreciation to my family and friends for their unwavering support, understanding, and patience throughout this journey. Their belief in me has been a constant source of motivation.

Contents

1	Introduction	1
1.1	Problem Statement	1
1.2	New Technologies and Solutions	2
1.3	Research Motivation and Objectives	3
1.4	Scope and Limitations	4
1.5	Research Aim, Questions and Objectives	4
1.5.1	Research Aim	4
1.5.2	Research Questions	5
1.5.3	Research Objectives	5
1.6	Key Terms and Concepts	6
2	Literature Review	8
2.1	History and Problem	8
2.2	WebXR	9
2.3	WebGPU	10
2.4	WebAssembly	11
2.5	WebAssembly and WebGPU	12
2.6	Critical Analysis of the Literature	13
3	Methodology	16
3.1	Research Methodology	16
3.2	Experimental Setup	16
3.3	Evaluation Plan	17
3.3.1	Quantitative Evaluation	18
3.3.2	Qualitative Evaluation	19
3.4	Proposed System Architecture	20
4	Implementation	22
4.1	WebGPU incompatibility	22
4.2	Finding a Solution for the Incompatibility	22

4.3	WebGPU-WebGL Interoperability	23
4.4	Performance Considerations	26
4.4.1	Merging Draw Calls	26
4.4.2	Render Bundles	27
4.4.3	Compute Shaders	27
4.4.4	Communication between JavaScript and C++	28
4.4.5	Compiler Optimizations	29
4.5	3D Model Import	30
4.6	Rendering Frameworks	31
4.7	Practical Implementation and Integration	32
4.7.1	ControlXR	32
4.7.2	WasmXRGPU	33
5	Results and Analysis	35
5.1	WebGPU-WebGL Interoperability Experiment	36
5.1.1	Using a Single High Poly 3D Model	37
5.1.2	Using Two 3D Models	40
5.2	ControlXR and WasmXRGPU Experiment - Quantitative Analysis	41
5.2.1	Virtual Reality (VR)	46
5.2.2	Augmented Reality (AR)	62
5.3	ControlXR and WasmXRGPU Experiment – Qualitative Analysis	88
5.3.1	Virtual Reality (VR)	88
5.3.2	Augmented Reality (AR)	88
6	Critical Evaluation of Results	95
6.1	WebGPU-WebGL Interoperability Experiment	95
6.1.1	Using a Single High Poly 3D Model	95
6.1.2	Using Two 3D Models	96
6.2	ControlXR and WasmXRGPU Experiment - Quantitative Evaluation	97
6.2.1	Virtual Reality (VR)	97
6.2.2	Augmented Reality (AR)	100
6.3	ControlXR and WasmXRGPU Experiment - Qualitative Evaluation	102
6.4	Virtual Reality vs. Augmented Reality	102
6.5	The Impact of WebGPU–WebGL Interoperability	103
6.6	Findings and Contributions	104
7	Conclusion	106
7.1	Overall Outcome	107
7.2	Limitations	108
7.3	Future Directions	109

References	112
A Appendix	118
A.1 Scatter Plots of Measured Metrics in VR	118
A.1.1 CPU Time (ms)	118
A.1.2 GPU Time (ms)	122
A.2 Violin Plots of Measured Devices in VR	125
A.2.1 Google Pixel 4A	125
A.2.2 Google Pixel 6A	126
A.2.3 Samsung Galaxy A15	127
A.2.4 Samsung Galaxy A52	128
A.2.5 Samsung Galaxy S23 Ultra	129
A.2.6 Meta Quest 2	130
A.3 Scatter Plots of Measured Metrics in AR	131
A.3.1 CPU Time (ms)	131
A.3.2 GPU Time (ms)	135
A.4 Violin Plots of Measured Devices in AR	138
A.4.1 Google Pixel 4A	138
A.4.2 Google Pixel 6A	139
A.4.3 Samsung Galaxy A15	140
A.4.4 Samsung Galaxy A52	141
A.4.5 Samsung Galaxy S23 Ultra	142
A.4.6 Meta Quest 2	143

List of Figures

3.1	Design Science Methodology (Krupitzer 2018)	17
3.2	WebXR Device API with WebGL to create a traditional WebXR app. (ControlXR Design)	20
3.3	WebXR Device API with WebGPU and WebAssembly to create a WebXR App. (Proposed WasmXRGPU Design)	21
4.1	Graphics Rendering Pipeline. (Bi et al. 2024)	23
4.2	WebGPU-WebGL interoperability approach where both graphics APIs work together	24
4.3	WebXR Device API with WebGPU and WebAssembly to create a WebXR App, that uses WebGL to copy WebGPU rendered frames. (WasmXRGPU Design)	25
4.4	ControlXR Implementation	32
4.5	WasmXRGPU Implementation	33
5.1	Comparison of Average Frame Time (ms) between defined scenarios in Table 5.1. Lower is better.	38
5.2	Comparison of Average FPS between defined scenarios in Table 5.1. Higher is better.	39
5.3	Comparison of Average WebGL Time (ms) between defined scenarios in Table 5.1. Lower is better.	39
5.4	Comparison of Average WebGPU Time (ms) between defined scenarios in Table 5.1. Lower is better.	40
5.5	Comparison of Average JavaScript Time (ms) between defined scenarios in Table 5.1. Lower is better.	40
5.6	Comparison of Average Frame Time (ms) between defined scenarios in Table 5.2. Lower is better.	42
5.7	Comparison of Average FPS between defined scenarios in Table 5.2. Higher is better.	42
5.8	Comparison of Average JavaScript Time (ms) between defined scenarios in Table 5.2. Lower is better.	43

5.9	Comparison of Average WebGL Time (ms) between defined scenarios in Table 5.2. Lower is better.	43
5.10	Comparison of Average WebGPU Time (ms) between defined scenarios in Table 5.2. Lower is better.	44
5.11	Used 3D model	45
5.12	Comparison of Average CPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	47
5.13	Comparison of Average GPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	48
5.14	Comparison of Average FPS between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.	48
5.15	Comparison of Average Frame Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	49
5.16	Heatmap Visualization of Average CPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower the color, better.	50
5.17	Heatmap Visualization of Average GPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower the color, is better.	51
5.18	Heatmap Visualization of Average FPS between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher the color, is better.	52
5.19	Heatmap Visualization of Average Frame Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower the color, better.	53
5.20	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.	55

5.21	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	55
5.22	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.	56
5.23	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	56
5.24	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.	57
5.25	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	57
5.26	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.	58
5.27	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	58
5.28	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.	59
5.29	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	59
5.30	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.	60
5.31	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	60

5.32	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Virtual Reality (VR)	62
5.33	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Virtual Reality (VR)	63
5.34	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Virtual Reality (VR)	63
5.35	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Virtual Reality (VR)	64
5.36	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Virtual Reality (VR)	64
5.37	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Virtual Reality (VR)	65
5.38	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Virtual Reality (VR)	65
5.39	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Virtual Reality (VR)	66
5.40	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Virtual Reality (VR)	66
5.41	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Virtual Reality (VR)	67
5.42	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Virtual Reality (VR)	67
5.43	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Virtual Reality (VR)	68
5.44	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 1, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).	68

5.45	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 2, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).	69
5.46	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 3, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).	69
5.47	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 4, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).	70
5.48	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 5, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).	70
5.49	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 6, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).	71
5.50	Comparison of Average CPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	72
5.51	Comparison of Average GPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	73
5.52	Comparison of Average FPS between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.	73
5.53	Comparison of Average Frame Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	74
5.54	Heatmap Visualization of Average CPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower the color, better.	75
5.55	Heatmap Visualization of Average GPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower the color, is better.	76

5.56	Heatmap Visualization of Average FPS between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher the color, is better.	77
5.57	Heatmap Visualization of Average Frame Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower the color, better.	78
5.58	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.	79
5.59	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	80
5.60	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.	80
5.61	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	81
5.62	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.	81
5.63	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	82
5.64	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.	82
5.65	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	83
5.66	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.	83

5.67	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	84
5.68	Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.	84
5.69	Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	85
5.70	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 1, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	85
5.71	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 2, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	86
5.72	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 3, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	86
5.73	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 4, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	87
5.74	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 5, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	87
5.75	Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 6, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.	88
5.76	Qualitative evaluation of ControlXR and WasmXRGPU in VR mode across multiple criteria.	89
5.77	Participant preference of ControlXR and WasmXRGPU in VR mode.	90
5.78	Qualitative evaluation of ControlXR and WasmXRGPU in VR mode across multiple criteria.	92
5.79	Participant preference of ControlXR and WasmXRGPU in AR mode.	93
A.1	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	118

A.2	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	119
A.3	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	119
A.4	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	120
A.5	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	120
A.6	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	121
A.7	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	122
A.8	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	122
A.9	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	123
A.10	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	123
A.11	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	124
A.12	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.	124
A.13	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Virtual Reality (VR)	125
A.14	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Virtual Reality (VR)	125

A.15	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Virtual Reality (VR)	126
A.16	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Virtual Reality (VR)	126
A.17	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Virtual Reality (VR)	127
A.18	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Virtual Reality (VR)	127
A.19	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Virtual Reality (VR)	128
A.20	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Virtual Reality (VR)	128
A.21	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Virtual Reality (VR)	129
A.22	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Virtual Reality (VR)	129
A.23	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Virtual Reality (VR)	130
A.24	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Virtual Reality (VR)	130
A.25	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	131
A.26	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	132
A.27	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	132

A.28	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	133
A.29	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	133
A.30	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	134
A.31	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	135
A.32	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	135
A.33	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	136
A.34	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	136
A.35	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	137
A.36	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better. . .	137
A.37	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Augmented Reality (AR)	138
A.38	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Augmented Reality (AR)	138
A.39	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Augmented Reality (AR)	139
A.40	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Augmented Reality (AR)	139

A.41	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Augmented Reality (AR)	140
A.42	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Augmented Reality (AR)	140
A.43	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Augmented Reality (AR)	141
A.44	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Augmented Reality (AR)	141
A.45	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Augmented Reality (AR)	142
A.46	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Augmented Reality (AR)	142
A.47	Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Augmented Reality (AR)	143
A.48	Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Augmented Reality (AR)	143

List of Tables

2.1	Analysis of surveyed work with respect to the technologies that they have used	14
4.1	Chrome hardware support for various devices in WebXR Device API (<i>"WebXR Device API - Chrome Hardware Support"</i> 2025) . . .	30
5.1	Polygon and draw call count for each scenario when using a single high poly 3D model	37
5.2	Polygon and draw call count for each scenario when using multiple instances of two 3D models	41
5.3	Polygon count for each scenario	44
5.4	Selected devices and the web browser details	45
5.5	Qualitative evaluation participant preference and justification - Virtual Reality (VR)	91
5.6	Qualitative evaluation participant preference and justification - Augmented Reality (AR)	94
6.1	ANOVA results for performance metrics when multiple instances of single high poly 3D model are used.	96
6.2	ANOVA results for performance metrics when multiple instances of two 3D models are used.	96

List of Acronyms

API Application Programming Interface. 2, 3, 7, 8, 18, 20–26, 28–35, 37, 39, 40, 88, 91–93, 95–101

AR Augmented Reality. 2, 6–9, 13–15, 61, 71–87, 90, 93–96, 98, 119–135

CPU Central Processing Unit. 2, 3, 11, 15, 17, 18, 26, 27, 34, 37, 43, 45, 46, 50, 70, 71, 75, 90, 91, 93, 94, 98, 99, 102, 104–106, 111, 112, 114, 115, 117, 118, 120–122, 127, 128, 130, 131, 133, 134

FBX Filmbox. 29

FPS Framerate Per Second. 14, 18, 34, 35, 37, 45–48, 54–59, 61–66, 70–73, 77–83, 89, 91, 92, 94, 95

GLSL OpenGL Shading Language. 27

gLTF Graphics Library Transmission Format. 29

GPU Graphics Processing Unit. 2, 3, 7, 8, 11, 12, 14, 15, 17, 18, 21, 24, 26–29, 32, 34, 35, 45–47, 51, 70, 72, 76, 90, 91, 93, 94, 98–101, 107–111, 113, 114, 116, 117, 119, 123–127, 129, 130, 132, 133, 135

JSON JavaScript Object Notation. 29

MR Mixed Reality. 6

VR Virtual Reality. 2, 6–9, 13, 15, 26, 45–52, 54–59, 61–71, 84, 90, 93–96, 98, 99, 103–119

W3C World Wide Web Consortium. 20

WGSL WebGPU Shading Language. 27

XR Extended Reality. 2, 3, 6–10, 13, 15–18, 20, 23, 31–35, 43, 60, 88, 91–93, 96–102

Chapter 1

Introduction

Technological advancements have made significant strides in recent years, particularly in the field of immersive technologies, which have had a profound impact on a wide array of sectors. One such breakthrough is Extended Reality (XR), a term that encompasses a range of immersive experiences designed to alter or enhance the user's perception of reality. XR serves as an umbrella term that includes three distinct components: Virtual Reality (VR), Mixed Reality (MR), and Augmented Reality (AR) (Mendoza-Ramírez et al. 2023). These technologies differ in their approach to user interaction and immersion. VR creates fully virtual worlds that replace the real world entirely, immersing the user in a completely digital environment. AR, on the other hand, overlays digital content onto the user's real-world view, enhancing their interaction with their surroundings. MR sits between these two extremes, blending the physical and digital worlds by allowing users to interact with both virtual and real-world objects simultaneously (Mendoza-Ramírez et al. 2023).

As immersive technologies like VR and AR continue to advance, they have found application in a variety of fields including entertainment, education, healthcare, architecture, gaming, and more. While AR has already gained significant traction, VR is also beginning to grow in prominence, particularly in gaming, training simulations, and virtual tourism (Carmigniani & Furht 2011). Both VR and AR hold great promise for reshaping the way people interact with digital content, creating new opportunities for innovation and engagement.

1.1 Problem Statement

The rapid evolution of XR technologies has been matched by the growth in their use on mobile devices, which are increasingly equipped with the hardware capa-

bilities necessary for delivering compelling immersive experiences (Carmigniani & Furht 2011). Traditionally, XR applications—whether AR or VR—have been distributed through app stores, requiring users to download and install platform-specific apps. However, this model has become less appealing to both users and developers. Users often face the inconvenience of needing multiple apps for different use cases, while developers are forced to create separate versions of their applications for each platform, leading to increased development time and cost (Qiao et al. 2019). Web browser-based applications offer a compelling alternative, as they are accessible across platforms without the need for installation, making them more user-friendly and less restrictive for developers. However, despite the advantages of cross-platform accessibility, web-based XR applications currently lag behind their native counterparts in terms of performance (Qiao et al. 2019).

Web applications are typically constrained by the limitations of the web browser environment, which adds layers of abstraction between the application and the device hardware. While native applications have direct access to the Graphics Processing Unit (GPU) and other hardware resources, web applications must work within the constraints of the browser, which impacts their ability to deliver high-performance immersive experiences. This discrepancy is particularly evident in resource-intensive tasks such as real-time 3D rendering for XR applications. Web browsers, being general-purpose tools designed for a wide range of tasks, are not optimized for the specific needs of hardware-intensive applications like XR. As a result, developers face challenges in delivering seamless, high-performance experiences for users, particularly in the context of mobile and web-based XR (Hamzaturrazak et al. 2023).

1.2 New Technologies and Solutions

A key component of immersive XR applications is the ability to render 3D objects accurately and in real-time, seamlessly blending virtual content with the physical world. This requires leveraging the full potential of the device’s graphics hardware, specifically the Graphics Processing Unit (GPU). For many years, OpenGL (*OpenGL* 1997) has been the standard graphics API for rendering 3D content, but it has limitations, particularly when working with modern hardware and large-scale immersive applications. To address these challenges, new, more efficient graphics APIs such as Vulkan (*Vulkan* 2016) and DirectX (*DirectX by Microsoft* 1995) have been developed. These APIs offer low-level access to the GPU, providing developers with greater control over hardware resources for enhanced performance. Apple’s Metal API (*Metal by Apple* 2014) further adds to

the landscape of platform-specific graphics APIs.

For web-based applications, WebGL (*WebGL* 2011) has traditionally been the go-to solution for rendering 3D graphics in the browser. While WebGL has been instrumental in bringing 3D content to the web, its capabilities are limited when compared to more modern APIs like Vulkan and Metal. WebGPU (*WebGPU* 2024) was developed as the next-generation graphics API for the web, designed to address the limitations of WebGL. WebGPU provides low-level, high-performance access to the GPU, enabling developers to create more sophisticated and efficient 3D graphics and computation pipelines. This makes WebGPU a critical component for improving the performance of web-based XR applications, offering closer-to-native performance for immersive experiences.

In addition to WebGPU, WebAssembly (*WebAssembly* 2017) plays a key role in optimizing the performance of web-based applications. By allowing developers to compile code written in languages like C, C++, and Rust into efficient, portable binary formats, WebAssembly enables near-native execution speeds for web applications. This capability is especially important for performance-sensitive tasks such as 3D rendering and real-time physics simulations, which are central to XR applications.

Finally, the WebXR Device API (*WebXR Device API* 2024) serves as a unified standard for building immersive XR experiences across different devices, including smartphones, headsets, and other specialized hardware. WebXR simplifies the process of creating cross-platform XR applications by providing a consistent API for both VR and AR experiences. This allows developers to create applications that work seamlessly across a wide range of devices, bridging the gap between different hardware platforms.

1.3 Research Motivation and Objectives

This research aims to explore how modern web technologies, particularly WebGPU, WebAssembly, and the WebXR Device API, can be leveraged to improve the performance and user experience of web-based XR applications. While native applications have long been considered the benchmark for performance in XR, recent advancements in web technologies offer promising opportunities to narrow the performance gap. By investigating the integration of WebGPU, WebAssembly, and WebXR, this research seeks to contribute to the development of high-performance, cross-platform XR applications that offer a seamless experience for users, whether they are interacting with virtual or augmented content.

1.4 Scope and Limitations

This research investigates the performance of web-based Extended Reality (XR) applications developed using WebAssembly and WebGPU. While the study aims to provide insights into the effectiveness and viability of these technologies in supporting immersive web-based XR experiences, the following scope and limitations apply:

Firstly, this study does not incorporate cloud servers or edge computing approaches for performance enhancement. Although previous studies such as Qiao et al. (2019) suggest that offloading computation to the cloud or edge servers may improve the performance of web-based XR applications, this research focuses solely on client-side processing. For instance, 3D models and other assets are rendered and managed locally within the browser environment without relying on remote servers.

Secondly, although dedicated XR devices such as AR glasses or VR headsets are increasingly becoming popular and specialized for immersive experiences (Mendoza-Ramírez et al. 2023), this study does not aim to cover the full range of such devices. Instead, the focus is on commonly available XR-compatible hardware. Experiments were conducted using a limited set of XR devices, but the generalization of results to a wide variety of dedicated XR hardware platforms is not within the scope of this research.

Lastly, this research does not investigate performance improvements for specific XR-related tasks such as 3D pose estimation or lighting estimation. As discussed in Bi et al. (2023), such tasks are essential for creating rich and immersive XR experiences, but often require substantial computational resources. Current web-based XR frameworks may not allocate enough time for accurate execution of these tasks. Therefore, this study evaluates performance improvements at a general level, focusing on the overall system responsiveness and rendering performance, rather than on enhancements to these individual computational tasks.

1.5 Research Aim, Questions and Objectives

1.5.1 Research Aim

It is evident that the use of emerging web technologies such as WebAssembly and WebGPU has led to notable improvements in the performance of immersive web applications. Thus, the main aim of this study is stated as follows:

To investigate the performance of web-based XR applications that are powered by WebAssembly and WebGPU

1.5.2 Research Questions

1. What are the suitable approaches to implement web-based XR applications using WebGPU and WebAssembly?
 - This question seeks to determine how WebAssembly and WebGPU can be effectively integrated in the context of WebXR. The answer will identify an approach that leverages the strengths of both technologies to support immersive XR experiences in the browser.
2. How can web-based XR applications be implemented using WebAssembly and WebGPU following the most suitable approach?
 - Once a suitable integration strategy is identified, the practical implementation of WebXR applications using that approach will be explored and demonstrated.
3. How do web-based XR applications perform when implemented using WebGPU and WebAssembly?
 - This question evaluates the performance of the implemented WebXR application in comparison to traditional WebXR implementations (e.g., using WebGL), focusing on key performance metrics such as frame rate, responsiveness, and rendering quality.

1.5.3 Research Objectives

To address the research questions and accomplish the overall aim of the study, the following research objectives are defined:

1. To identify the challenges of using WebAssembly and WebGPU together for implementing web-based XR applications
2. To implement a web-based XR application using WebAssembly and WebGPU
3. To assess the performance of web-based XR applications implemented using WebAssembly and WebGPU

1.6 Key Terms and Concepts

This section defines several technical terms that are used throughout the thesis to ensure clarity and consistency.

Framebuffer A memory buffer that stores the final rendered image before it is displayed on the screen. It holds color, depth, and stencil information required for rendering a frame.

Draw Call A command issued by the CPU to instruct the GPU to render a set of primitives (such as triangles or lines) using specified resources. Each draw call typically includes:

- Vertex buffer(s) – containing the geometry data such as positions, normals, UVs, etc.
- Index buffer (optional) – defining how vertices are reused to form primitives.
- Pipeline state – including the shader programs, blending, depth, rasterization, and other GPU state settings.
- Bind groups or descriptor sets – providing access to textures, uniform buffers, and other GPU resources.
- Number of vertices/indices to draw – determining the size of the draw.
- Instance count (for instanced rendering) – if applicable.

Render Bundle In WebGPU, a render bundle is a pre-recorded set of rendering commands that can be efficiently reused in a render pass. It helps reduce CPU overhead and improve rendering performance by avoiding redundant state setup.

Compute Shader A programmable shader stage that runs general-purpose computations on the GPU. Unlike vertex or fragment shaders, compute shaders are not bound to the graphics pipeline and are ideal for parallel computations like physics, lighting, or image processing.

Viewport The rectangular area of the window where the final image is rendered. It defines how normalized device coordinates (from -1 to 1) map to screen coordinates.

Projection Matrix A matrix that transforms 3D coordinates into a 2D representation for display on the screen. It handles perspective distortion to give the illusion of depth.

View Matrix A matrix that represents the camera's position and orientation in the scene. It transforms world coordinates into camera (view) coordinates.

Model Matrix A matrix that transforms object coordinates (also known as local coordinates) into world coordinates. It is used to position, rotate, and scale an object within a scene.

Uniform A uniform is a variable in shaders that holds constant data, like transformation matrices or lighting settings, which doesn't change during a single draw call.

Uniform Bind Group A uniform bind group is a collection of uniforms that are bound together and used by a shader during rendering, allowing efficient management of shader inputs.

GPU Buffer A GPU buffer is a chunk of memory allocated on the GPU used to store data, such as vertex positions, indices, or other resources, that the GPU can access during rendering or computation tasks.

Chapter 2

Literature Review

2.1 History and Problem

The technology of simulation dates back to the 1920s, originally developed to mimic real-life flight experiences. Over time, these systems evolved and found applications in numerous domains. In 1989, Jaron Lanier coined the term Virtual Reality (VR) to describe immersive, computer-generated environments (Zhao 2009). As computing power advanced, particularly with the rise of portable devices such as smartphones, Zhao (2009) also highlighted their growing relevance in VR applications, referring to them as hand-held displays capable of supporting intensive computations.

With the expansion of web technologies, the idea of delivering VR experiences through the browser gave rise to web-based VR. Neelakantam & Pant (2017) gave an introduction to the WebVR API, outlining the potential of implementing VR directly on the web platform. Building upon this, Toasa G et al. (2019) evaluated the performance of WebVR and WebGL across different devices and browsers. Their findings indicate that while both perform comparably in terms of Frames Per Second (FPS) on desktop systems, WebVR has an advantage in mobile environments due to lower VRAM usage and more stable frame rates—making it more suitable for mobile-based 3D VR content. However, the WebVR API has since been deprecated (*WebVR* 2024), paving the way for more modern APIs like WebXR.

According to Lee (2012), the term "Augmented Reality (AR)" was given by a researcher named Thomas Caudell back in 1990. However, Carmigniani & Furht (2011) acknowledges that AR technology dates back to the 1950s. The work of Azuma (1997) serves as one of the earliest surveys for AR. While both of

these surveys discuss specific devices catered toward AR, Carmigniani & Furht (2011) addresses that mobile device-specific applications in AR are worthy of being studied as mobile devices are starting to become an integral part of people’s lives at that time. Following that, Qiao et al. (2019) surveys the literature based on mobile AR and discusses what the capabilities are when mobile AR meets the web. They mention the 5G network speed, computation outsourcing to the cloud, and edge computing as some of the methods to improve the performance of WebAR. Moreover, they also list down various web technologies that enable the possibility of implementing AR on web, such as WebRTC (*WebRTC* 2018), WebAssembly (*WebAssembly* 2017), Web Workers (MozDevNet 2023), and WebGL (*WebGL* 2011). Apart from this, McNally & Koviland (2024) reviews some of the web-based AR platforms and their features with respect to the different web browsers, different devices, and different platforms. Yet they do not assess the performance capabilities.

2.2 WebXR

The WebXR Device API (*WebXR Device API* 2024), provided by modern browser vendors, enables the creation of immersive and interactive sessions for both Virtual Reality (VR) and Augmented Reality (AR). Leveraging this capability, Ferrão et al. (2023) proposed a framework aimed at enhancing the efficiency of WebAR development by exploiting the interactive features offered by WebXR. Specifically, they have used “depth”, “lighting-estimation”, “hit-test”, “anchors”, and “geo-alignment” features to implement shadow casting, physics simulation, occlusion, etc. Their work evaluates performance primarily through frame time metrics and compares the results with other WebAR frameworks such as MyWebAR and DepthLab (Du et al. 2020). However, they also highlight a significant limitation: the tight coupling between WebGL rendering textures and the WebXR Device API introduces challenges for implementing real-time reflections, restricting rendering flexibility in Web-based AR/VR applications.

Another noteworthy advancement in WebAR research is the integration of massive 3D point clouds with semantics into web-based, marker-less mobile AR applications. (Kharroubi et al. 2020) propose a system that supports real-time visualization and interaction with point clouds comprising up to 29 million points, achieving frame rates between 27 and 60 FPS on mobile browsers. Their approach involves a hierarchical structuring of the point cloud data using an adapted version of Potree’s Octree, allowing the system to operate within a manageable point budget while maintaining interactivity. The study also addresses key challenges

in mobile WebAR, including network dependency and battery consumption, especially as sensor usage remains continuous during the experience. Their findings highlight that the system’s responsiveness depends more on the number of displayed points rather than the total dataset size, suggesting scalability for even larger point clouds.

Bi et al. (2023) addresses the performance of WebXR in their work by comparing a few of the popular JavaScript frameworks used for creating XR experiences on the web. To evaluate, they have also created a separate tool to capture the results in the web browser, which is currently publicly accessible. Bi et al. (2023) shows that each framework has its own strengths and weaknesses, but there is no framework that dominates in all aspects. Because of this, they give implications catered towards different stakeholders, such as WebXR developers, web browser developers, and XR framework developers. Bi et al. (2023)’s one of the important findings shows that the current frameworks are incapable of fulfilling the real-world understanding task as the rendering and the camera capture take the most of the time to have interactive FPS.

Taking the rendering process of web-based AR applications into account, Hamzaturrazak et al. (2023)’s work compares WebGL and OpenGL shading languages. Their motivation for this study is to show that WebGL and WebAR applications in general have limited capabilities to access the hardware compared to native applications, and they try to investigate this by using OpenGL shader and WebGL shader and comparing their rendering performance. Hamzaturrazak et al. (2023) states that WebGL consists of two shaders, namely WebGL raw shader and GLSL shader. By using these two shaders, they have obtained a result indicating that GLSL shader performs better when the rendering mesh consists of a higher number of polygons. They further state that rendering low-poly and medium-poly meshes didn’t indicate differences between the shaders, yet their suggestion for this observation is the limitation of the hardware that they have used. They conclude by stating that GLSL shader is the recommended rendering shader for web based AR applications.

2.3 WebGPU

WebGL has been the standard for rendering 3D objects in the web browser (*WebGL* 2011). With the release of its successor, WebGPU (*WebGPU* 2024) is said to have better performance than WebGL due to it’s capability of accessing the device’s GPU directly. Usta (2024) studies the performance of WebGPU in the context of web-based geographic information systems. Since it also considers the

rendering aspect of the web, this study’s results are relevant. Here, Usta (2024) shows their results, indicating that WebGPU is at-most three times more performant than WebGL. To obtain such results, they rendered some number of objects using both of the technologies and kept increasing that number, which ended up increasing the number of vertices. Their reasoning for the improvement is that WebGPU’s capabilities include direct access to the GPU, parallelism, and multi-threading. An important aspect of this study is that they haven’t used available JavaScript frameworks that include WebGPU. Usta (2024) states that it is unfair to compare using the frameworks as WebGL is more mature compared to WebGPU within those frameworks. Another study done by Chickerur et al. (2024) shows how WebGPU performs better than WebGL in a Web 3.0 environment.

However, a recent work by Bi et al. (2024) analyzes the performance of current frameworks when they use WebGPU. Their findings have shown that WebGL implementations give better performance than WebGPU implementations. From these findings, they conclude that the frameworks have not utilized WebGPU’s capabilities correctly. Thus, they introduce their own system called **FusionRender** that overcomes the previously found limitations. In their proposed system, it identifies 3D objects that can be rendered together, so those can be rendered using a single draw call, thus optimizing the performance. Furthermore, they have incorporated this system with three.js (*Three.js* 2010) to use some graphical rendering components such as cameras and lighting effects. Their results show that their proposed system outperforms both WebGL implementations and current WebGPU implementations of the frameworks. But Bi et al. (2024) states that the performance difference can vary if the rendering scene does not contain a large amount of 3D objects. This is because having a small number of 3D objects means there are fewer to merge, so the number of draw calls will not differ that significantly. Furthermore, it is also worth noting that this study focuses purely on rendering 3D graphics on the web.

2.4 WebAssembly

Khomtchouk (2021) also addresses the issue of web-based AR/VR applications being less performant than their native counterparts. Their argument for this issue is the use of JavaScript in web applications, which is a Just-In-Time compiling language. They state that using an Ahead-Of-Time compiling language can cause better performance. Therefore, they propose to use WebAssembly (*WebAssembly* 2017) that acts as a compilation target. They compared the performance of WebAssembly with JavaScript and asm.js (*Asm.js Specification* 2011) by im-

plementing an algorithm. Khomtchouk (2021)’s results show that WebAssembly implementation is twenty times faster than JavaScript and two times faster than asm.js. Hence, they recommend integrating WebAssembly into WebXR libraries and standardizing it by introducing a WebXR framework that is based on WebAssembly.

Liu et al. (2023) in their work compares the performance of WebAssembly to JavaScript in the context of WebXR. To conduct the experiment, they have used Magnum Engine (*Magnum Engine* 2022) and OpenCV (*OpenCV* 2000) to write the code and cross-compile using Emscripten (*Emscripten* 2015) to port the code into JavaScript and WebAssembly. Based on these implementations, their results show that WebAssembly has major improvements over JavaScript in various aspects such as Page load Time, Object Tracking and slightly on Central Processing Unit (CPU) and GPU utilization (Liu et al. 2023). However, they note that the memory utilization in WebAssembly falls short behind JavaScript as it can dynamically handle the memory allocations, which is not done by WebAssembly automatically. They conclude that WebAssembly performs better than JavaScript, but it has more room for improvement in closing the gap between native applications and web-based applications. Finally, Liu et al. (2023) state that WebGPU (*WebGPU* 2024) should be used since GPU is used extensively in XR applications. They indicate that accelerating and optimizing the performance of WebGPU by integrating with WebAssembly is their future work.

2.5 WebAssembly and WebGPU

The combined use of WebAssembly and WebGPU has been explored across various performance-critical domains. For instance, Erazo & Demir (2023) introduced a novel computational library targeting hydrology and environmental sciences that leverages both WebAssembly and WebGPU to enable high-performance scientific simulations directly in the browser. Similarly, Ammann et al. (2022) developed a map rendering library as a proof-of-concept, demonstrating how these technologies can enhance rendering performance and responsiveness for interactive geospatial applications.

Image processing represents another domain that benefits significantly from this integration. Nam et al. (2024) showed that combining WebAssembly with WebGPU can yield execution speeds up to ten times faster than traditional JavaScript implementations, due to the efficient utilization of lower-level programming languages and parallelized GPU computation. In the context of AI, Odume et al. (2024) investigated the integration of WebAssembly and WebGPU for running

machine learning models directly in web applications. Their results indicate that WebGPU accelerates parallelizable tasks such as image classification, while WebAssembly optimizes CPU-bound operations. This synergy enables real-time inference, enhances privacy by reducing server-side dependencies, and lowers operational costs—making it a compelling approach for deploying AI models on the web.

2.6 Critical Analysis of the Literature

A comprehensive survey of recent work in web-based XR and rendering technologies, summarized in Table 2.1, reveals a notable gap in the joint utilization of WebGPU and WebAssembly for implementing immersive XR applications. While various studies have explored WebGL-based XR frameworks (Ferrão et al. 2023, Kharroubi et al. 2020, *8th Wall* 2024), and others have adopted WebAssembly for computational acceleration (Liu et al. 2023, Khomtchouk 2021, *Wonderland Engine* 2024), none have effectively combined WebGPU’s modern GPU rendering capabilities with WebAssembly’s near-native performance benefits in the context of WebXR.

Several works have demonstrated the potential of WebGPU and WebAssembly for general-purpose rendering and computation on the web, especially in domains like GIS visualization (Usta 2024), machine learning (Odume et al. 2024), and image processing (Nam et al. 2024), but these are not targeted towards WebXR use cases. Importantly, Liu et al. (2023) explicitly identify the need for further exploration of WebGPU and WebAssembly in XR contexts as future work, signaling an open challenge in the field.

This lack of integration highlights a crucial gap: despite both technologies being highly promising for delivering performant and interactive experiences in web environments, they have not yet been jointly explored within immersive WebXR applications.

This identified gap directly informs the research questions outlined in Section 1.5.2, each of which is detailed below to frame the core investigative focus of this thesis:

- The first research question asks:
What are the suitable approaches to implement web-based XR applications using WebGPU and WebAssembly?
 - This seeks to address the methodological vacuum revealed by the current literature, where integration patterns between these technologies

Work	Context	Technologies	WebGPU	WebAssembly
Hamzaturrazak et al. (2023)	WebAR	WebGL, OpenGL	✗	✗
Liu et al. (2023)	WebXR	WebGL, WebAssembly	✗	✓
Ferrão et al. (2023)	WebAR Framework	WebGL (Three.js)	✗	✗
Kharroubi et al. (2020)	WebAR Point Cloud Intergration	WebGL (Three.js)	✗	✗
Liu et al. (2023)	WebXR	WebGL, WebAssembly	✗	✓
Bi et al. (2024)	3D rendering on web	WebGPU, WebGL	✓	✗
Usta (2024)	WebGIS	WebGPU, WebGL	✓	✗
Chickerur et al. (2024)	Rendering on Web 3.0	WebGPU, WebGL	✓	✗
Bi et al. (2023)’s review	WebXR	WebGL, JavaScript Frameworks	✗	✗
Khomtchouk (2021)’s review	WebAR, WebVR	WebAssembly, asm.js, JavaScript	✗	✓
<i>Wonderland Engine</i> (2024)	Engine for WebXR development	WebGL, WebAssembly	✗	✓
<i>8th Wall</i> (2024)	WebXR	JavaScript, WebGL	✗	✗
Ammann et al. (2022)	Map Renderer	WebGPU, WebAssembly,	✓	✓
Erazo & Demir (2023)	Computational Library	WebGPU, WebAssembly,	✓	✓
Nam et al. (2024)	Image Processing on Web	WebGPU, WebAssembly,	✓	✓
Odume et al. (2024)	Machine Learning on Web	WebGPU, WebAssembly,	✓	✓

Table 2.1: Analysis of surveyed work with respect to the technologies that they have used

in XR contexts remain largely undocumented.

- The second question follows up with:
How can web-based XR applications be implemented using WebAssembly and WebGPU following the most suitable approach?
 - This reflects the implementation challenge that arises from the lack of real-world systems combining these technologies in WebXR pipelines.
- The third question addresses:
How do web-based XR applications perform when implemented using WebGPU and WebAssembly?
 - Since no prior work evaluates the performance of such a combination in XR, this study contributes novel empirical findings, especially regarding CPU/GPU times, frame rates, and draw call behavior in hybrid rendering pipelines.

Thus, the absence of prior work combining WebGPU and WebAssembly in WebXR, alongside the explicit need for such research mentioned in existing studies, serves as a clear and compelling justification for this thesis.

Chapter 3

Methodology

3.1 Research Methodology

This study adopts a Design Science Research (DSR) methodology, supported by empirical experimentation. As illustrated in Figure 3.1, the central problem is identified as the performance gap between native and web-based Extended Reality (XR) applications. This gap can hinder the widespread adoption of WebXR experiences due to latency, reduced frame rates, or limited feature support.

To address this, the research proposes the use of WebGPU and WebAssembly as an alternative rendering and execution pipeline to improve the performance of web-based XR applications. Within the DSR process, the proposed combination serves as an artifact, which is iteratively designed, developed, and evaluated.

This iterative approach allows exploration of different integration methods between WebGPU and WebAssembly, assessing their effectiveness in delivering smooth, high-performance XR experiences. The final outcome is expected to provide practical guidance for developers, while also contributing new insights for researchers and browser vendors. The design and results of the study are intended to be shared with the wider community of WebXR practitioners.

3.2 Experimental Setup

To implement the methodology and evaluate the proposed solution, the experimental approach is inspired by the comparative study from Hamzaturrazak et al. (2023), adapted for the context of WebXR. The study consists of developing and comparing two distinct XR applications, followed by a detailed performance evaluation.

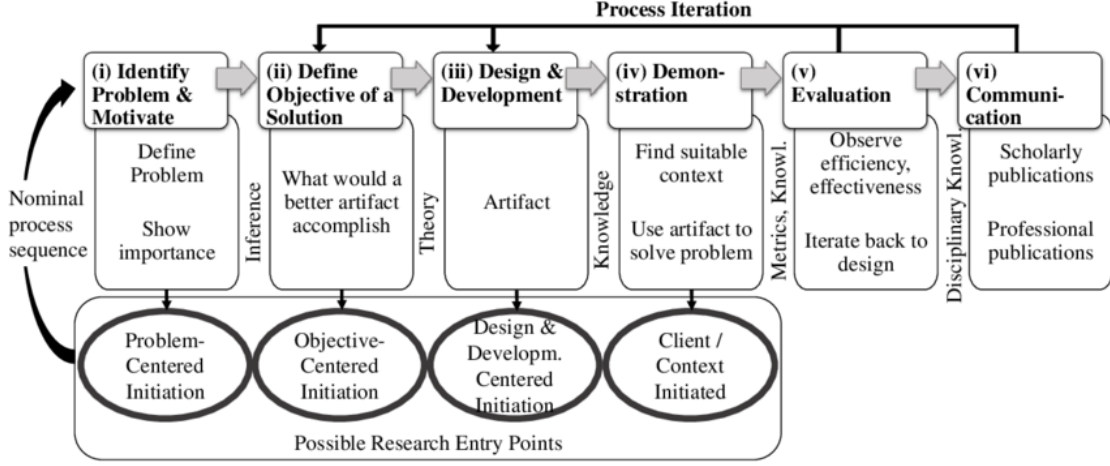


Figure 3.1: Design Science Methodology (Krupitzer 2018)

1. **WebXR Application using WebGL (ControlXR):** An XR application is implemented using WebGL, the conventional standard for rendering 3D graphics in web environments. This application represents the control group, providing a performance baseline for web-based XR experiences using traditional technologies. It is referred to as **ControlXR**.
2. **WebXR Application using WebGPU and WebAssembly (WasmXRGPU):** The proposed solution is implemented by combining **WebGPU** and **WebAssembly** to build an advanced web-based XR application. This approach aims to overcome the limitations of WebGL and JavaScript, and the application is referred to as **WasmXRGPU**.
3. **Performance Evaluation and Comparison:** All three implementations are evaluated in terms of key performance metrics such as frame rate, rendering time, CPU and GPU times. The findings are analyzed to assess the feasibility, advantages, and limitations of the WebGPU and WebAssembly approach for delivering practical WebXR experiences.

This experimental setup enables a structured and measurable comparison of rendering pipelines across web environments, ensuring that the research conclusions are evidence-based and practically relevant.

3.3 Evaluation Plan

To determine the effectiveness and practicality of the proposed WebXR solution, the evaluation consists of both quantitative and qualitative assessments. These evaluations are inspired by the methodologies and metrics used in prior work, particularly Liu et al. (2023), Bi et al. (2023), and Bi et al. (2024).

3.3.1 Quantitative Evaluation

Quantitative evaluation focuses on measuring system performance using technical metrics commonly used in the context of WebXR and web rendering. Based on the literature, the following performance metrics will be considered:

- **Framerate Per Second (FPS)**

- FPS is a direct indicator of rendering performance. Since rendering complexity and the number of 3D objects significantly influence FPS, experiments in this study will involve variations in both factors to evaluate performance consistency. In the work by Bi et al. (2023), FPS is a primary metric used to assess the performance of different frameworks. They measure it over a one-minute duration and compute the average to represent the framework’s rendering efficiency.

- **Frame Time**

- Frame time represents the duration needed to render a single frame. It gives insight into how rendering load is distributed between the CPU and GPU. Bi et al. (2024) puts an emphasis into this metric because as they quote “The FPS matches the screen’s refresh rate when sufficient resources are available. Framework differences can be observed through frame time when FPS remains constant for scenes with less complexity.”

- **Session Load Time**

- In this study, performance metrics are recorded starting from the moment the user explicitly initiates an XR session via an event. This design choice stems from a restriction imposed by the WebXR Device API, which mandates user consent before initiating any XR session. While previous works such as Liu et al. (2023) consider page load time—measuring the duration from page request to first frame render including asset decoding and model preparation—this metric does not fully reflect startup performance in an immersive context. Instead, session load time is adopted as a more representative metric in this research, capturing the time from session request to the rendering of the first immersive frame. This better aligns with user-perceived performance in WebXR applications, where the immersive experience only begins once the session is explicitly started.

- **CPU and GPU Times**

- Since WebGPU provides lower-level access to the GPU, analyzing the

distribution of computational load between the CPU and GPU offers valuable insights into performance characteristics and potential bottlenecks. Additionally, as WebAssembly code is executed on the CPU, measuring CPU performance becomes crucial for a comprehensive evaluation. Liu et al. (2023) examine these metrics to compare the performance of JavaScript and WebAssembly and to understand how execution time is distributed over different phases. Moreover, different frameworks and libraries adopt varied optimization strategies, resulting in distinct low-level instructions executed on the CPU and GPU. To investigate these differences, Bi et al. (2023) measure and visualize CPU and GPU usage over time and under varying scene complexities, enabling deeper insights into how different frameworks utilize hardware resources.

These metrics will be evaluated across the following experimental dimensions:

- Rendering scenarios with varying levels of complexity
- A range of devices with diverse hardware specifications

Due to platform limitations on Apple/iOS devices—particularly restricted access to WebXR Device API—this study will focus solely on Android-based devices. Additionally, Google Chrome will be used as the primary web browser, as it offers the most up-to-date support for the relevant WebXR and WebGPU.

3.3.2 Qualitative Evaluation

While quantitative results reveal system-level performance, the user experience provides insights into practical usability. To complement the technical evaluation, a qualitative study will be conducted involving human participants. Each participant will interact with all two implementations: ControlXR and WasmXRGPU.

Participants will be asked to respond to the following questions, rating each on a scale from 1 (very poor) to 10 (excellent), with justification:

- How do you rate the **responsiveness** of the *implementation*?
- How do you rate the **accuracy and precision of the virtual objects** in the *implementation*?
- How do you rate the **overall experience** of the *implementation*?
- Which implementation did you prefer overall, and why? How did the *implementation* compare to the other?

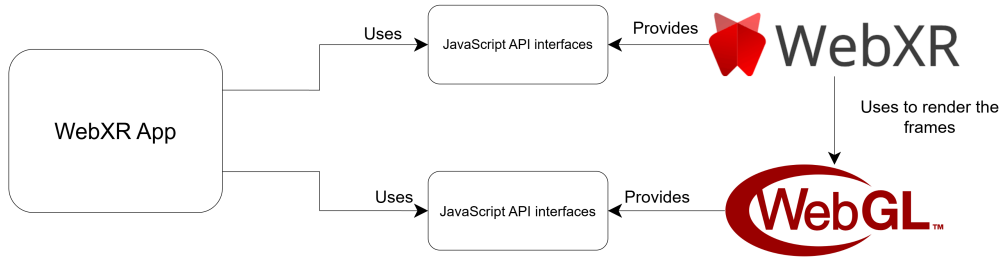


Figure 3.2: WebXR Device API with WebGL to create a traditional WebXR app. (**ControlXR Design**)

By triangulating quantitative performance metrics with qualitative user feedback, the study aims to present a well-rounded evaluation of the proposed solution’s viability in real-world WebXR scenarios.

3.4 Proposed System Architecture

In the realm of WebXR literature, the WebXR Device API (*WebXR Device API* 2024) is widely used (Liu et al. 2023, Qiao et al. 2019, Khomtchouk 2021, McNally & Koviland 2024, Hamzaturrazak et al. 2023, Lee et al. 2021, MacIntyre & Smith 2018) to leverage XR functionalities in web environments. This is primarily because the WebXR Device API is developed based on specifications provided by the World Wide Web Consortium (W3C) (W3C 2024), ensuring a standardized interface for accessing XR capabilities in web browsers. Furthermore, the API is open-source and actively developed, enabling progressive integration of new features. Given this, using the WebXR Device API to access XR functionalities becomes a natural choice.

However, the API specification clearly states that its role is limited to computing the necessary XR-related tasks—it does not perform any 3D rendering itself. Instead, developers must use the results of these computations to render 3D content using their preferred graphics pipeline. To support this rendering process, the WebXR Device API requires access to a rendering context’s framebuffer. For example, in a WebAR application, the API must access the device’s camera and render the feed within the browser, which necessitates a valid framebuffer from a rendering context. Figure 3.2 depicts how a simple WebXR application works with the use of above stated APIs. This architecture can be followed to implement the control group of the experiment, ControlXR.

Since 3D rendering only depends on the graphics API, it should be trivial to switch

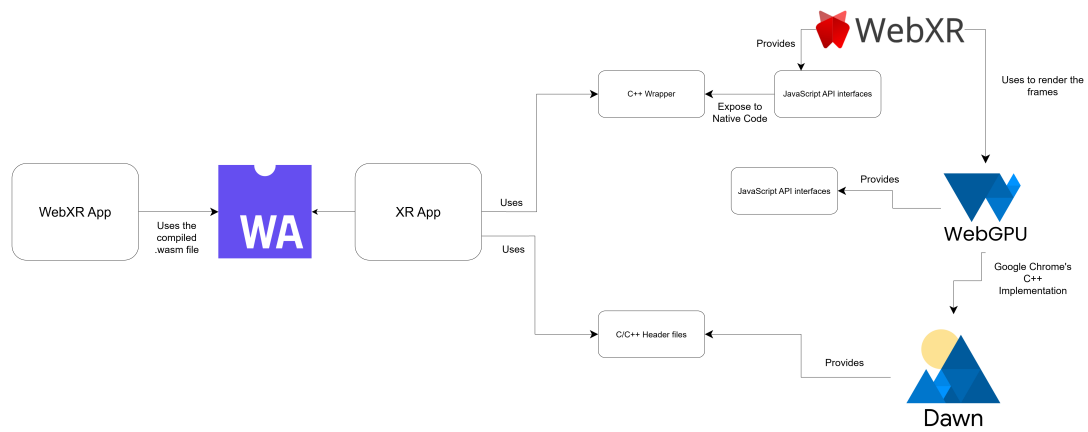


Figure 3.3: WebXR Device API with WebGPU and WebAssembly to create a WebXR App. (**Proposed WasmXRGPU Design**)

WebGL with WebGPU in Figure 3.2 and expose WebXR Device API to C++ and obtain the architecture for WasmXRGPU as depicted in Figure 3.3.

Chapter 4

Implementation

Conducting this research involves investigating the defined research problem and formulating answers to the corresponding research questions. The research objectives serve as fundamental guides that support and direct this process. This study involves two major implementations: **ControlXR**, which serves as the control group, and **WasmXRGPU**, which represents the experimental group.

4.1 WebGPU incompatibility

Currently, the WebXR Device API only supports WebGL-based rendering contexts. While support for WebGPU has been proposed, it has not yet been fully implemented or shipped within the API (*"WebXR/WebGPU Binding Module"* 2025). This limitation poses a challenge for this research, particularly when considering the use of WebGPU as part of the proposed solution.

4.2 Finding a Solution for the Incompatibility

Understanding the distinction between different graphics APIs—namely WebGL and WebGPU—is essential here. Graphics APIs provide the interface through which developers issue rendering commands to the GPU. As shown in Figure 4.1, the graphics rendering pipeline consists of multiple stages. Among these, the vertex and fragment shader stages are programmable, meaning developers can write custom shader code to define how those stages operate. Other stages are considered fixed-function, though they can still be configured to some degree (Bi et al. 2024).

One key distinction is that WebGPU offers finer-grained control and configurability over the pipeline compared to WebGL, making it a more modern, low-level

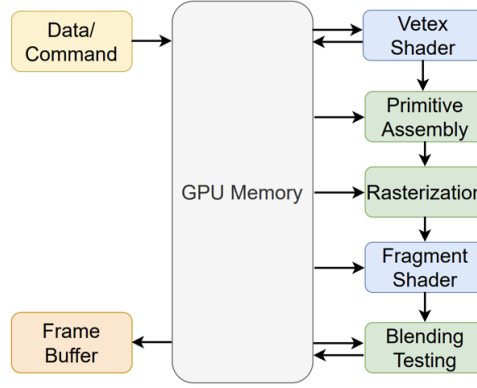


Figure 4.1: Graphics Rendering Pipeline. (Bi et al. 2024)

API. In programmable stages, shaders perform essential tasks such as 3D-to-2D projection (via matrix multiplication), lighting calculations, and color computations. Once these calculations are complete, the final output is stored in the framebuffer.

This leads to a potential workaround for the WebXR limitation discussed earlier: **WebGPU–WebGL interoperability**.

4.3 WebGPU-WebGL Interoperability

WebGPU–WebGL interoperability refers to the approach where both APIs are used in conjunction, each handling specific parts of the rendering pipeline. At first glance, this might seem to undermine the objective of this study—which is to explore the performance benefits of WebGPU. However, it is important to note that the WebXR Device API requires only a WebGL-based framebuffer—that is, the final output of the rendering pipeline. The actual computation, including vertex processing, shading, and rasterization, can still be carried out using WebGPU. The rendered output can then be transferred from WebGPU to the WebGL framebuffer for display within the WebXR pipeline. The process of transferring the texture data from one place to another, can be done using a technique called **”Texture blitting”**.

In this setup, the role of WebGL is reduced to a minimal one, acting only as a passthrough for the rendered frame. Specifically, its responsibilities can be summarized as follows:

1. Create a WebGL texture.
2. Copy the rendered texture/frame from WebGPU to the WebGL’s texture.

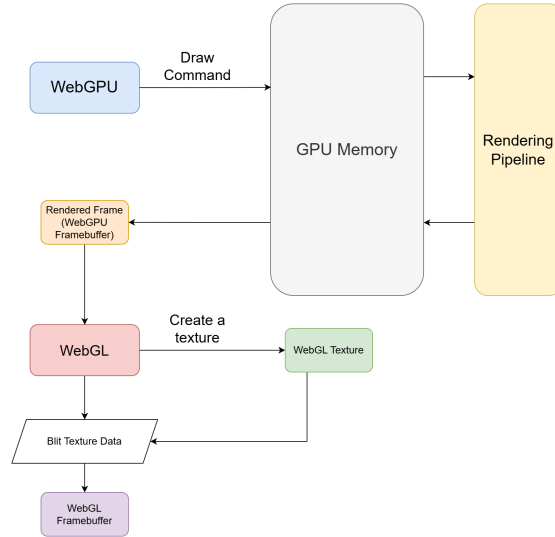


Figure 4.2: WebGPU-WebGL interoperability approach where both graphics APIs work together

3. Upload that texture data to WebGL’s framebuffer.

This interoperability is feasible due to the nature of framebuffers. Ultimately, a framebuffer holds the final rendered output, which is essentially a texture—a 2D image with specific dimensions. Thus, the rendered result from WebGPU can be treated as a regular image or frame and transferred accordingly.

The overall process can be outlined in the following steps and it is depicted in Figure 4.2.

1. Initialize the WebXR Device API with a WebGL rendering context.
2. Initialize an Offscreen Canvas (*OffScreenCanvas* - MDN Web Docs 2024) using a WebGPU context.
3. Use the WebXR Device API to handle XR-specific tasks such as tracking and pose estimation.
4. Pass the results of the WebXR computations to the WebGPU pipeline for rendering.
5. Use WebGL’s `blitFramebuffer` (*BlitFramebuffer* - MDN Web Docs 2024) or a similar method to copy the content from the WebGPU-based offscreen canvas to the WebGL framebuffer.

One significant challenge in the WebGPU-WebGL interoperability approach is the difference in framebuffer coordinate systems. WebGPU uses a left-handed coordinate system, while WebGL operates in a right-handed coordinate system. This can lead to discrepancies when copying frames between the two APIs.

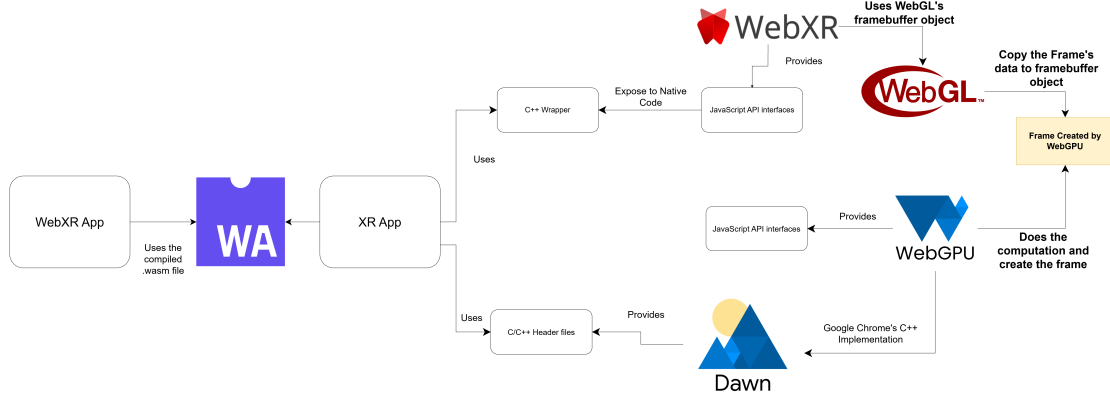


Figure 4.3: WebXR Device API with WebGPU and WebAssembly to create a WebXR App, that uses WebGL to copy WebGPU rendered frames. (**WasmXRGPU Design**)

To address this, the *blitFramebuffer* function is configured to invert the source framebuffer’s vertical axis when transferring the frame to WebGL. This is achieved by swapping the *srcY0* and *srcY1* coordinates in the *blitFramebuffer* call as specified in Listing 4.1.

The inversion ensures that the WebGPU-rendered frame is correctly oriented when displayed by WebGL, maintaining visual consistency across rendering pipelines.

Listing 4.1: Blitting a texture

```
gl.blitFramebuffer(
    // Source coordinates (inverted vertically)
    0, gpuCanvas.height, gpuCanvas.width, 0,
    // Destination coordinates
    0, 0, gpuCanvas.width, gpuCanvas.height,
    gl.COLOR_BUFFER_BIT, gl.NEAREST
);
```

This workaround introduces the possibility of combining the computational advantages of WebGPU with the current constraints of the WebXR Device API, allowing to implementation an app that follows an architecture depicted in Figure 4.3. However, this also raises the question of performance impact due to the interoperation overhead. Therefore, in addition to the main experiments described earlier, a set of preliminary experiments has been conducted to evaluate this hybrid rendering approach. While previous studies have demonstrated WebGPU’s superior performance over WebGL (Chickerur et al. 2024, Usta 2024), they also caution against direct translation of WebGL applications into WebGPU. The performance gains stem not only from the API design differences, but also from how

effectively developers utilize each API’s features to interact with the GPU.

Keeping this in mind, a preliminary experiment is carried out to evaluate the feasibility and performance implications of this WebGPU–WebGL interoperability approach. The results of the experiment are available in Chapter 5 and it’s detailed discussion is available in Chapter 6.

4.4 Performance Considerations

The WebGPU-WebGL interoperability method demonstrated no significant overheads, making it a viable approach for implementing the proposed system architecture. However, its results revealed some noteworthy observations. Specifically, the performance impact caused by the number of draw calls in WebGPU highlights the need to explore appropriate optimization strategies.

4.4.1 Merging Draw Calls

The issue of issuing individual draw calls per object has been addressed by Bi et al. (2024). They suggest that WebGPU’s asynchronous and stateless API design offers greater flexibility compared to WebGL, which relies on a global state model. In their work, they propose an optimization technique that merges draw calls wherever possible to reduce the total number of draw invocations.

By merging draw calls, the following process typically occurs:

1. Objects that use the same pipeline and shader programs are grouped together.
2. Their vertex and index data are combined into two large buffers: one for vertices and the other for indices.
3. A single draw call is then issued, using offsets and counts to access the relevant geometry for each object.
4. Per-object data—such as transformation matrices or material properties—is passed via uniform or storage buffers (or vertex attributes) to distinguish between instances during rendering.

This approach reduces CPU overhead by minimizing the number of state changes and API calls required per frame. It also leverages WebGPU’s efficient buffer and pipeline management to achieve higher performance.

However, the above steps are not API-specific; they can be implemented in both WebGPU and WebGL. Bi et al. (2024) emphasize that current web-based 3D

frameworks and libraries do not fully utilize the capabilities of WebGPU. Their experiments highlight the differences between their approach and the performance of other existing frameworks. Consequently, it is important to investigate how WebGL performs when draw calls are merged as well.

To address this, both graphics backends in the ControlXR and WasmXRGPU implementations include the draw call merging feature. This ensures a fair comparison, minimizing any discrepancies between the two backends.

Moreover, in contrast to (Bi et al. 2024), this study focuses on rendering the same 3D object multiple times. As a result, there won't be different shader programs used to differentiate object groups for separate draw calls. Instead, the entire application creates a single draw call to render all instances of the 3D objects with a single rendering pipeline.

4.4.2 Render Bundles

In WebGPU, a feature called Render Bundles "*WebGPU Specification - GPURenderBundle*" (2025) allows the recording of a set of GPU commands so that the application does not have to re-encode or recreate these commands for every frame. These commands typically include setting the rendering pipeline, binding uniform bind groups, and issuing draw calls for a specific configuration.

Uniform bind groups are associated with GPU buffers. If the buffer itself does not change (meaning the buffer is the same, though its content may change), there is no need to recreate these commands in every frame. Recreating commands each frame introduces unnecessary work for the CPU, as it is responsible for generating the relevant commands that will be sent to the GPU.

WebGPU's Render Bundles address this by bundling the commands for reuse in future frames, reducing the overhead on the CPU. This is particularly beneficial in scenarios like VR, where there are two views per eye. In such contexts, only certain parameters—such as the projection and view matrices and the viewport configurations—need to be updated, while the rest of the rendering commands can remain unchanged.

4.4.3 Compute Shaders

Another powerful feature provided by WebGPU is Compute Shaders ("*WebGPU Specification - GPUComputePipeline*" 2025). Unlike traditional graphics rendering shaders, compute shaders enable general-purpose computations on the GPU—such as large-scale, highly parallel matrix multiplications.

According to the study by Kligge (2024), while WebGL can be used for such general-purpose computing by uploading matrix data to textures, performing computations in fragment shaders, and reading back results from the framebuffer, this approach is suboptimal. WebGPU has demonstrated significantly better performance for these tasks (Kligge 2024).

In a 3D scene, a scene hierarchy is typically represented as a tree. This hierarchical structure allows the transformations of child objects to be influenced by their parent’s transformation. Specifically, the model matrix of a parent node is multiplied with the model matrices of its children to compute their respective world transformations. When the root node represents the world origin, its model matrix must be propagated down to all first-level child objects and recursively through the entire scene graph.

In the experiments conducted for this study, a large number of objects are rendered, resulting in a significant number of matrix multiplications. Initially, both implementations—ControlXR and WasmXRGPU—performed these computations on the CPU. In ControlXR, JavaScript was responsible, while in WasmXRGPU, the computations were handled by C++.

These matrix operations were implemented using third-party libraries: **glmMatrix** for JavaScript and **glm** for C++. However, profiling revealed a bottleneck in WasmXRGPU related to computing model matrices on the CPU. This observation motivated the use of compute shaders in WebGPU to offload the matrix multiplications to the GPU, thereby improving performance and reducing CPU workload. As for ControlXR, the third-party library was used to perform the matrix multiplications.

Moreover, the inverse of each model matrix is required to accurately calculate lighting effects on surfaces—particularly the diffuse component. Unlike WebGL’s OpenGL Shading Language (GLSL), WebGPU’s WebGPU Shading Language (WGSL) does not provide a built-in inverse function for matrix inversion. Offloading this computation to the CPU for a large number of objects can lead to performance bottlenecks. Therefore, performing matrix inversion on the GPU is a more suitable and scalable approach.

4.4.4 Communication between JavaScript and C++

Chrome’s WebGPU implementation is called Dawn, an open-source project (*“Dawn, a WebGPU implementation”* 2025). It provides C/C++ header files that can be used to configure the graphics pipeline and issue GPU commands directly, making it suitable for use in the graphics backend of a WebXR application. With a

toolchain like Emscripten (*Emscripten* 2015), such C++ programs can be compiled into WebAssembly for web deployment.

However, the same flexibility is not available for the WebXR Device API. Its source code is not publicly available, and access is restricted to JavaScript interfaces. In their work, Liu et al. (2023) focus on improving the performance of WebXR applications using WebAssembly. Their experimental system is based on the Magnum Engine (*Magnum Engine* 2022), an open-source C++ engine. An investigation of Magnum’s source code reveals that they implemented a minimal C++ wrapper to interface with the WebXR Device API via JavaScript.

While this approach can be adopted in this study’s setup, it is important to recognize a key performance implication. Internally, the WebXR Device API ultimately delegates XR tasks to native backends written in languages like C++. Table 4.1 summarizes the types of runtime environments used by the WebXR Device API depending on the device. This indicates a runtime transition from JavaScript to native backends such as C++.

Using a wrapper, therefore, introduces an indirect execution path: $C++ \rightarrow \text{JavaScript} \rightarrow C++$

To minimize this back-and-forth transition overhead, this study proposes a more efficient alternative: initialize the WebXR session and call the API directly in JavaScript, then write the resulting data (such as view matrices or input states) into the WebAssembly memory heap. This allows the C++/WebAssembly side to directly access the data, reducing runtime switches and improving performance.

4.4.5 Compiler Optimizations

Compiler optimization plays a crucial role in improving the performance of applications, especially in high-performance computing domains like graphics rendering. In this study, the WebAssembly backend of the system is implemented in C++ and compiled using the Emscripten toolchain. The efficiency of the generated WebAssembly code heavily depends on how well the compiler optimizes the source code during the compilation process.

Emscripten supports several optimization levels, similar to traditional compilers like GCC and Clang. The optimization level is specified using flags such as -O0, -O1, -O2, -O3, or -Os. In this study, the -O3 optimization level is used to generate WebAssembly code for the C++-based system (WasmXRGPU). This ensures that the WebAssembly output is heavily optimized for execution speed, which is essential when performing computationally intensive tasks such as matrix

Device	OS	Runtime	Supported Session Modes
Oculus, SteamVR, Windows Mixed Reality, OpenXR compatible HMDs	Windows	OpenXR	immersive-vr
Cardboard, Daydream View, Lenovo Mirage Solo	Android	Google VR	immersive-vr
ARCore-compatible mobile devices	Android	ARCore	immersive-ar
Android XR devices	Android	OpenXR	immersive-vr, immersive-ar

Table 4.1: Chrome hardware support for various devices in WebXR Device API (*"WebXR Device API - Chrome Hardware Support"* 2025)

multiplication, memory management, and GPU pipeline configuration.

Emscripten also supports link-time optimization (LTO), which enables the compiler to perform global optimizations across multiple translation units during the linking phase. This is particularly effective in eliminating unused code and reducing memory usage in the final WebAssembly binary.

4.5 3D Model Import

3D model rendering is a fundamental component of any graphics application. Although a 3D model is essentially a collection of vertices, indices, and texture coordinates, various file formats exist to store and organize this data—such as FBX, glTF, and OBJ. Among these, glTF stands out as an open-source format that uses JavaScript Object Notation (JSON) and supports a scene graph structure, making it well-suited for representing parent-child relationships between primitives. Additionally, Nam et al. (2019) highlight that glTF offers superior performance on mobile web browsers.

Parsing a glTF file, however, involves additional complexity and falls outside the scope of this study. Therefore, third-party libraries are utilized to handle parsing: `loaders.gl` (*"loaders.gl - A collection of loaders modules for Geospatial and 3D visualization use cases"* 2025) is used for the JavaScript-based ControlXR, and `fastgltf`

(*"fastglTF - Documentation"* 2025) is used in the C++-based WasmXRGPU.

Once the glTF files are parsed, the subsequent extraction and storage of relevant data follow an identical process in both implementations.

4.6 Rendering Frameworks

As discussed in Section 4.3, two separate frameworks were developed to evaluate the WebGPU-WebGL interoperability method. Similarly, the rendering backends for the two major implementations—ControlXR and WasmXRGPU—are designed as independent rendering frameworks. This design approach ensures consistency and minimizes discrepancies in implementation details, enabling a fair comparison.

To enhance performance while leveraging the unique capabilities of WebGPU and WebAssembly, both frameworks incorporate the performance considerations described in Section 4.4. The ControlXR backend is implemented in TypeScript, benefiting from type safety and reduced runtime errors. In contrast, WasmXRGPU's backend is developed in C++, compiled to WebAssembly for browser execution.

Despite the difference in languages, the two frameworks follow similar architecture and design principles. The following are key classes commonly implemented in both frameworks:

- **Renderer** – Initializes and configures the rendering pipeline for the chosen graphics API (WebGL or WebGPU).
- **SceneObject** – Represents a single 3D object in the scene, including transformation data such as position, rotation, and scale.
- **Mesh** – Stores geometry data, including vertices, indices, and texture coordinates.
- **Texture** – Represents a texture resource that can be applied to one or more meshes.
- **Model** – Handles model importing and constructs scene objects based on the imported data.
- **Camera** – Represents the scene's view camera, responsible for computing the view and projection matrices.

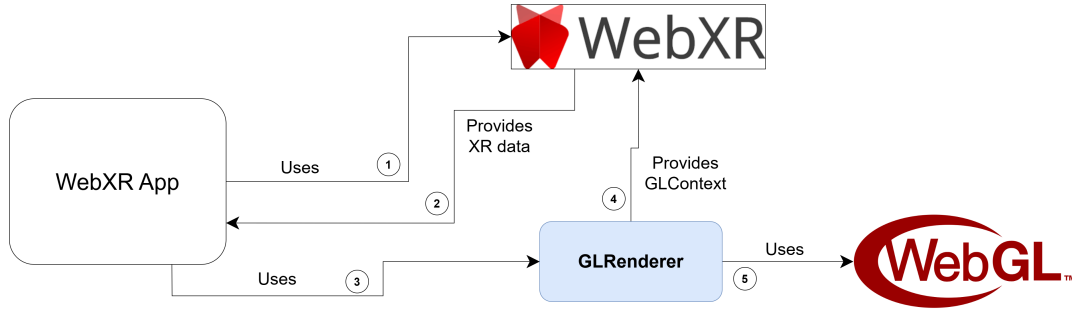


Figure 4.4: ControlXR Implementation

4.7 Practical Implementation and Integration

Based on the concepts discussed thus far, appropriate implementations were developed for both groups—ControlXR and WasmXRGPU. This section outlines the specific implementation details for each group and demonstrates how the performance-oriented techniques were practically integrated.

4.7.1 ControlXR

This implementation serves as the control group of the study. It represents traditional WebXR applications that rely solely on JavaScript and WebGL. While numerous 3D libraries and frameworks currently exist for building WebXR applications, using them in this context would introduce potential bias, as they may include internal optimizations (e.g., frustum culling, LOD management) that are not present in the experimental group. Therefore, to ensure a fair comparison, only essential third-party libraries were used—specifically those that solve specific problems without introducing unrelated optimizations.

The overall implementation architecture is illustrated in Figure 4.4. The interactions can be described as follows.

1. The main file of the application initiates the process by calling essential functions such as `navigator.xr.requestSession()` to create a WebXR session using the WebXR Device API.
2. Once the session is created, the application utilizes it to access the relevant XR data provided by the WebXR Device API.
3. This XR data is then passed to the **GLRenderer**, which is the core rendering framework responsible for rendering the frames.

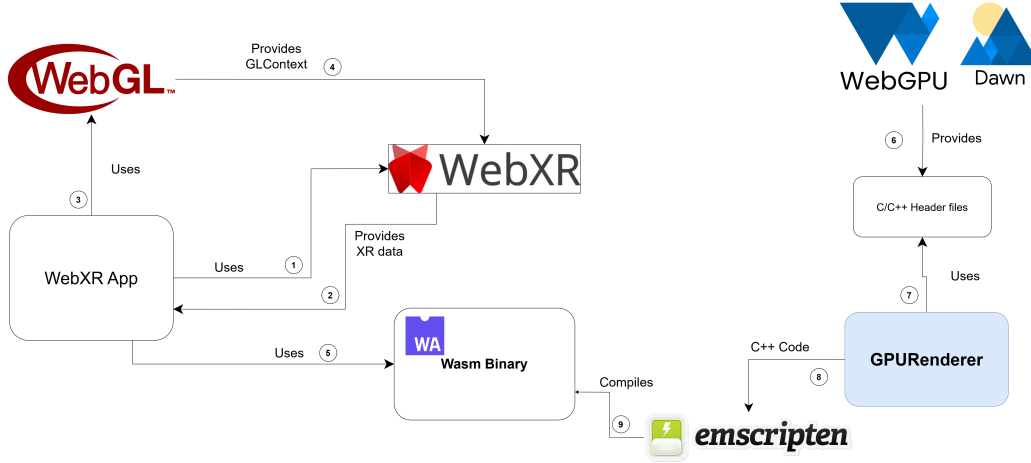


Figure 4.5: WasmXRGPU Implementation

4. As previously mentioned, WebXR relies on WebGL to display XR content. Therefore, the necessary WebGL rendering context (GLContext) is created and managed by the **GLRenderer**.
5. The **GLRenderer** encapsulates the low-level WebGL operations, handling buffer configurations, pipeline setup, and issuing rendering commands to the GPU.

The deployed version of ControlXR is publicly accessible *here*.

4.7.2 WasmXRGPU

This implementation represents the experimental group of the study, showcasing the performance advantages of using WebGPU and WebAssembly. In contrast to traditional WebXR applications written entirely in JavaScript, this implementation is primarily written in C++ and compiled to WebAssembly using Emscripten. It leverages WebGPU as the graphics API. As with ControlXR, certain third-party libraries are used to handle non-core tasks such as model parsing, to maintain consistency across both implementations and avoid deviation from the research scope. Notably, the rendering backend is built as a custom C++ framework that handles GPU resource management, rendering pipelines, and frame submission logic.

The overall architecture of WasmXRGPU is illustrated in Figure 4.5, and the interactions can be summarized as follows:

1. The main file of the application initiates the WebXR workflow by invoking `navigator.xr.requestSession()` to create a WebXR session using the WebXR Device API.

2. Once the session is established, the application uses it to access the relevant XR data provided by the WebXR Device API.
3. To comply with WebXR’s requirement for a WebGL context, a WebGL rendering context is still used—both to initiate the session and to blit the WebGPU-rendered frame into the WebGL framebuffer for display.
4. The created WebGL context is provided to WebXR Device API to create the rendering layer.
5. When the XR data (e.g., view and projection matrices) is retrieved, it is written to the WebAssembly memory heap, allowing it to be accessed directly from the C++ side of the application.
6. The WebGPU rendering backend is implemented in C++ using Dawn—the native C/C++ implementation of WebGPU, which provides the necessary headers and low-level control.
7. The **GPURenderer** class is the core rendering framework for WasmXRGPU. It is responsible for managing pipeline configuration, buffer setup, and issuing draw commands.
8. The **GPURenderer** source code is compiled using Emscripten, which translates the C++ code into WebAssembly binaries and the accompanying JavaScript glue code required for browser execution.
9. The resulting `.wasm` binary is then used within the WebXR application to render the scene through WebGPU, while still maintaining communication with the WebXR runtime via JavaScript.

The deployed version of WasmXRGPU is publicly accessible ***here***.

Chapter 5

Results and Analysis

This chapter presents the outcomes derived from the implementation of the two experimental groups—ControlXR and WasmXRGPU—and analyzes their performance based on the evaluation plan proposed in Section 3.3. Two major experiments are conducted as part of this study. The first focuses on evaluating the WebGPU-WebGL interoperability method, while the second compares the overall performance of the two experimental groups. The results from both experiments are analyzed to assess the practical effectiveness of the interoperability technique and the performance benefits of using WebGPU and WebAssembly in real-time XR applications.

The following metrics are measured in the both experiments except for the CPP Time as CPP time can only be measured in the implementation that uses C++.

- **Frame Time (ms):** The average time taken to render a single frame, which is inversely related to FPS. This metric provides a comprehensive measure of overall application performance, encompassing both CPU and GPU contributions.
- **JavaScript Time (ms):** The total time spent executing JavaScript code during the rendering process. This metric highlights CPU-side performance, including scene updates, API calls, and resource management, which can impact the efficiency of graphics rendering.
- **CPP Time (ms):** The total time spent executing C++ code during the rendering process. This metric highlights CPU-side performance, including scene updates, API calls, and resource management, which can impact the efficiency of graphics rendering.
- **Framerate Per Second (FPS):** The number of frames rendered per second,

serving as a direct indicator of rendering smoothness and real-time performance. A higher FPS generally signifies a more responsive and visually fluid experience.

- WebGL Time (ms): The GPU-side rendering time measured using WebGL’s *EXT_disjoint_timer_query* extension. This metric isolates the GPU workload in WebGL, helping to analyze rendering efficiency and potential bottlenecks.
- WebGPU Time (ms): The GPU-side rendering time measured using WebGPU’s *timestamp-query* extension. This metric provides insights into GPU processing efficiency within WebGPU, facilitating comparisons with WebGL and evaluating WebGPU’s potential advantages.

5.1 WebGPU-WebGL Interoperability Experiment

Due to a limitation in WebXR Device API that doesn’t support WebGPU, a method was introduced in Section 4.3. As that section suggests, one needs to evaluate that method to check for potential overheads. To conduct the experiment, following approach is taken.

Since this interoperability does not depend on any other factors such as WebXR tasks, the experiment is setup to render some scenes with different complexities. For this, only JavaScript APIs of WebGL and WebGPU are used. Furthermore, two frameworks are built on top of WebGL and WebGPU to replicate the same tasks and functionalities that are offered by these APIs. For example, WebGPU provides flexibility in defining how the scene is multisampled. But, WebGL automatically does this by providing the canvas with the relevant options. Hence, to reduce the discrepancy between the two APIs, such features are turned off.

The experiment is conducted on a laptop with the following specification.

- MSI GF63 Thin 10SC
- Intel Core i5-10300H 2.50GHz
- 16GB DDR4 Memory
- Integrated Graphics : Intel UHD Graphics (VRAM : 128MB)
- Dedicated Graphics : NVIDIA Geforce GTX 1650 with Max-Q Design (VRAM: 3937MB)

The browser used for the experiment:

- Chrome : Version 131.0.6778.205 (Official Build) (64-bit)

However, despite having Dedicated Graphics, the browser was forced to use integrated graphics to simulate lower-end devices. Furthermore, the use of dedicated graphics requires scenes with very high complexity to affect the performance.

In the experiment, the following two scenarios are taken into account.

5.1.1 Using a Single High Poly 3D Model

Five 3D scenes were designed to simulate various rendering complexities. These scenes were composed of multiple instances of a 3D model. The scenes were organized as presented in Table 5.1.

Scenario	Polygons	Draw Calls
1	0	0
2	6,405,421	446
3	12,810,842	892
4	19,216,263	1338
5	25,621,684	1784

Table 5.1: Polygon and draw call count for each scenario when using a single high poly 3D model

Despite utilizing multiple instances of the same 3D model, instanced drawing was deliberately not adopted in these experiments. Instead, each instance was treated as a separate model within the scenario. This approach ensures that the performance impact of handling individual draw calls for each instance is evaluated, providing insights into the overhead associated with increased draw calls rather than leveraging the optimization capabilities of instanced rendering.

Results

We conducted experiments considering each rendering scenario, as outlined above. The results were analyzed to compare the performance of the WebGL-only, WebGPU-only, and WebGPU-WebGL interoperability methods.

The experimental results of the scenarios were analyzed by averaging the values of key performance metrics across all cases. To calculate the average, 1000 frames were recorded. These averages were plotted in bar graphs for clear comparative visualization. The performance of the WebGPU-WebGL interoperability approach is represented by the green bar in the graphs. The key observations on these results are stated below for each experiment. The detailed discussion is available in Section 6.1 of Chapter 6.

1) Average Frame Time

For the WebGPU-WebGL interop approach, frame time does not reflect the sum of WebGL and WebGPU times, contrary to expectations if the two APIs operated independently for a single frame. The results can be seen in Figure 5.1.

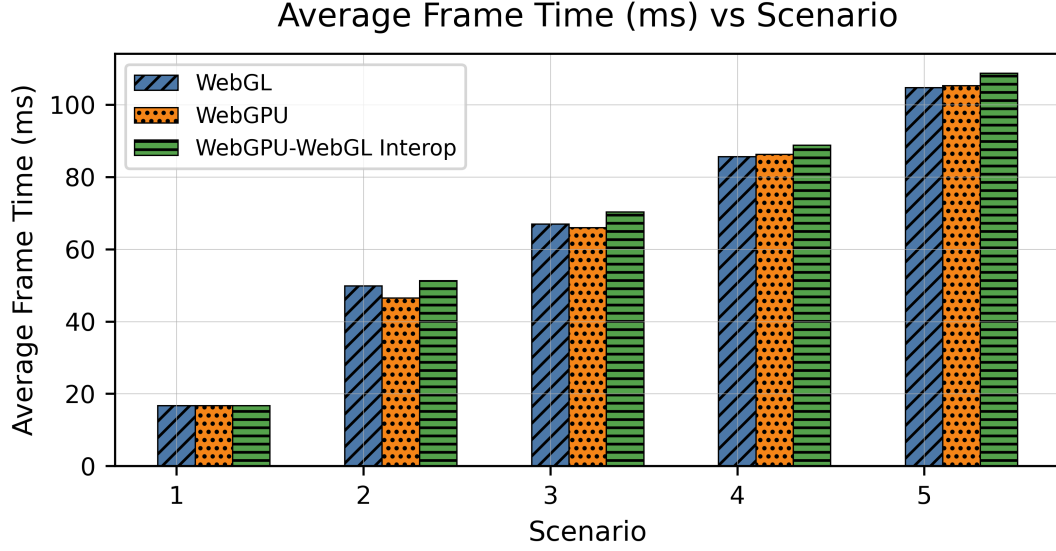


Figure 5.1: Comparison of Average Frame Time (ms) between defined scenarios in Table 5.1. Lower is better.

2) Average Framerate Per Second (FPS)

Figure 5.2 illustrates the average FPS across all scenarios. As anticipated, the FPS decreases with increasing scene complexity. Notably, all methods demonstrated comparable performance, suggesting that the observed variations in FPS are primarily attributable to the overall increase in computational load rather than differences between the rendering approaches.

3) Average WebGL Time and Average WebGPU Time

In the WebGL-only application, WebGPU time cannot be measured, resulting in a flat, zero-value line in the corresponding graph depicted in Figure 5.3. Similarly, in the WebGPU-only application, WebGL time is absent as shown in Figure 5.4.

4) Average JavaScript Time

Applications leveraging WebGPU (both WebGPU-only and WebGPU-WebGL interop) exhibit higher JavaScript execution time compared to the WebGL-only application. This is apparent in the graph visualized in Figure 5.5. This discrepancy likely arises from increased configuration overhead on the CPU-side in WebGPU, where more detailed and explicit control is required compared to the relatively streamlined WebGL pipeline.

One interesting observation is that the WebGPU-WebGL interoperability method

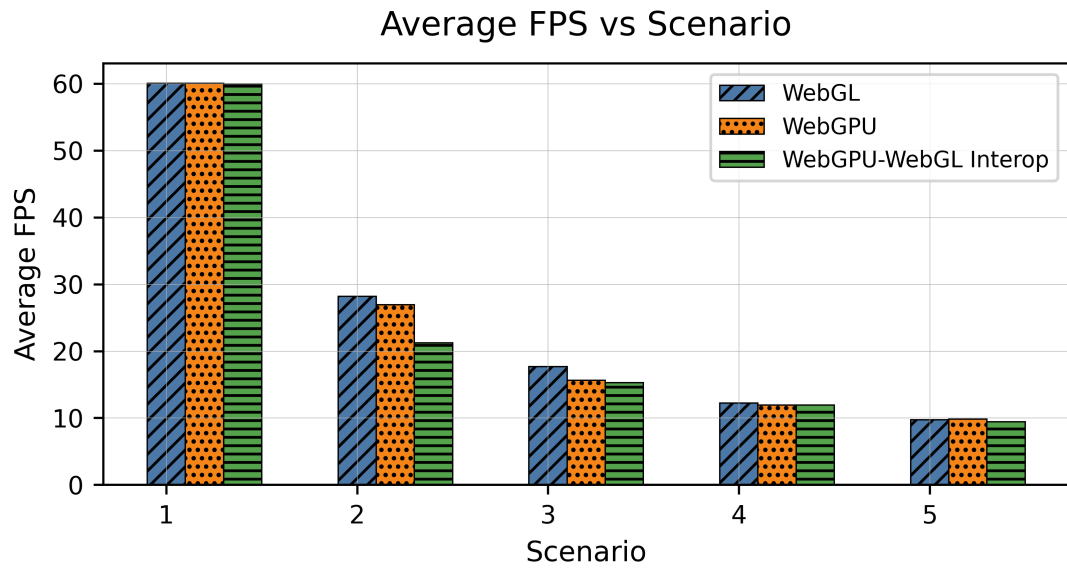


Figure 5.2: Comparison of Average FPS between defined scenarios in Table 5.1. Higher is better.

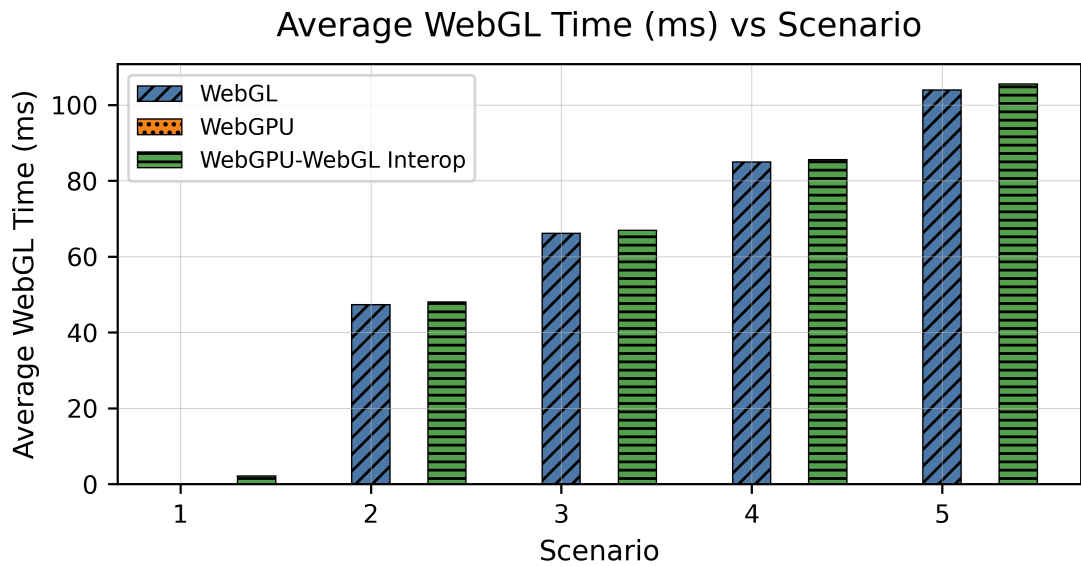


Figure 5.3: Comparison of Average WebGL Time (ms) between defined scenarios in Table 5.1. Lower is better.

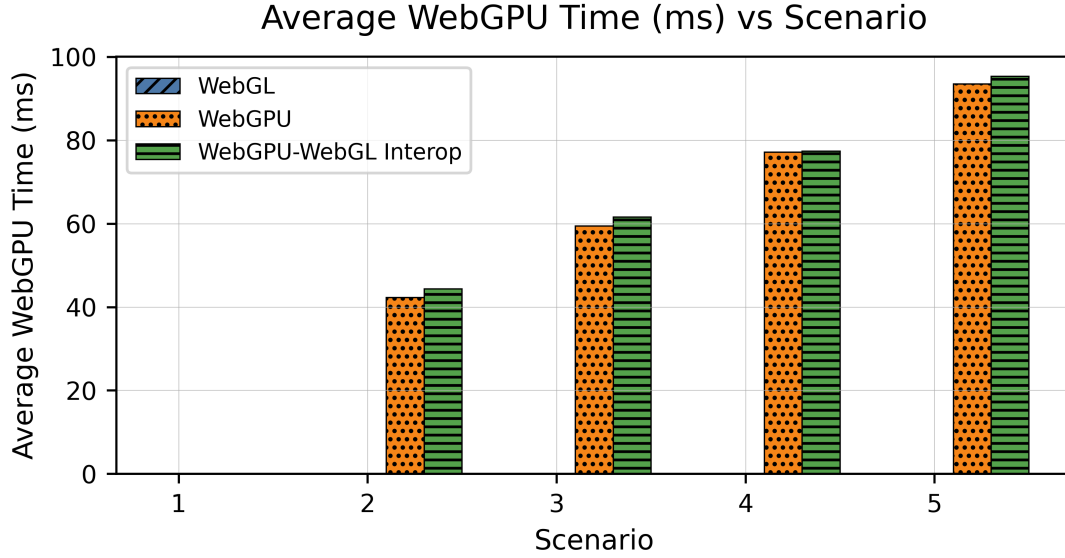


Figure 5.4: Comparison of Average WebGPU Time (ms) between defined scenarios in Table 5.1. Lower is better.

exhibits a gradual reduction in Average JavaScript Time as the scene complexity increases. Surprisingly, despite the interoperability method requiring additional JavaScript code to bridge the two APIs, it still outperforms the WebGPU-only method in terms of timing performance.

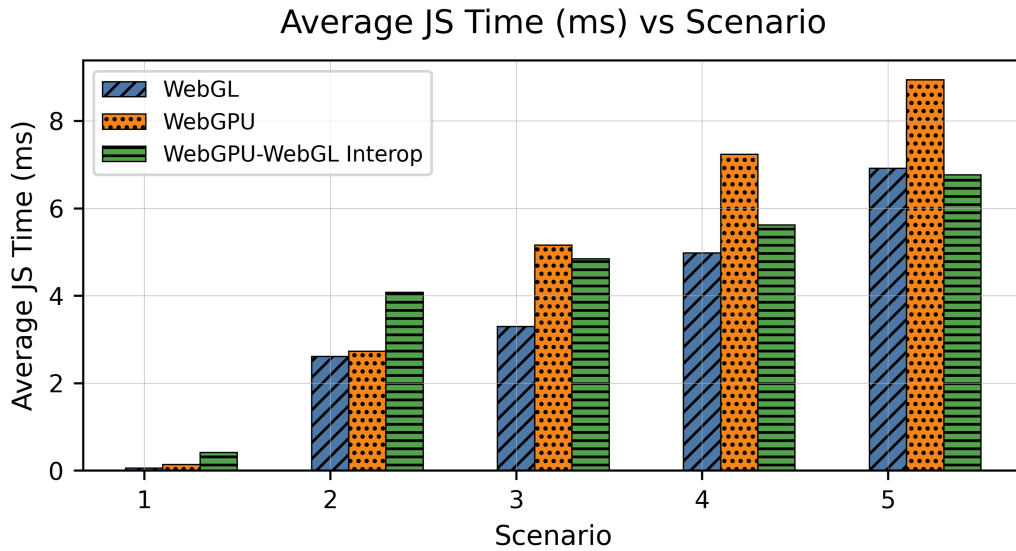


Figure 5.5: Comparison of Average JavaScript Time (ms) between defined scenarios in Table 5.1. Lower is better.

5.1.2 Using Two 3D Models

Furthermore, we also created 3D scenarios using two 3D models with fewer polygons than the model specified in Table 5.1. In this experiment, we considered six

different scenarios, gradually increasing the complexity of the scenes by scaling the number of instances of these models. The details of the scenarios, including polygon and draw call counts, are provided in Table 5.2.

Scenario	Polygons	Draw Calls
1	0	0
2	4,217,325	1150
3	8,434,650	2300
4	12,651,975	3450
5	16,869,300	4600
6	21,086,625	5750

Table 5.2: Polygon and draw call count for each scenario when using multiple instances of two 3D models

The purpose of this experiment is to investigate the impact of draw call volume on the overall performance of graphics APIs, even when the scene complexity remains similar.

Results

Notably, it is clear that WebGL outperforms WebGPU significantly as the scene complexity increases. However, when comparing Scenario 6 in Table 5.2 (denoted as A) with Scenario 5 in Table 5.1 (denoted as B), we observe that the total polygon count in Scenario A (21,086,625) is lower than that in Scenario B (25,621,684). Despite this, the total number of draw calls in Scenario A (5750) is considerably higher than in Scenario B (1784). The results of the experiment are depicted in Figures 5.6, 5.7, 5.8, 5.9, and 5.10 respectively. The detailed discussion is available in Section 6.1 of Chapter 6.

5.2 ControlXR and WasmXRGPU Experiment - Quantitative Analysis

The results of the WebGPU–WebGL interoperability experiment indicate that there is no significant overhead introduced when utilizing both graphics APIs in conjunction. This confirms the viability of the proposed interoperability method and allows us to proceed to the core experiment of the study: evaluating the runtime performance of the two experimental groups.

As outlined in the evaluation plan, this experiment compares the performance of **ControlXR** and **WasmXRGPU** using the metrics introduced earlier. Given the considerations discussed in the previous chapters, and based on the outcomes

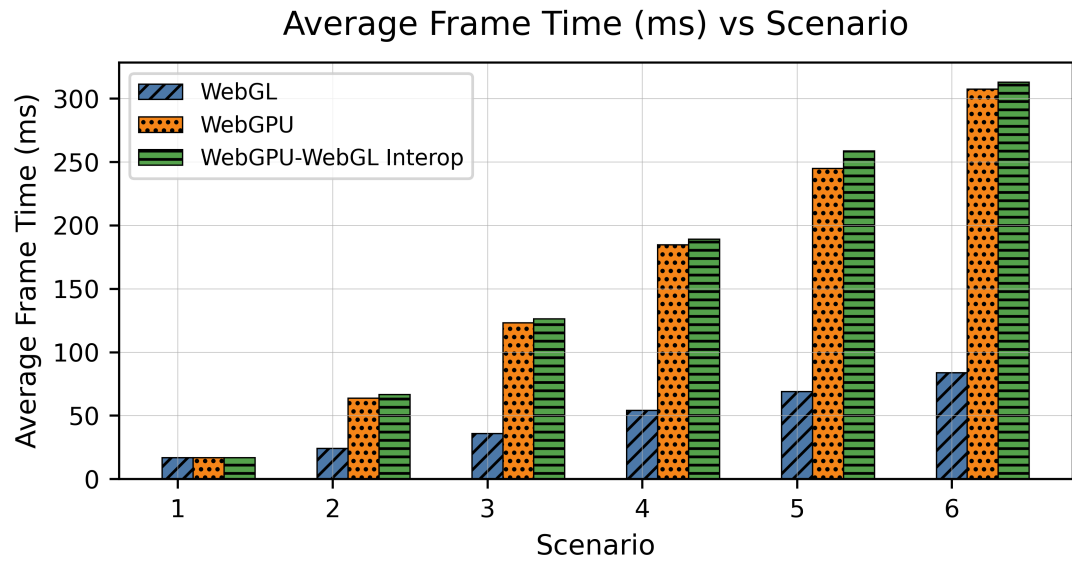


Figure 5.6: Comparison of Average Frame Time (ms) between defined scenarios in Table 5.2. Lower is better.

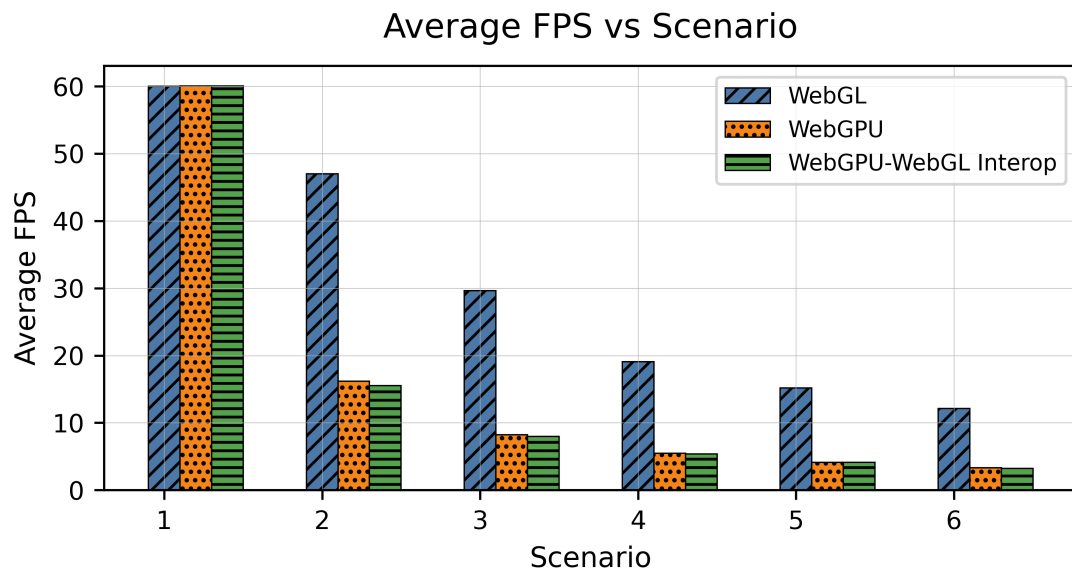


Figure 5.7: Comparison of Average FPS between defined scenarios in Table 5.2. Higher is better.

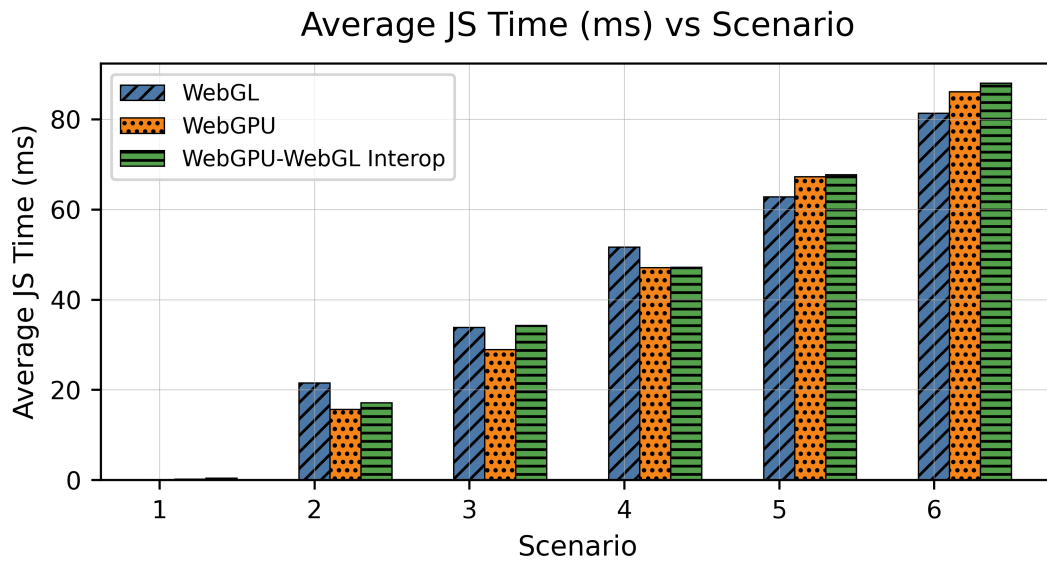


Figure 5.8: Comparison of Average JavaScript Time (ms) between defined scenarios in Table 5.2. Lower is better.

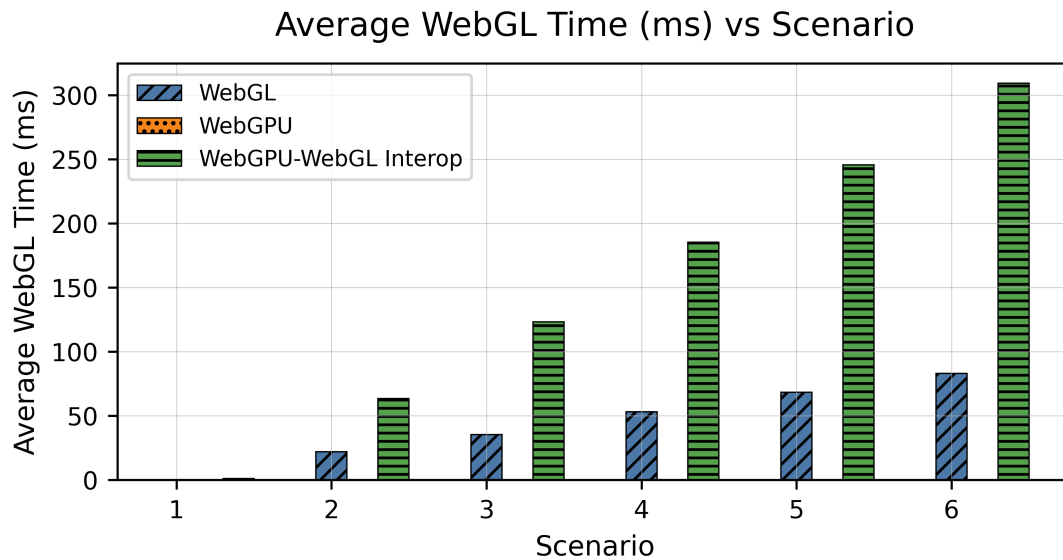


Figure 5.9: Comparison of Average WebGL Time (ms) between defined scenarios in Table 5.2. Lower is better.

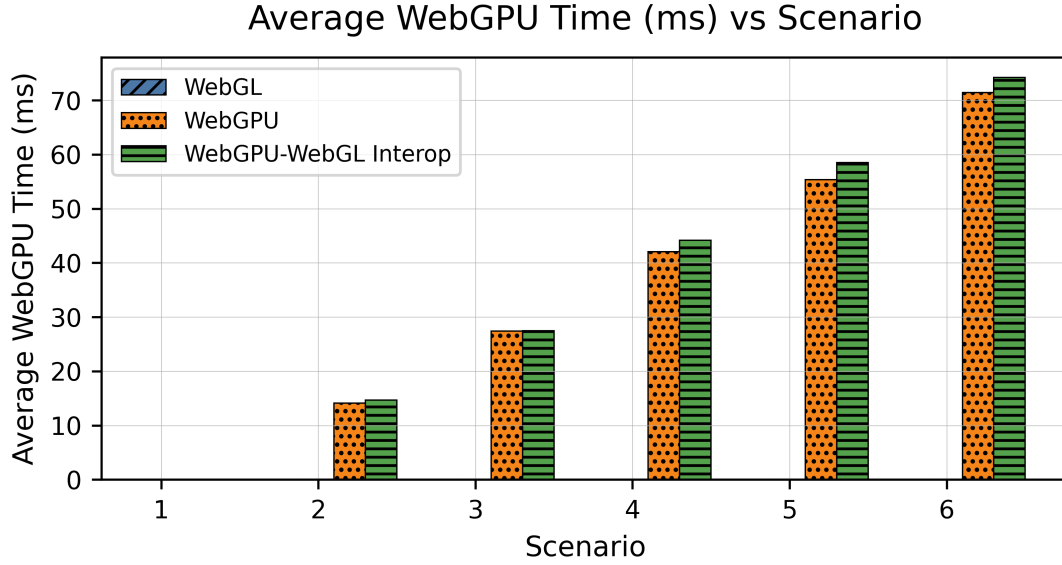


Figure 5.10: Comparison of Average WebGPU Time (ms) between defined scenarios in Table 5.2. Lower is better.

of the interoperability experiment, a single 3D model is used to construct the rendering scenarios. This model consists of 5,252 triangles and 15,756 vertices and is illustrated in Figure 5.11. It should be noted that due to implementation differences in the fragment shaders, particularly regarding the color handling, the final rendered appearance of the model differs slightly from the figure itself.

Using this model, a total of six rendering scenarios were created to evaluate performance under varying scene complexities and configurations. These scenarios aim to simulate different rendering loads and stress levels, enabling a comprehensive comparison between the two implementations. The details of each scenario are summarized in Table 5.3.

Scenario	Polygons
1	131,300
2	262,600
3	525,200
4	1,050,400
5	2,100,800
6	4,201,600

Table 5.3: Polygon count for each scenario

Moreover, to ensure the experiment accounts for a range of hardware capabilities, the devices listed in Table 5.4 have been selected, encompassing varying specifications and performance levels.

In both implementations, once the user initiates the XR session, the previously

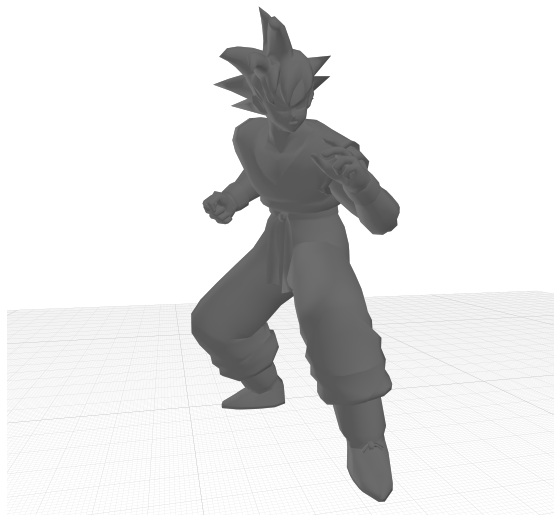


Figure 5.11: Used 3D model

Device	Browser and Version
Google Pixel 4A	Google Chrome: 133.0.6943.121
Google Pixel 6A	Google Chrome: 134.0.6998.39
Samsung Galaxy A15	Google Chrome: 134.0.6998.39
Samsung Galaxy A52	Google Chrome: 133.0.6943.137
Samsung Galaxy S23 Ultra	Google Chrome: 133.0.6943.138
Meta Quest 2	Browser: 37.2.0.6.62

Table 5.4: Selected devices and the web browser details

mentioned metrics are recorded over a span of 500 frames. These recorded values are then appropriately plotted to facilitate performance evaluation. From this point forward, it is important to note that CPU time refers to JavaScript execution time in the ControlXR implementation and C++ execution time in the WasmXRGPU implementation, whereas GPU time refers to WebGL time in the ControlXR implementation and WebGPU time in the WasmXRGPU implementation.

5.2.1 Virtual Reality (VR)

The VR sessions for each scenario listed in Table 5.3 are rendered independently in both experimental groups.

Average Performance

The recorded results are averaged over the 500-frame capture window and plotted to visualize the average performance. The captured results are visualized and compared between the two implementations for each device.

1) Average CPU Time

Figure 5.12 illustrates the average CPU time across all tested devices for each scenario. As the scene complexity increases, a clear performance trend emerges: CPU time in WasmXRGPU implementation consistently improves relative to ControlXR. The average CPU time values for each device are visualized in a heatmap form which is depicted in Figure 5.16.

2) Average GPU Time

Figure 5.13 illustrates the average GPU time across all tested devices for each scenario. As scene complexity increases, a clear performance trend emerges: contrary to the improvements observed in CPU time, the GPU time in the WasmXRGPU implementation increases significantly compared to ControlXR. The average GPU time values for each device are visualized in a heatmap form which is depicted in Figure 5.17.

3) Average Framerate Per Second (FPS)

Figure 5.14 illustrates the average FPS across all tested devices for each scenario. As scene complexity increases, both implementations exhibit a gradual decline in FPS, which aligns with expected performance behavior. However, ControlXR displays certain anomalies on specific devices, where the FPS unexpectedly increases despite a rise in scene complexity. The average FPS values for each device are

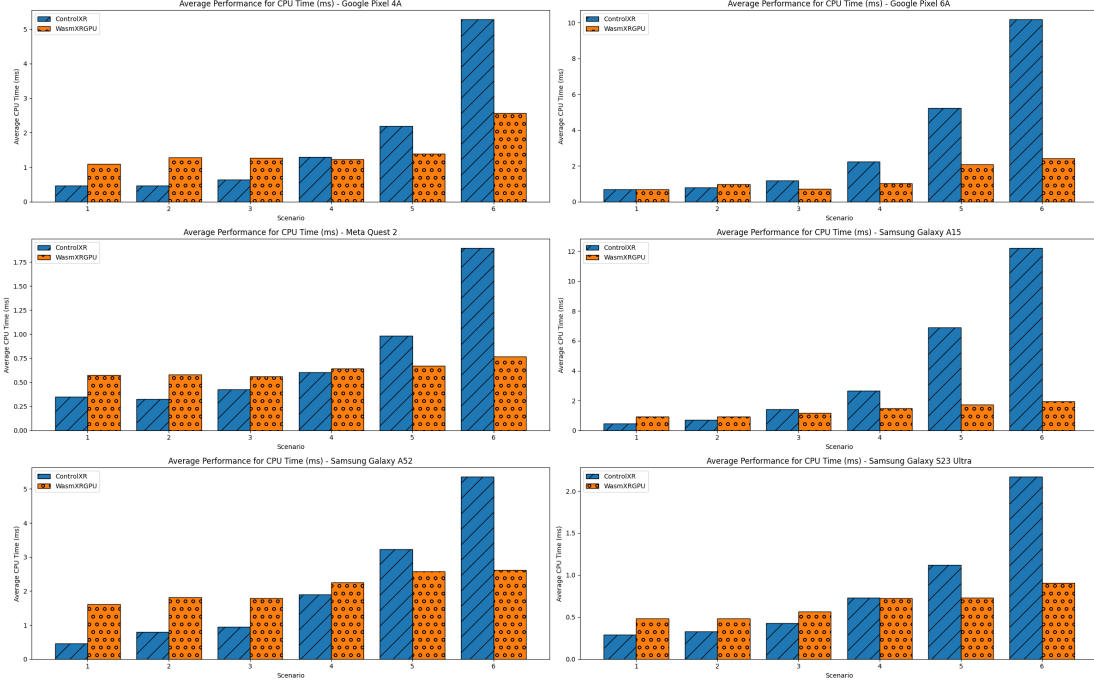


Figure 5.12: Comparison of Average CPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

visualized in a heatmap form which is depicted in Figure 5.18.

4) Average Frame Time

Figure 5.15 illustrates the average frame time across all tested devices for each scenario. As scene complexity increases, both implementations exhibit a gradual rise in frame time, which is consistent with expected performance behavior. Overall, ControlXR demonstrates better performance than WasmXRGPU across all scenarios and devices, with the exception of the sixth scenario on the Google Pixel 6A, where WasmXRGPU slightly outperforms ControlXR. The average frame time values for each device are visualized in a heatmap form which is depicted in Figure 5.19.

Per-Frame Performance

While average values provide a general understanding of the overall performance, anomalies were observed in some of the metrics. To identify the cause of these anomalies and gain a deeper understanding of the performance distribution, it is crucial to examine the full range of recorded values. In this section, scatter plots and violin plots are used to visualize the distribution of each performance metric, allowing for the detection of trends, outliers, and inconsistencies.

The following plots present per-frame data for key performance metrics (e.g., CPU

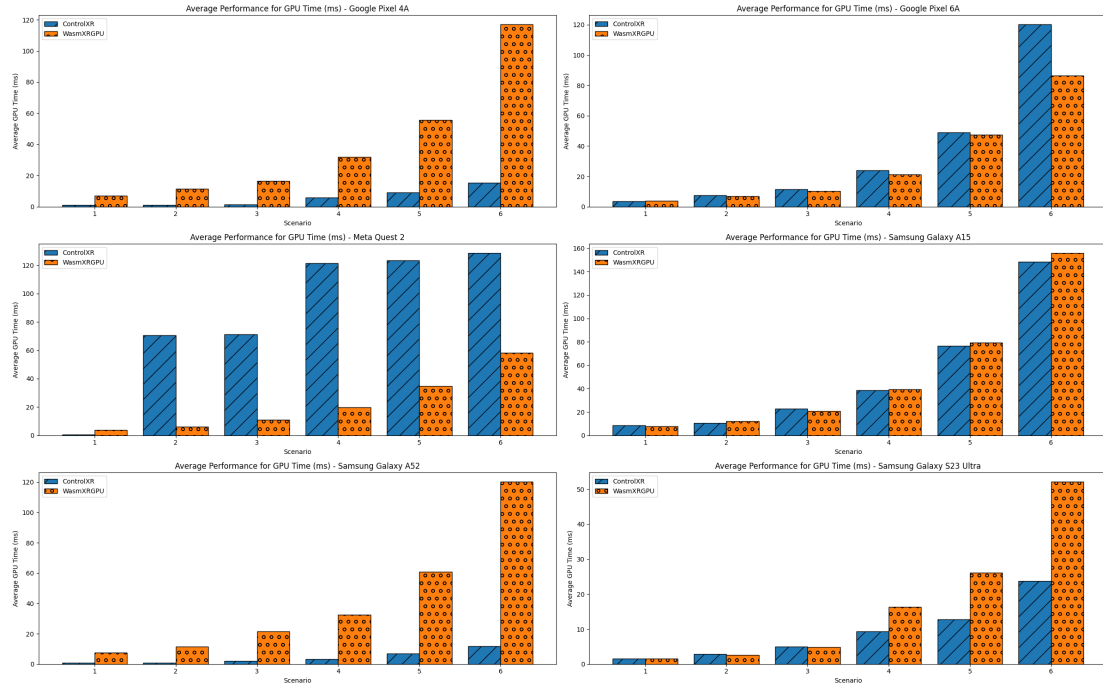


Figure 5.13: Comparison of Average GPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

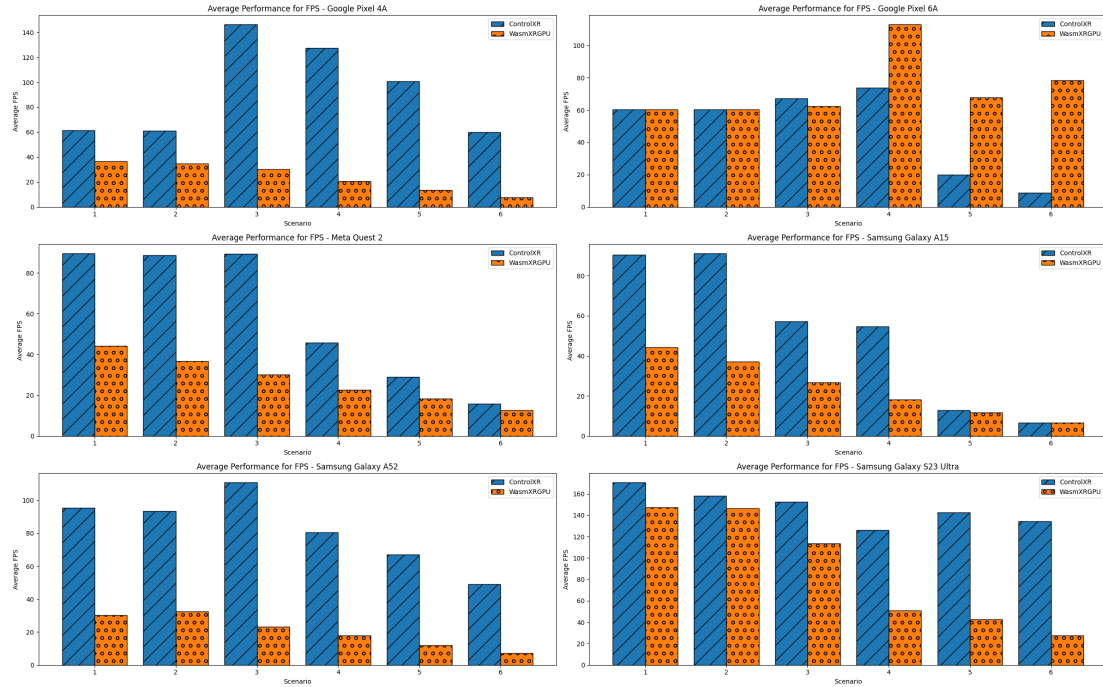


Figure 5.14: Comparison of Average FPS between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.

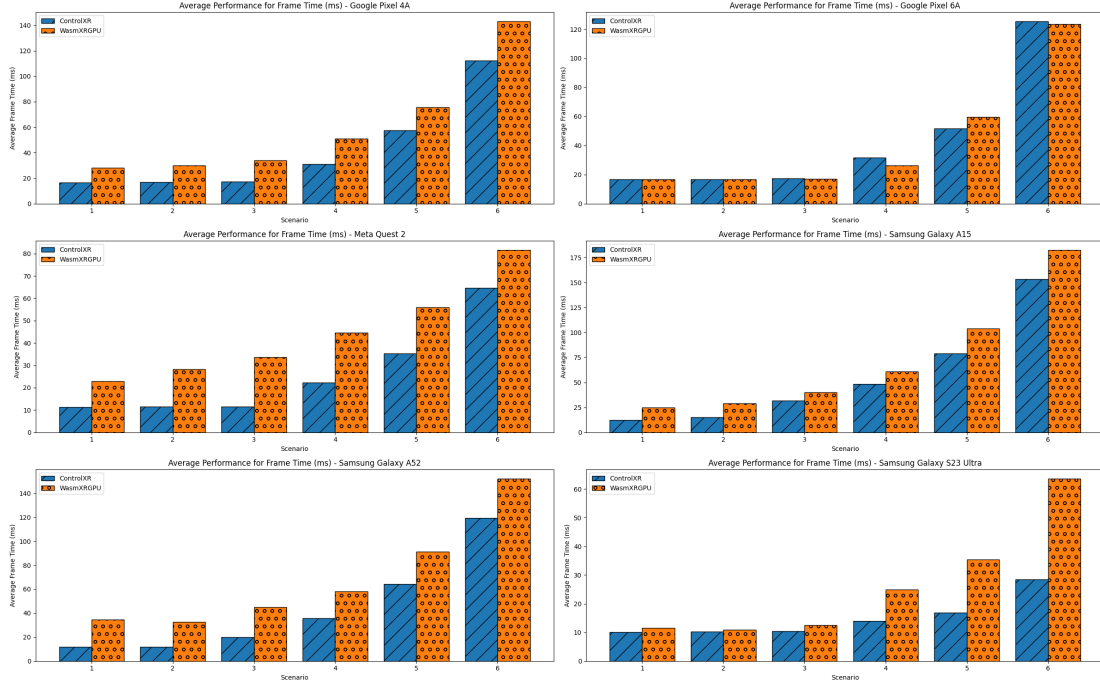


Figure 5.15: Comparison of Average Frame Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

time, GPU time, frame time, FPS) across different scenarios and devices. This detailed analysis helps to highlight the variability in performance and provides insight into the causes behind the observed anomalies. For brevity, this section will focus on presenting the FPS and Frame Time metrics, as they are directly related and offer a clear view of overall performance. Visualizations for all other metrics can be found in the Appendix A.1 and A.2.

1) Scatter Plot

This section will visualize all the recorded data using scatter plots. The first 10 records are omitted from the visualization, as they may skew the data representation. These initial 10 records correspond to the initialization phase of the experiment.

1.1) Scenario 1

Figure 5.20 and Figure 5.21 visualize the distributions of the FPS and Frame Time (ms) metric values, respectively, for Scenario 1.

1.2) Scenario 2

Figure 5.22 and Figure 5.23 visualize the distributions of the FPS and Frame Time (ms) metric values, respectively, for Scenario 2.

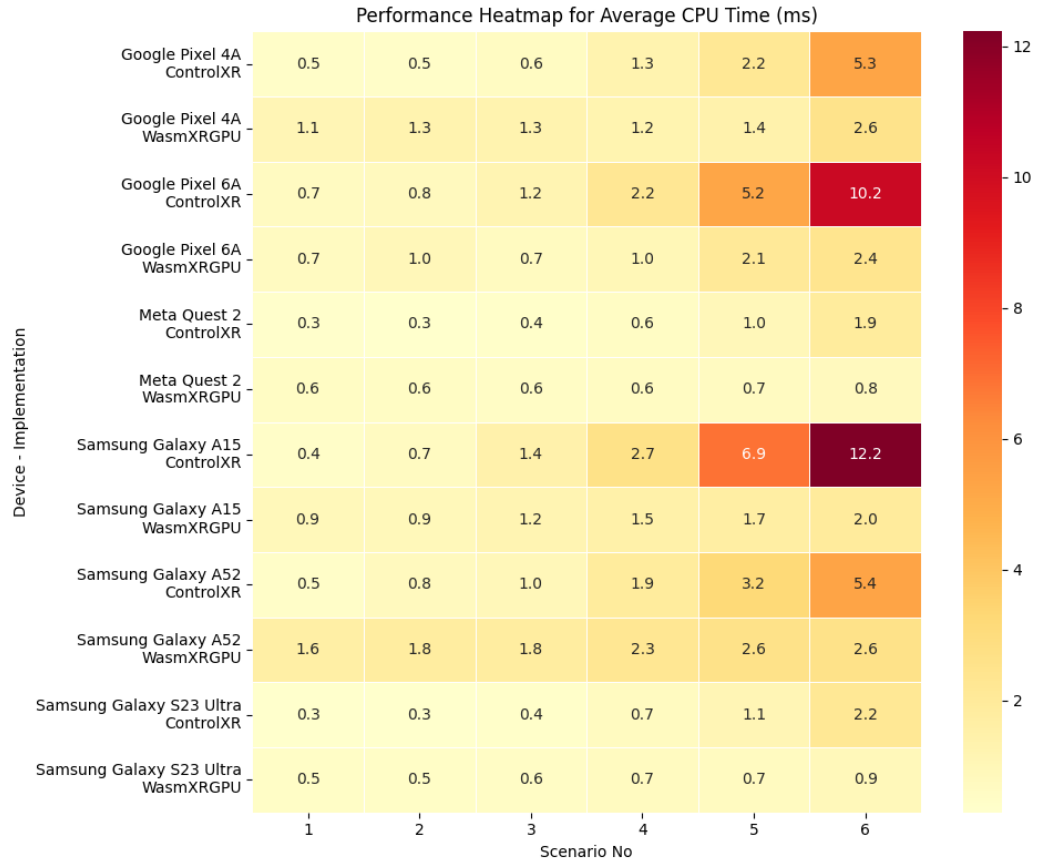


Figure 5.16: Heatmap Visualization of Average CPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower the color, better.



Figure 5.17: Heatmap Visualization of Average GPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower the color, is better.



Figure 5.18: Heatmap Visualization of Average FPS between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher the color, is better.

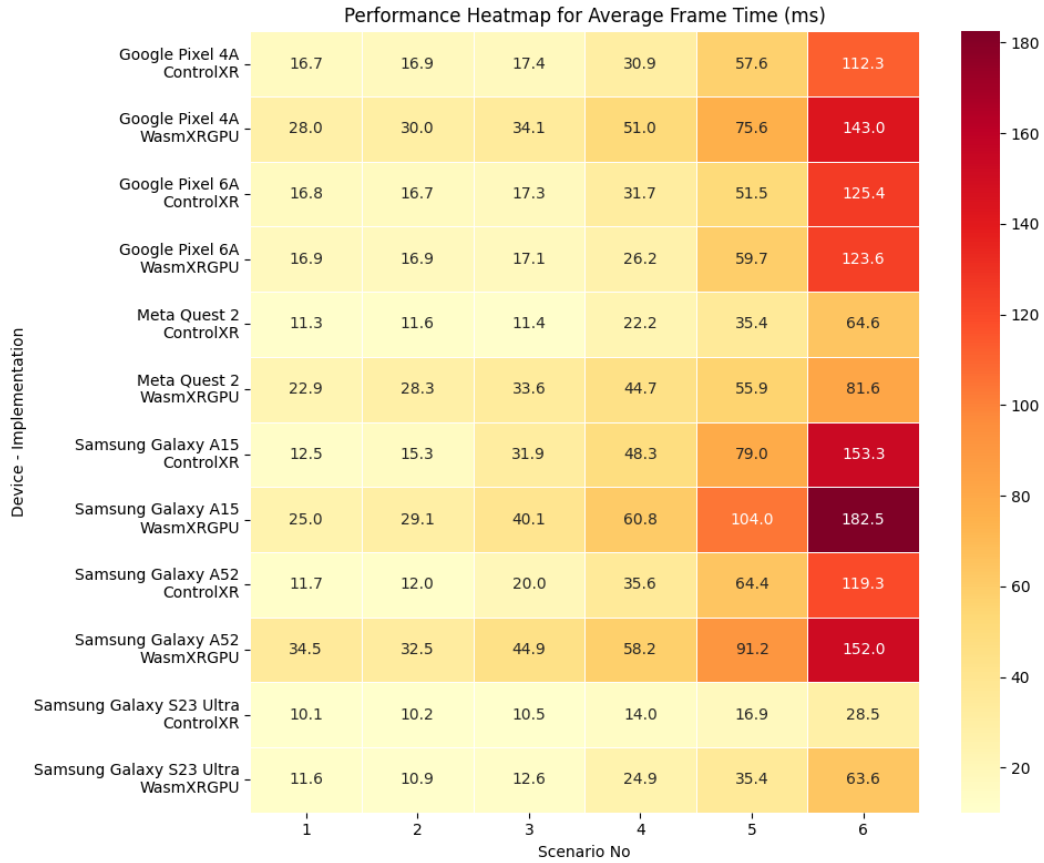


Figure 5.19: Heatmap Visualization of Average Frame Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower the color, better.

1.3) Scenario 3

Figure 5.24 and Figure 5.25 visualize the distributions of the FPS and Frame Time (ms) metric values, respectively, for Scenario 3. While the Samsung Galaxy S23 Ultra previously showed a high spread in data across both implementations, in Scenario 3, the metric values for ControlXR are concentrated between two distinct ranges: a higher end and a lower end.

1.4) Scenario 4

Figure 5.26 and Figure 5.27 visualize the distributions of the FPS and Frame Time (ms) metric values, respectively, for Scenario 4. Similar to Scenario 3, ControlXR exhibits values that are concentrated between two distinct ranges, indicating variability in frame performance. In contrast, WasmXRGPU maintains a more consistent distribution across most devices.

1.5) Scenario 5

Figure 5.28 and Figure 5.29 illustrate the distributions of FPS and Frame Time (ms) metric values for Scenario 5. The pattern of values clustering between two distinct ranges becomes more pronounced in ControlXR, indicating persistent variability in frame performance. Interestingly, WasmXRGPU begins to exhibit a similar behavior on certain devices.

1.6) Scenario 6

Figure 5.30 and Figure 5.31 illustrate the distributions of FPS and Frame Time (ms) metric values for Scenario 6, which features the highest scene complexity. The clustering of values between two distinct ranges is most prominent in ControlXR, suggesting heightened variability under heavier rendering loads. In contrast, WasmXRGPU displays similar dual-range behavior only on the Google Pixel 6A, and to a much lesser extent on the Samsung Galaxy A15 and Samsung Galaxy S23 Ultra.

2) Violin Plot

Violin plots are a method of visualizing the distribution of numerical data. They combine features of a box plot and a kernel density plot, providing both summary statistics (like median and interquartile range) and the probability density of the data at different values. This allows for a clearer understanding of the shape and spread of the data, making them especially useful for identifying multimodal distributions, outliers, and variations in performance across scenarios or devices.

In this section, all six scenarios are visualized in a single plot for each device, providing a comprehensive comparison of performance trends across different scene

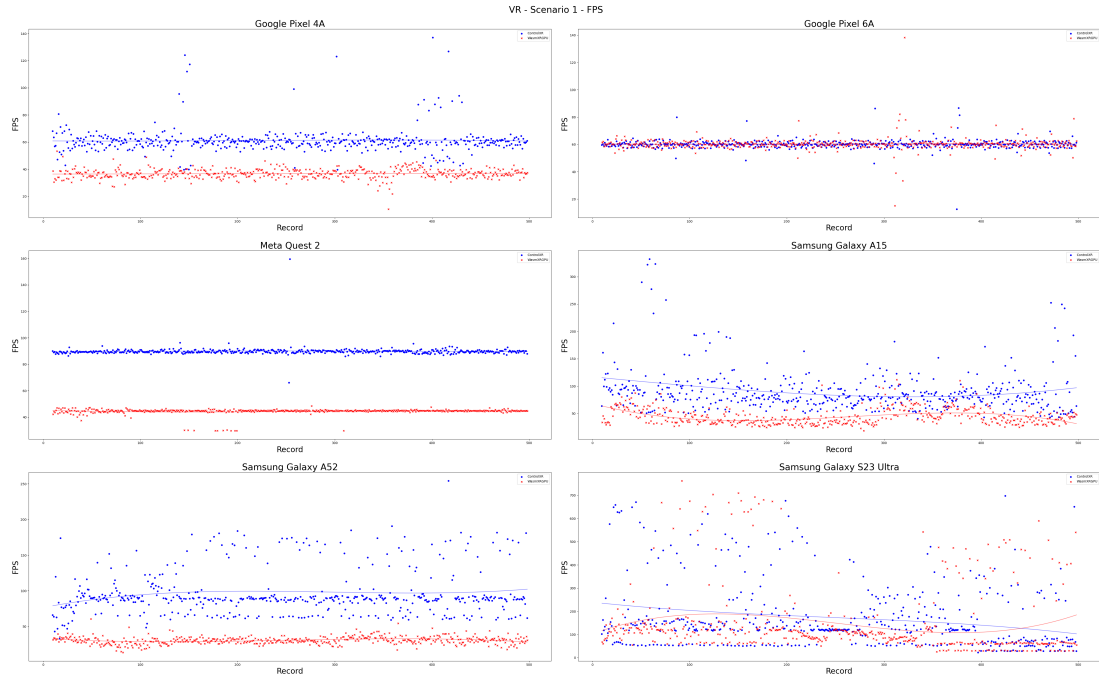


Figure 5.20: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.

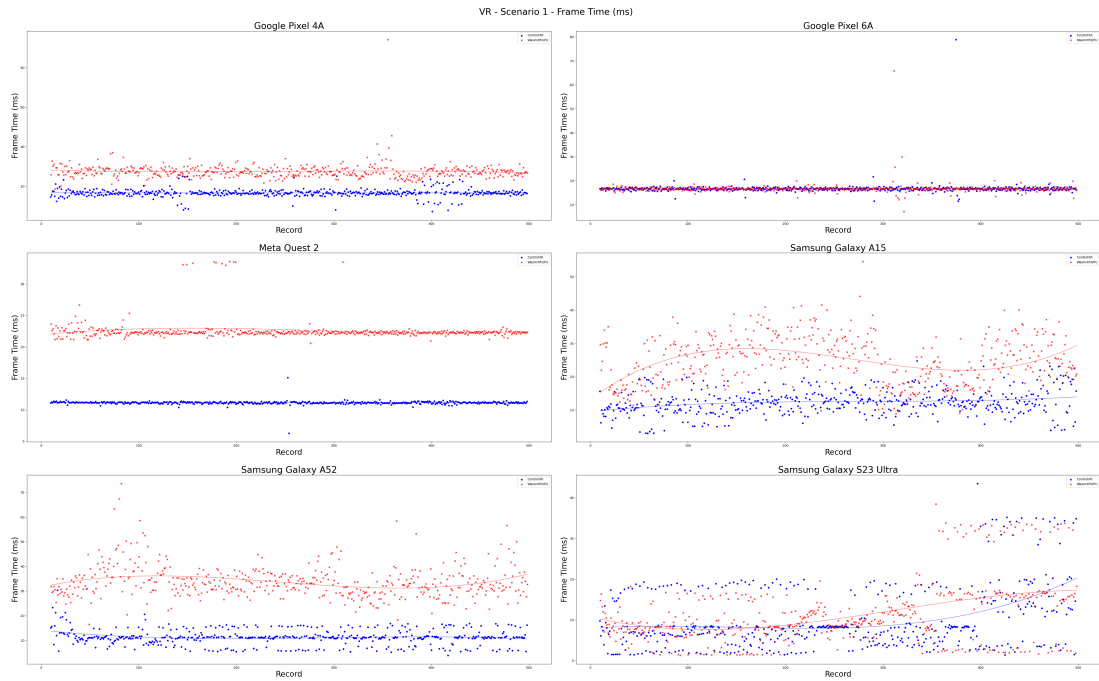


Figure 5.21: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

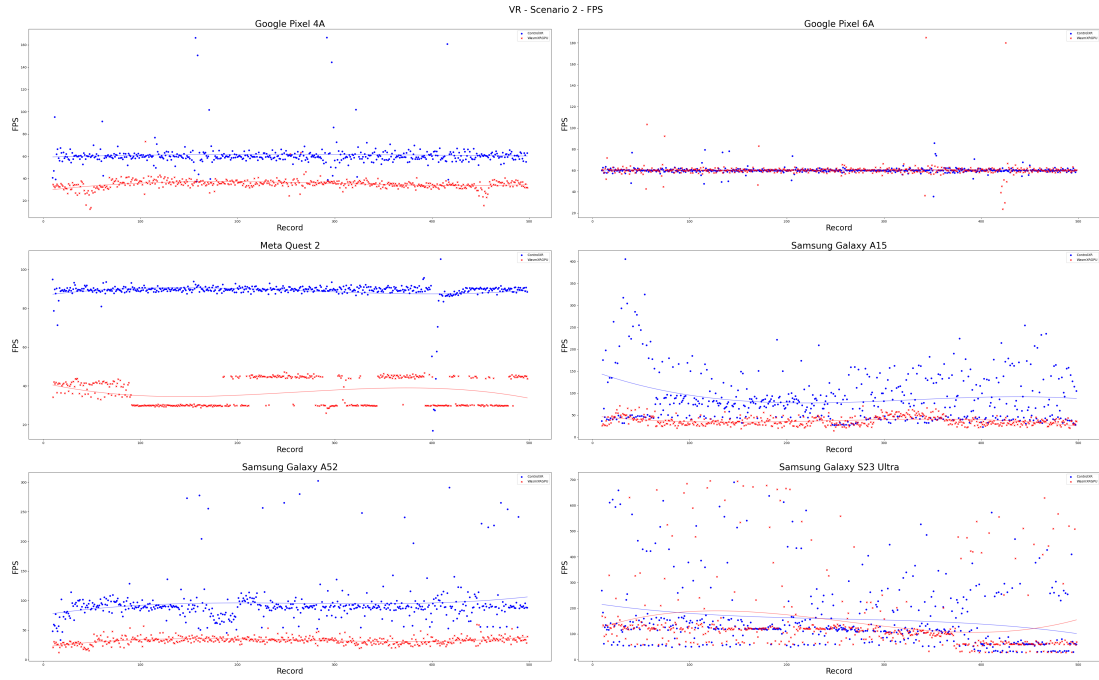


Figure 5.22: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.

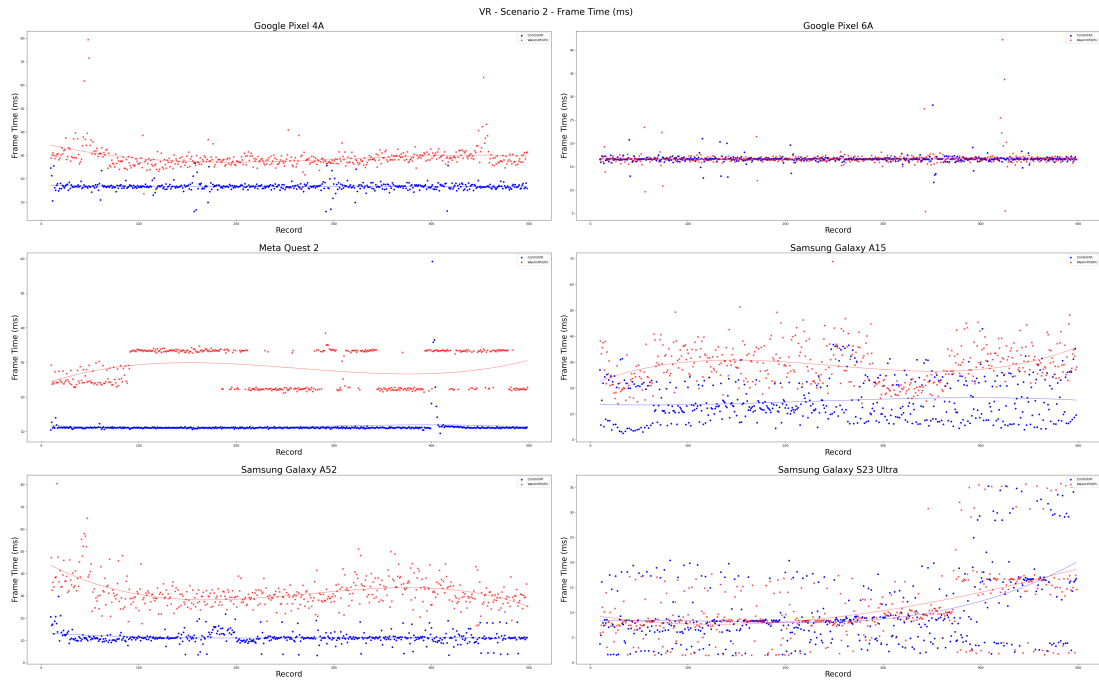


Figure 5.23: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

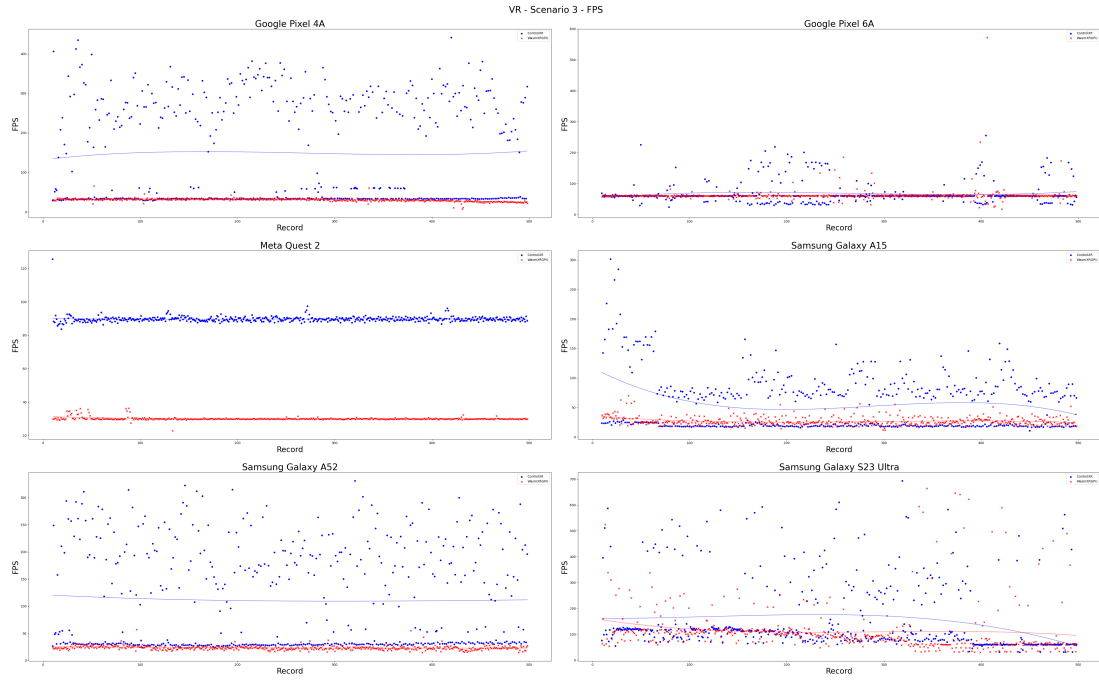


Figure 5.24: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.

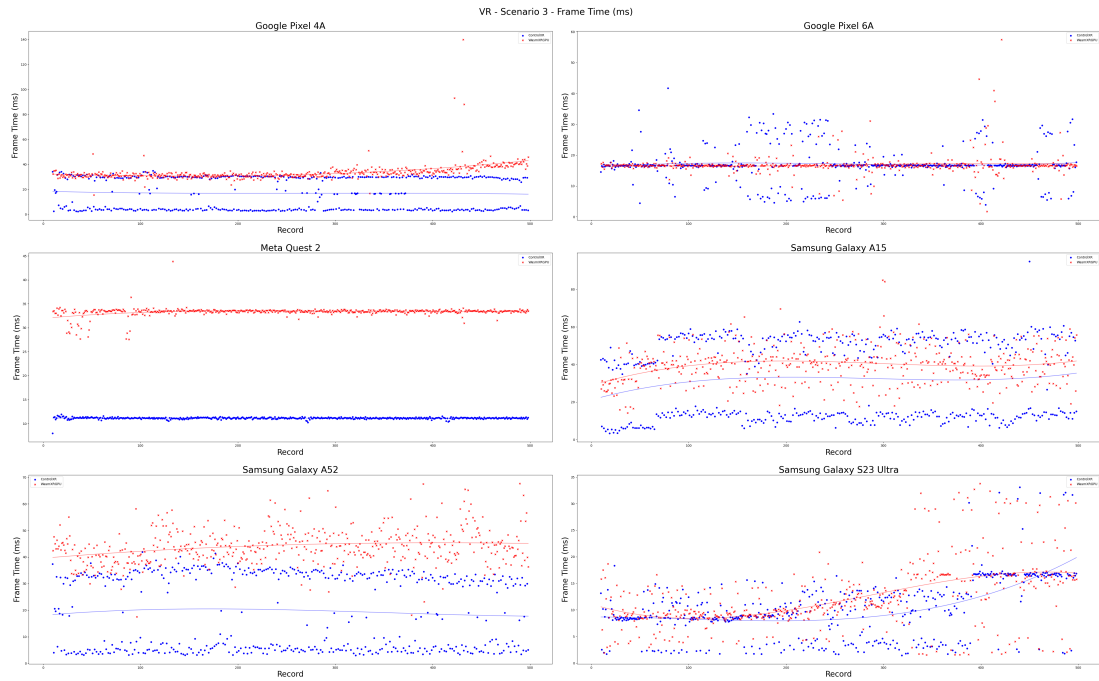


Figure 5.25: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

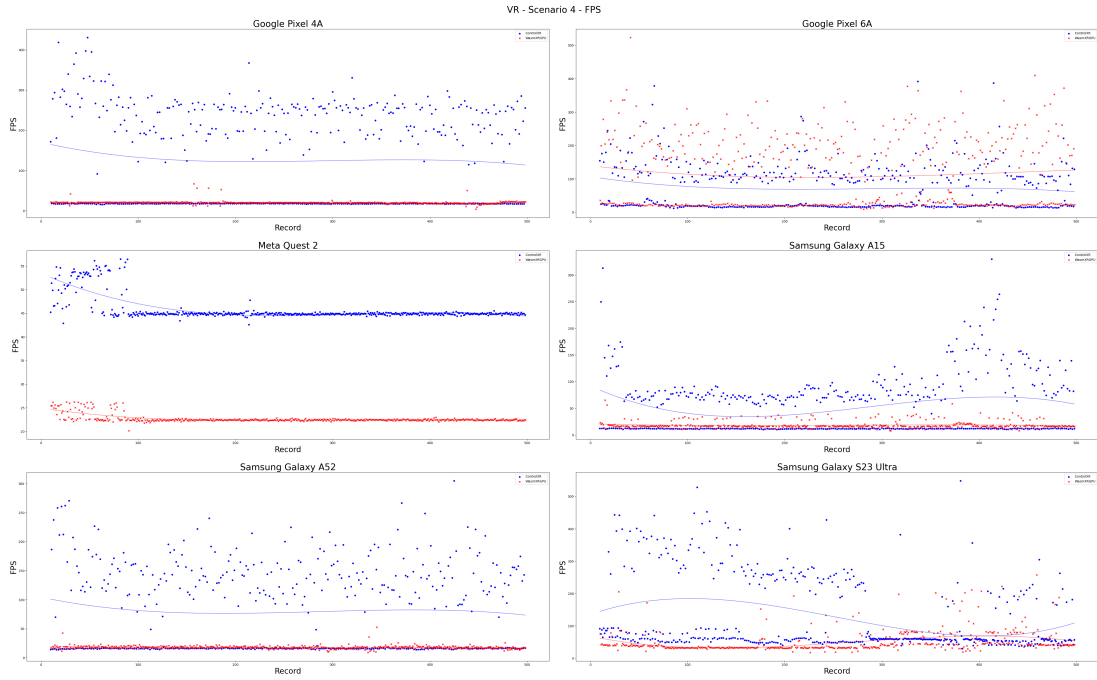


Figure 5.26: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.

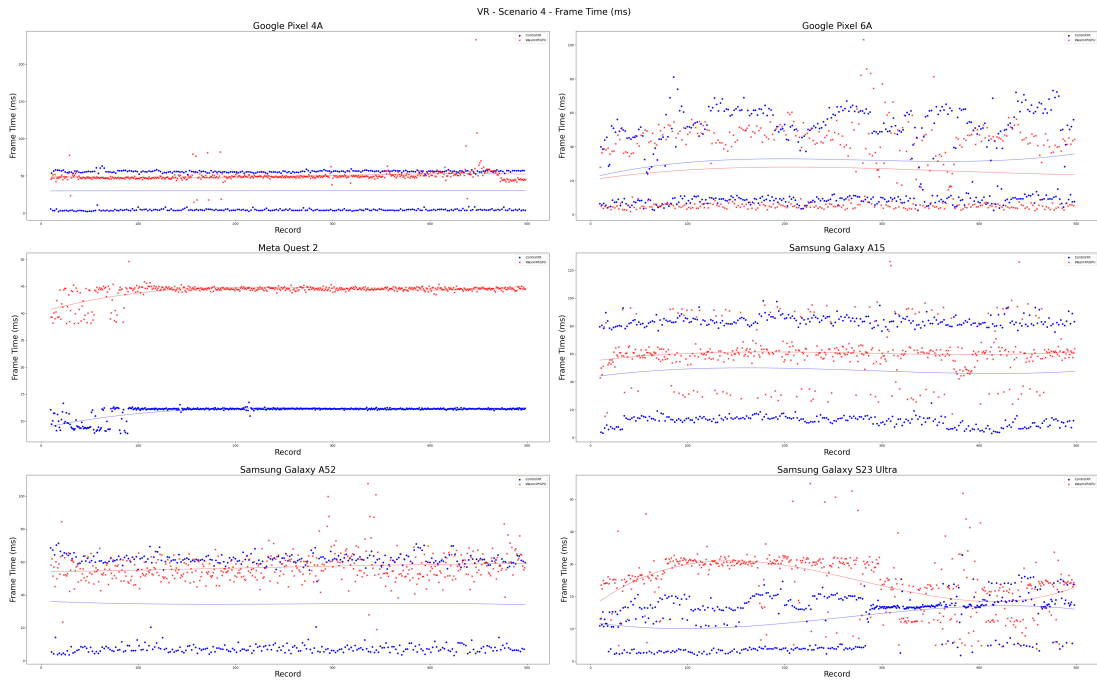


Figure 5.27: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

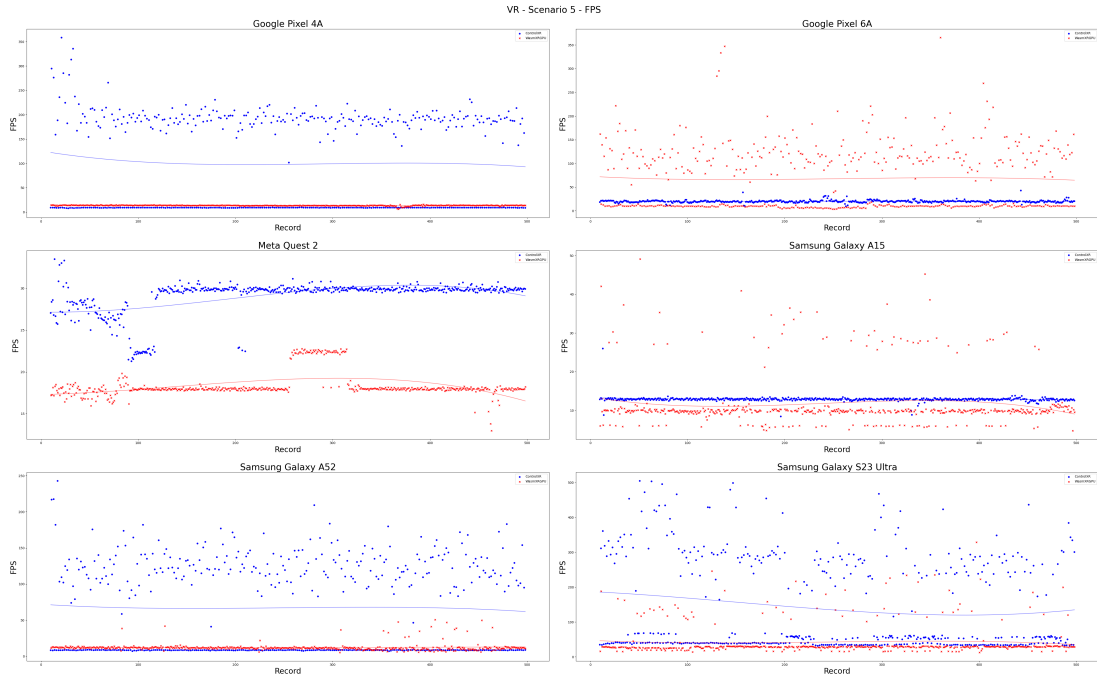


Figure 5.28: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.

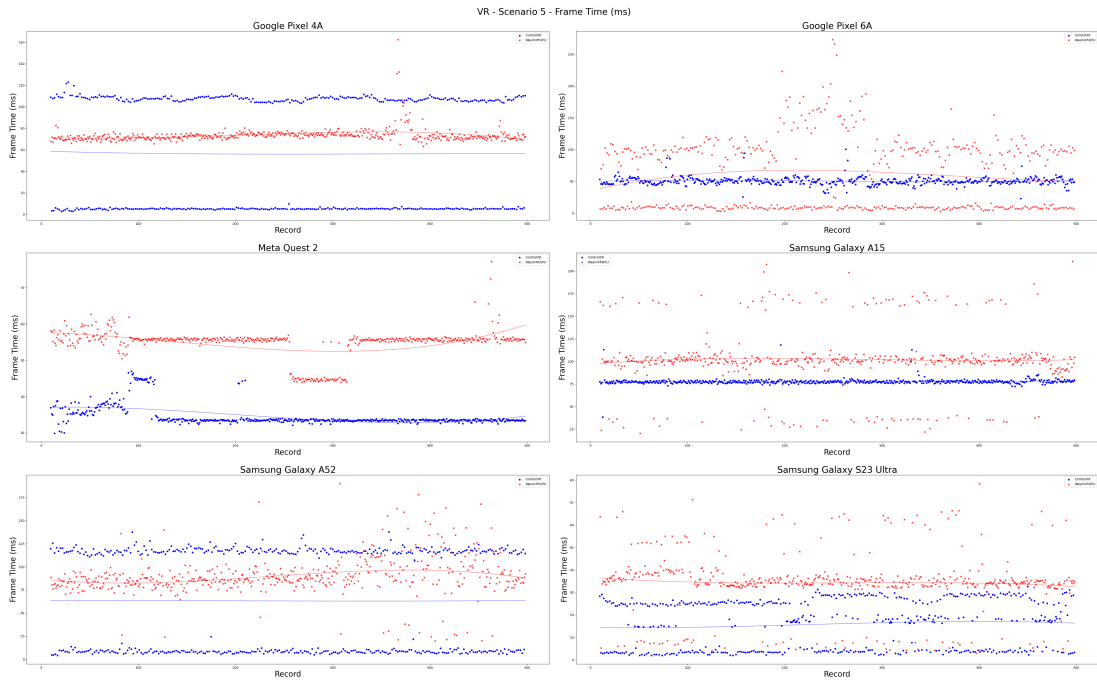


Figure 5.29: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

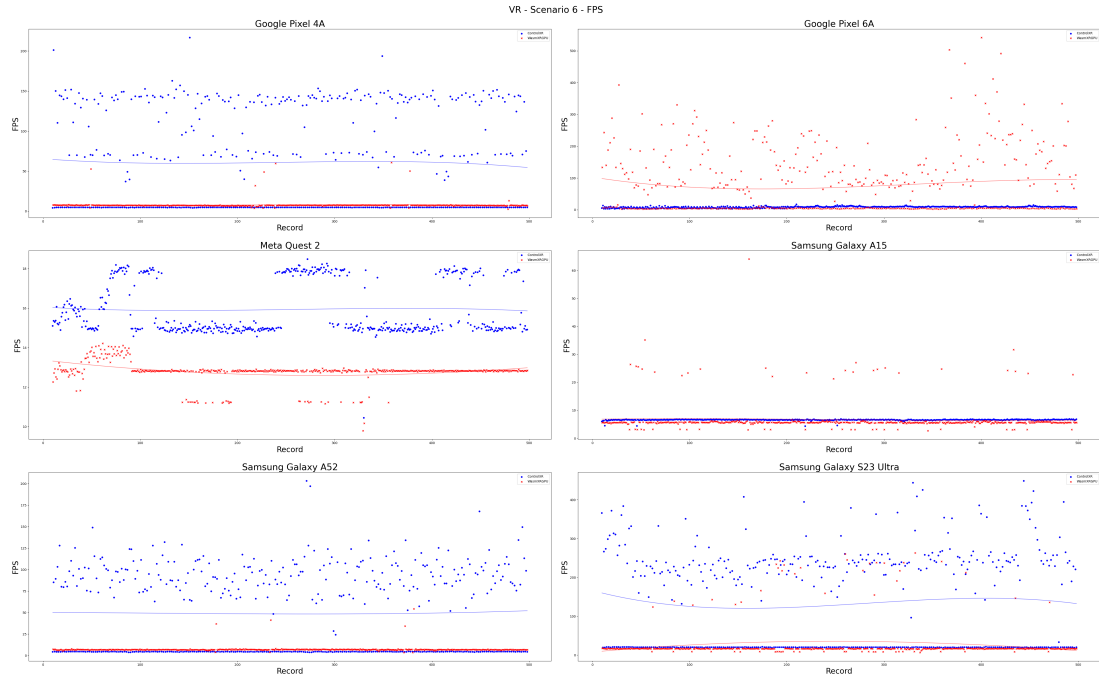


Figure 5.30: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Higher is better.

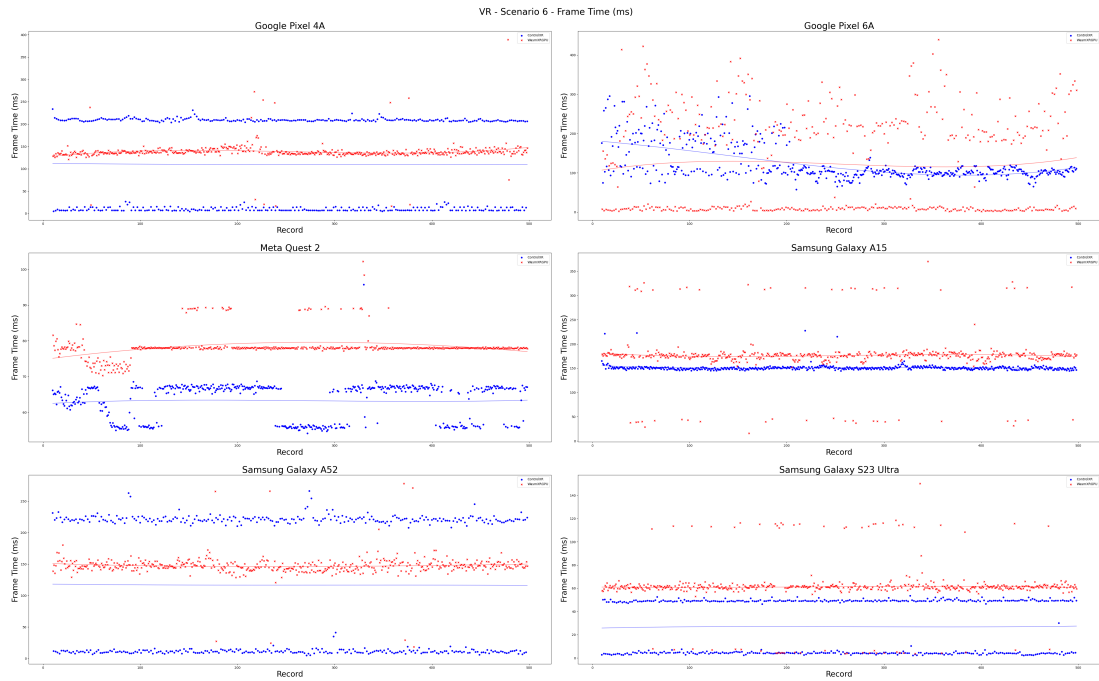


Figure 5.31: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

complexities within the same hardware environment. To ensure that outliers do not skew the data, the first 10 records are excluded from the plots.

2.1) Google Pixel 4A

Figure 5.32 and Figure 5.33 illustrate the distributions of FPS and Frame Time (ms) metric values for the Google Pixel 4A. In the case of ControlXR, the distributions show noticeable variation across different value ranges, suggesting fluctuating performance. In contrast, WasmXRGPU exhibits a more concentrated distribution, with most values clustered around a single range, although a few significant outliers are observed in the Frame Time metric.

2.2) Google Pixel 6A

Figure 5.34 and Figure 5.35 illustrate the distributions of FPS and Frame Time (ms) metric values for the Google Pixel 6A. In contrast to the Google Pixel 4A, the results for this device reveal an opposite trend—WasmXRGPU displays a wider distribution of values, indicating higher variability in performance, whereas ControlXR shows a more concentrated and consistent performance profile. However, in the scenarios with higher complexities, both exhibit outliers.

2.3) Samsung Galaxy A15

Figure 5.36 and Figure 5.37 illustrate the distributions of FPS and Frame Time (ms) metric values for the Samsung Galaxy A15. For this device, ControlXR's FPS values appear widely dispersed without any clear concentration, while the Frame Time values are grouped into two distinct ranges in scenario 3 and 4. In contrast, WasmXRGPU exhibits more stable and consistent distributions across in FPS, whereas slight variation in Frame Time.

2.4) Samsung Galaxy A52

Figure 5.38 and Figure 5.39 illustrate the distributions of FPS and Frame Time (ms) metric values for the Samsung Galaxy A52. The instability in ControlXR becomes more pronounced on this device, with metric values fluctuating across multiple distinct ranges. In contrast, WasmXRGPU maintains stable performance, with values tightly clustered and only a few minor outliers.

2.5) Samsung Galaxy S23 Ultra

Figure 5.40 and Figure 5.41 illustrate the distributions of FPS and Frame Time (ms) metric values for the Samsung Galaxy S23 Ultra. Similar to the Samsung Galaxy A52, ControlXR exhibits greater variability in both metrics as the scenario complexity increases. In contrast, WasmXRGPU maintains more consistent performance, with values concentrated within a narrower range.

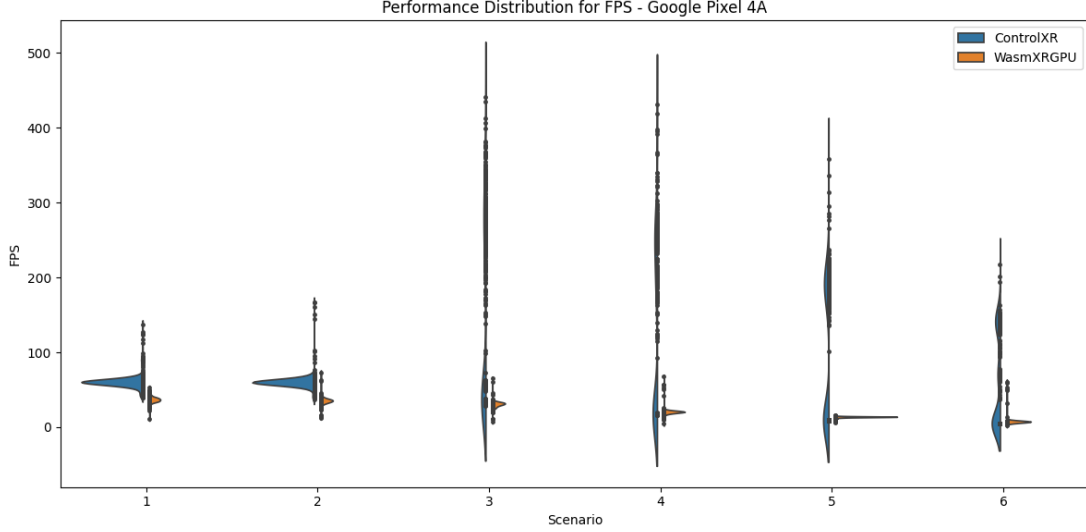


Figure 5.32: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Virtual Reality (VR)

2.6) Meta Quest 2

Figure 5.42 and Figure 5.43 illustrate the distributions of FPS and Frame Time (ms) metric values for the Meta Quest 2. As a dedicated XR device, its data distribution significantly differs from that observed in mobile devices. ControlXR demonstrates consistent performance with virtually no variation across both metrics, whereas WasmXRGPU exhibits slight variability in certain scenes, reflected by a few distinct distribution ranges.

Session Load Time

In this section, the time required to load the very first frame is analyzed by examining the frame time of the first recorded entry. To provide a comparative view between the initial and subsequent frames, the first 10 recorded frame times are visualized using bar plots. These visualizations are shown in Figure 5.44 through Figure 5.49, corresponding to each scenario respectively. No consistent or distinguishable patterns were observed across different devices and scenarios. However, in the majority of cases, ControlXR demonstrated better initial performance than WasmXRGPU.

5.2.2 Augmented Reality (AR)

The AR sessions for each scenario listed in Table 5.3 were rendered independently in both experimental groups. However, the WasmXRGPU implementation did not function as expected on the Google Pixel 6A device, resulting in the inability

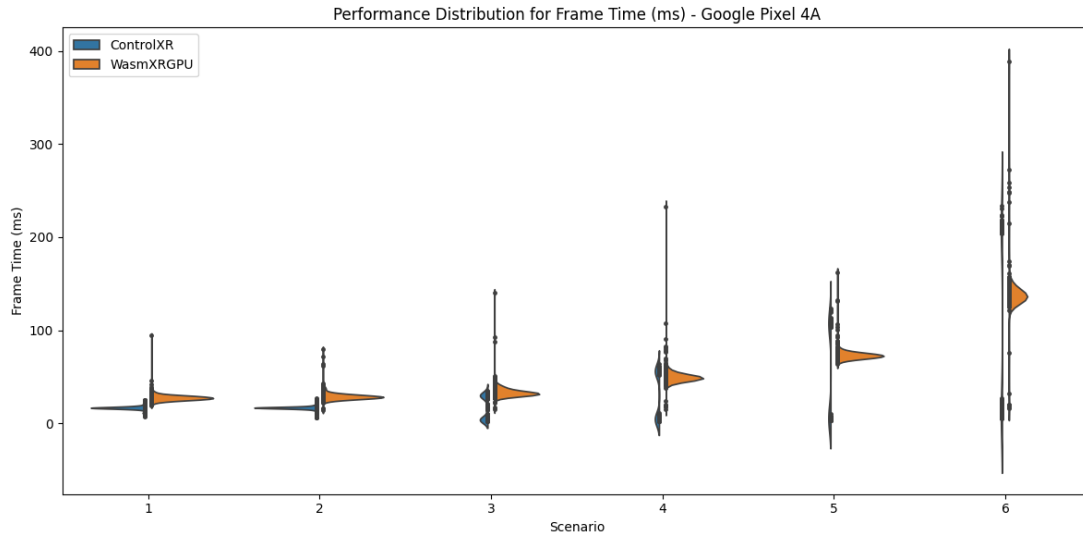


Figure 5.33: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Virtual Reality (VR)

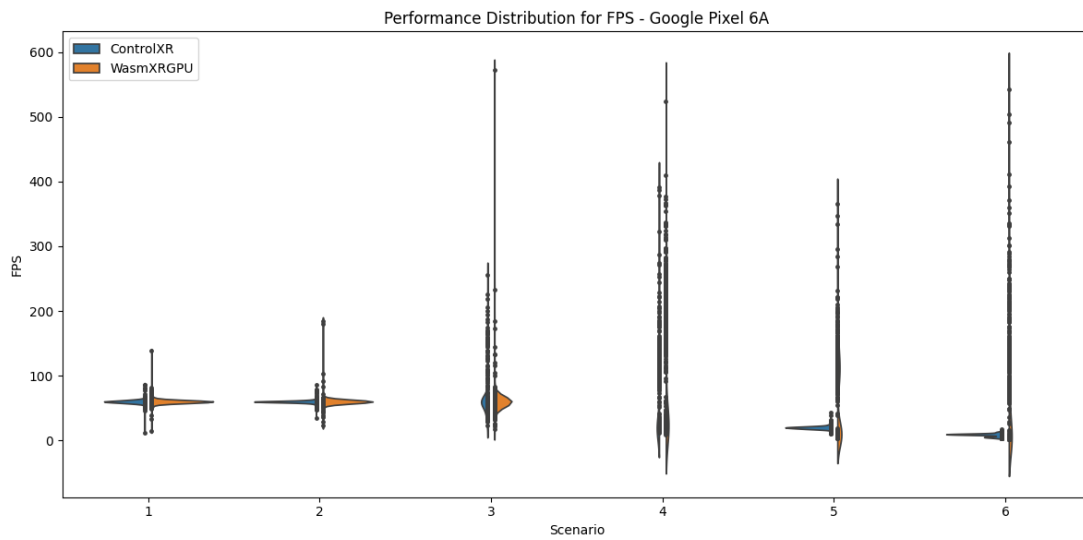


Figure 5.34: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Virtual Reality (VR)

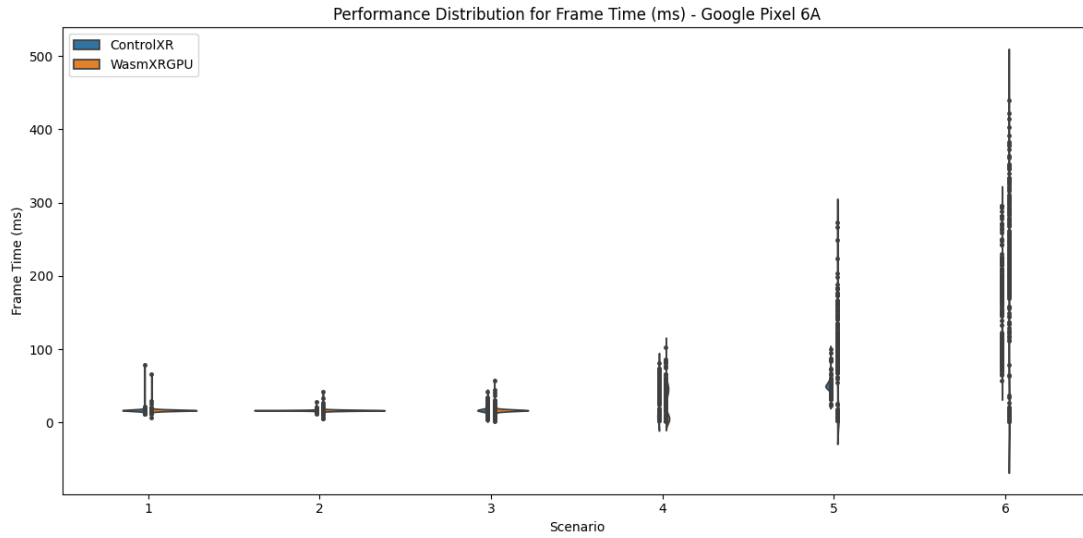


Figure 5.35: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Virtual Reality (VR)

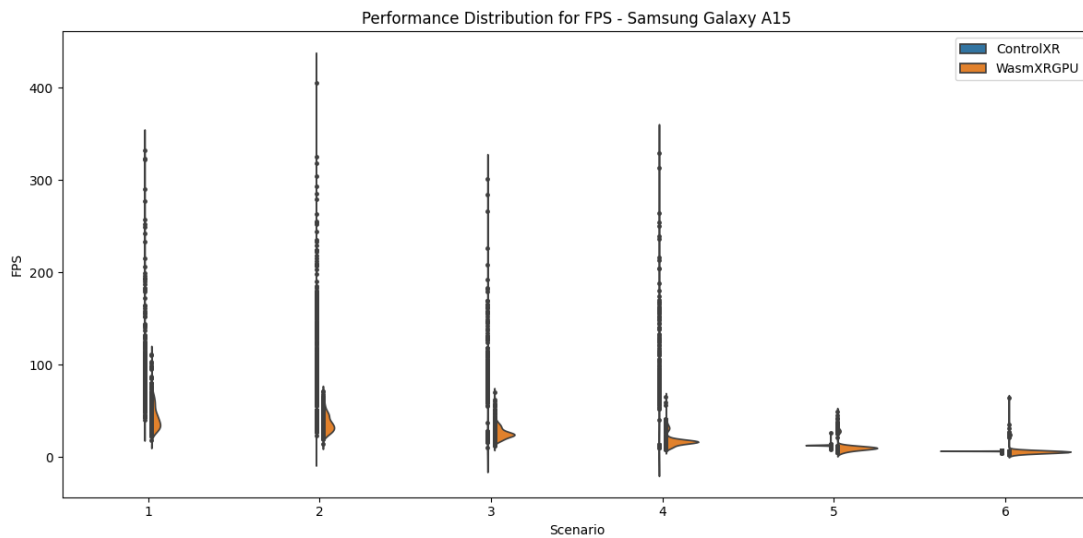


Figure 5.36: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Virtual Reality (VR)

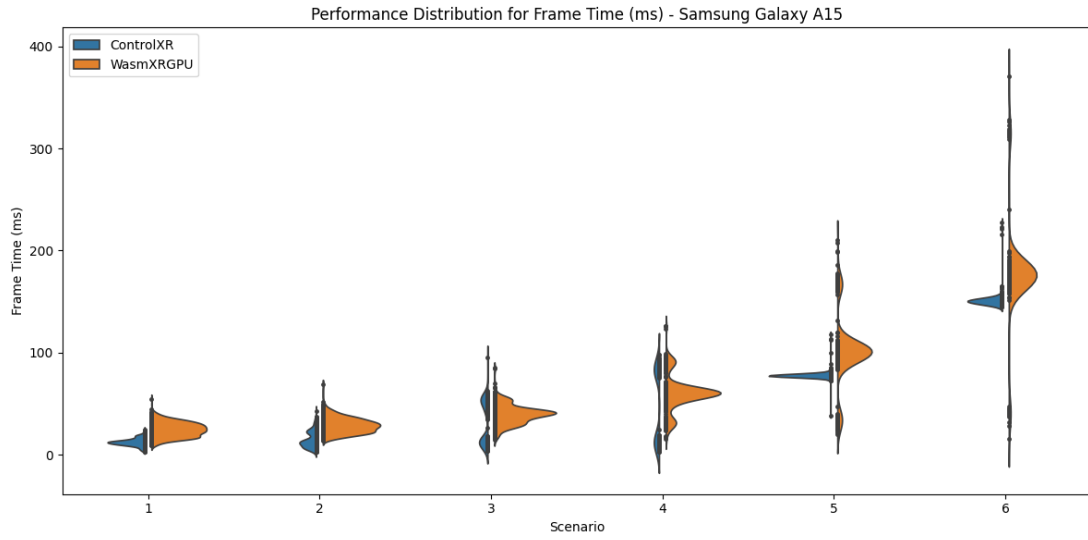


Figure 5.37: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Virtual Reality (VR)

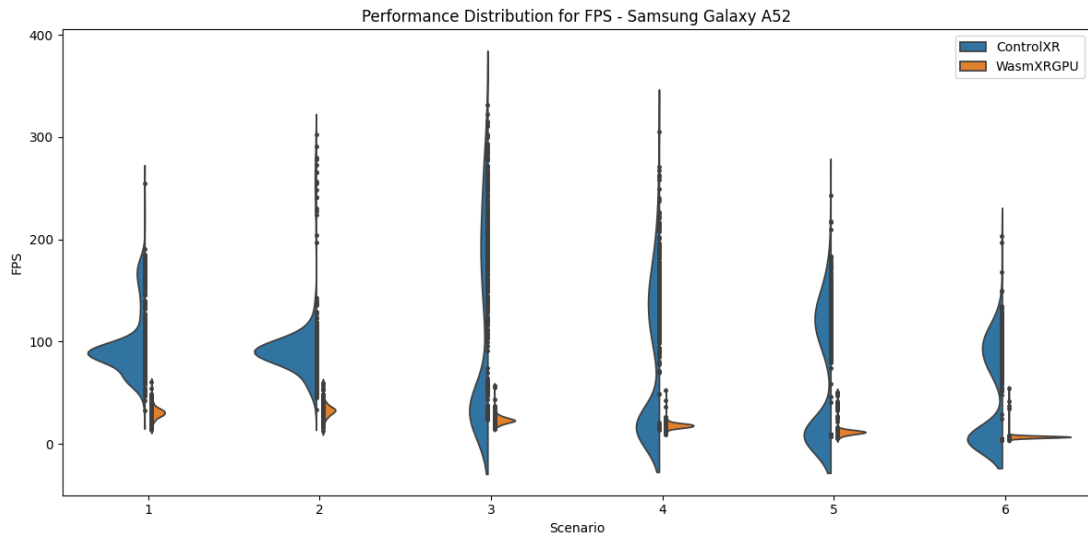


Figure 5.38: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Virtual Reality (VR)

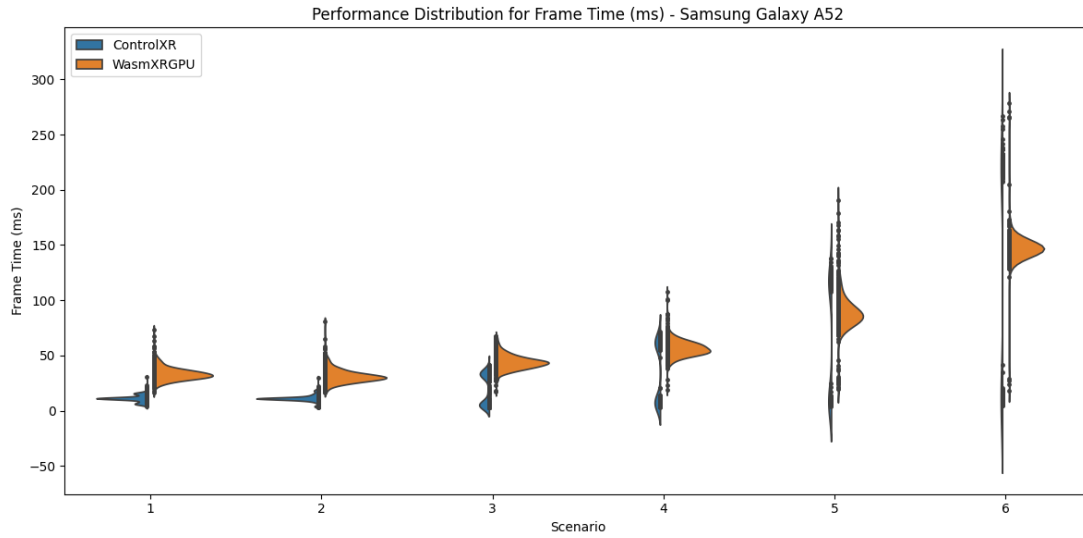


Figure 5.39: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Virtual Reality (VR)

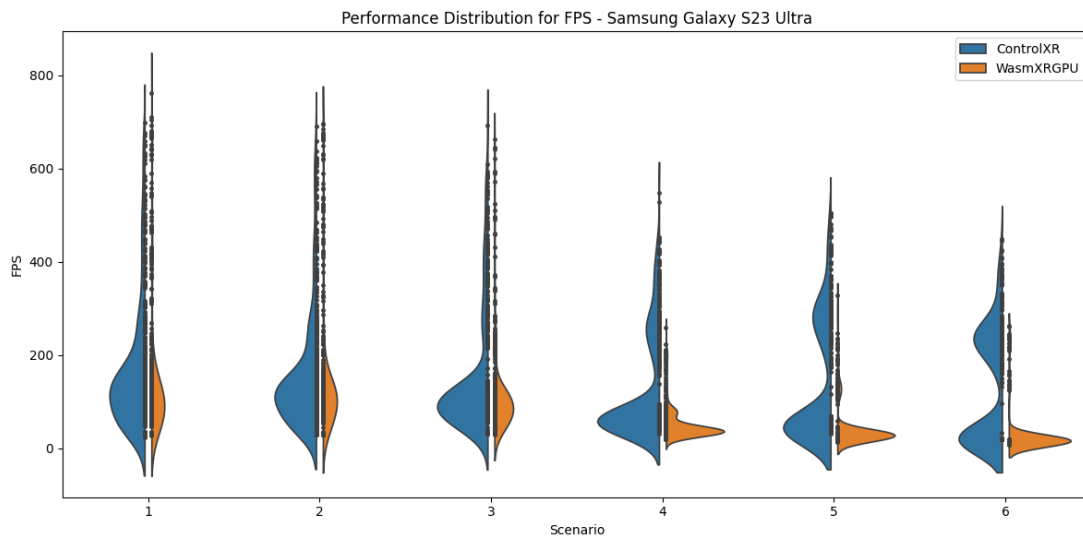


Figure 5.40: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Virtual Reality (VR)

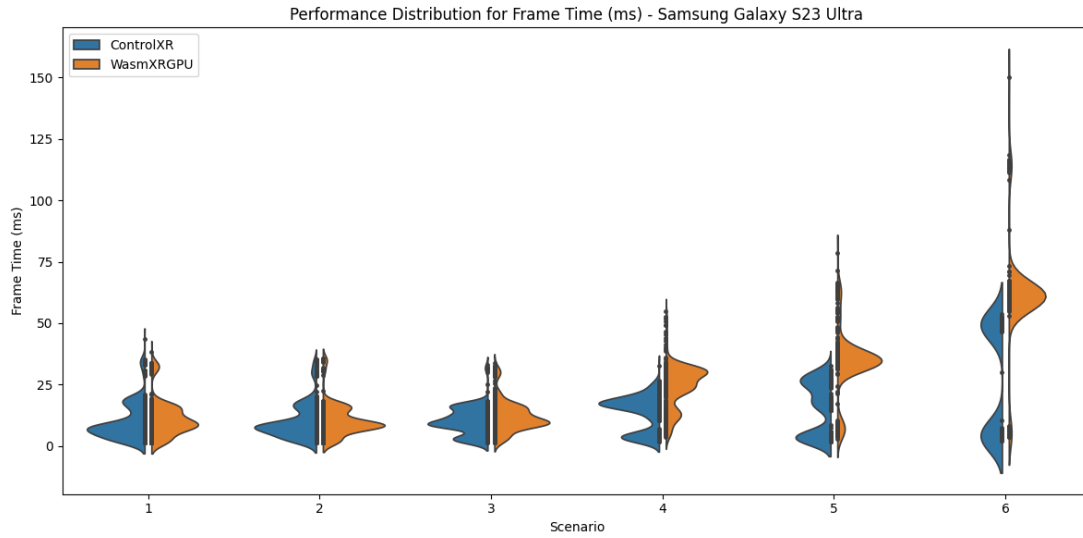


Figure 5.41: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Virtual Reality (VR)

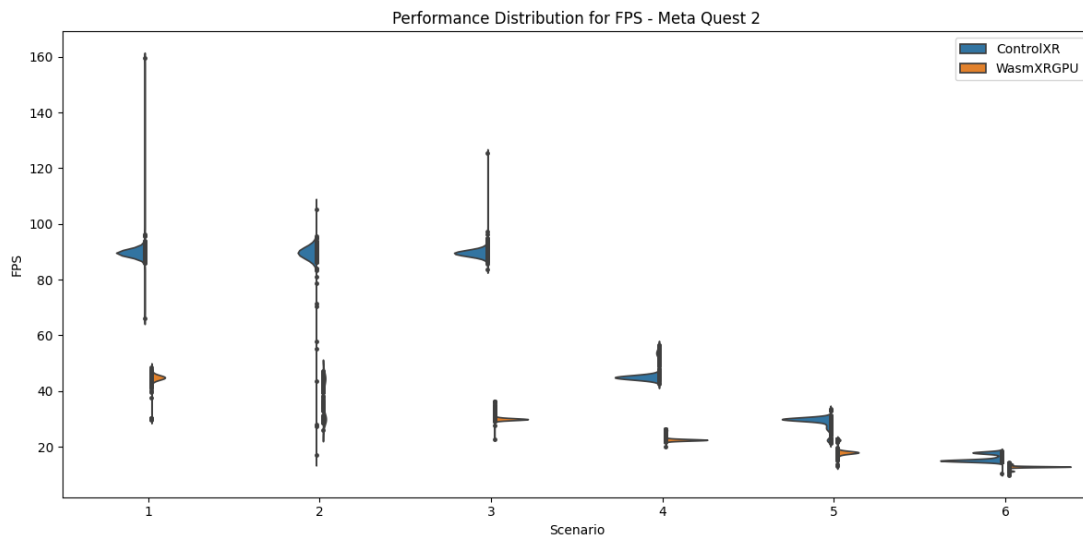


Figure 5.42: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Virtual Reality (VR)

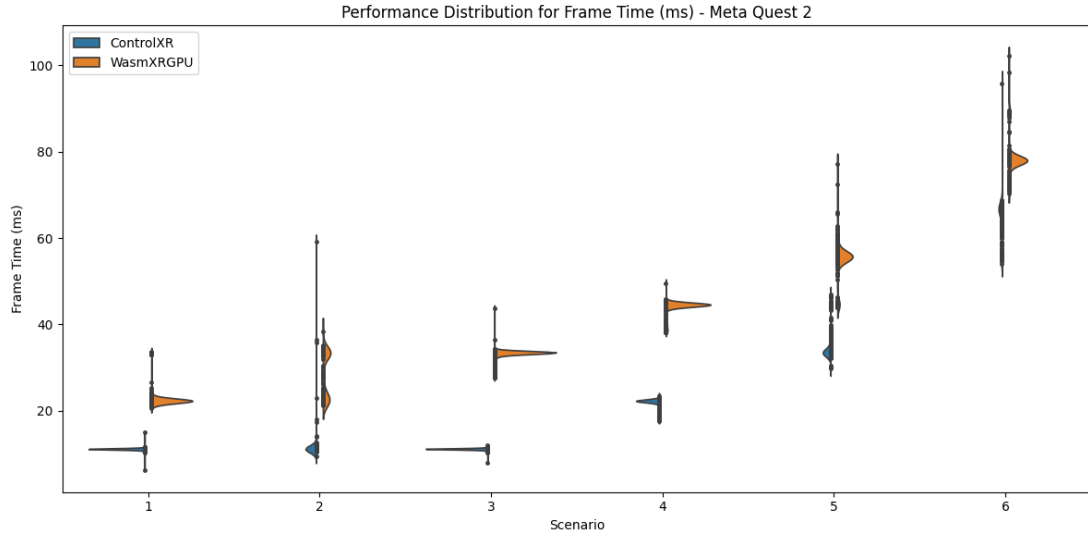


Figure 5.43: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Virtual Reality (VR)

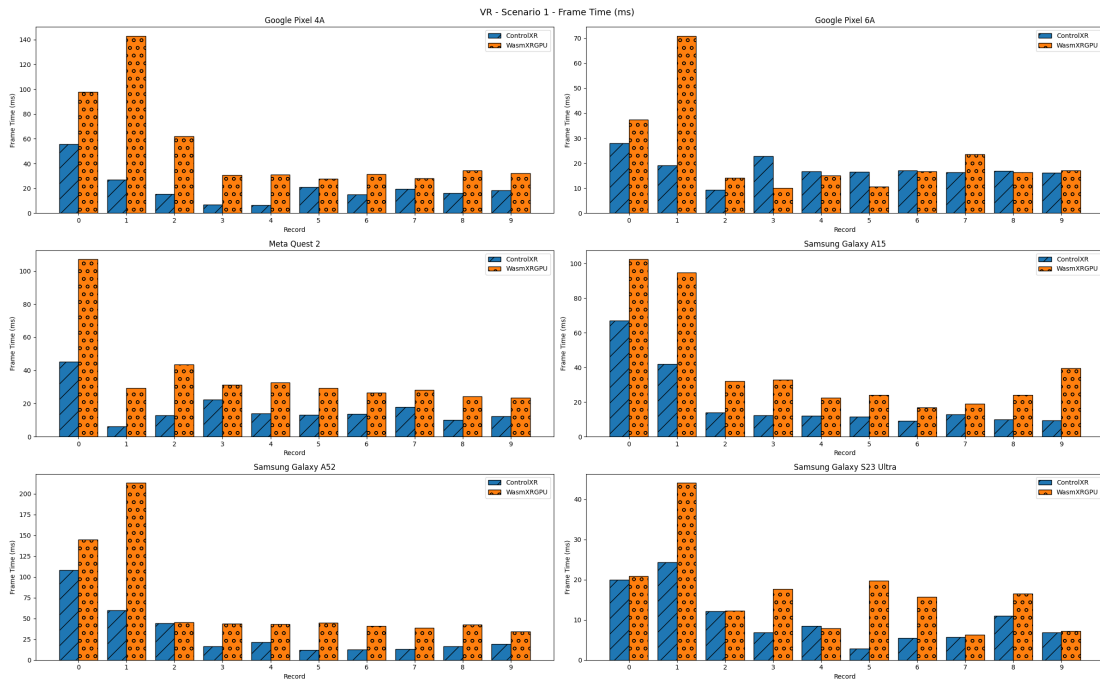


Figure 5.44: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 1, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).

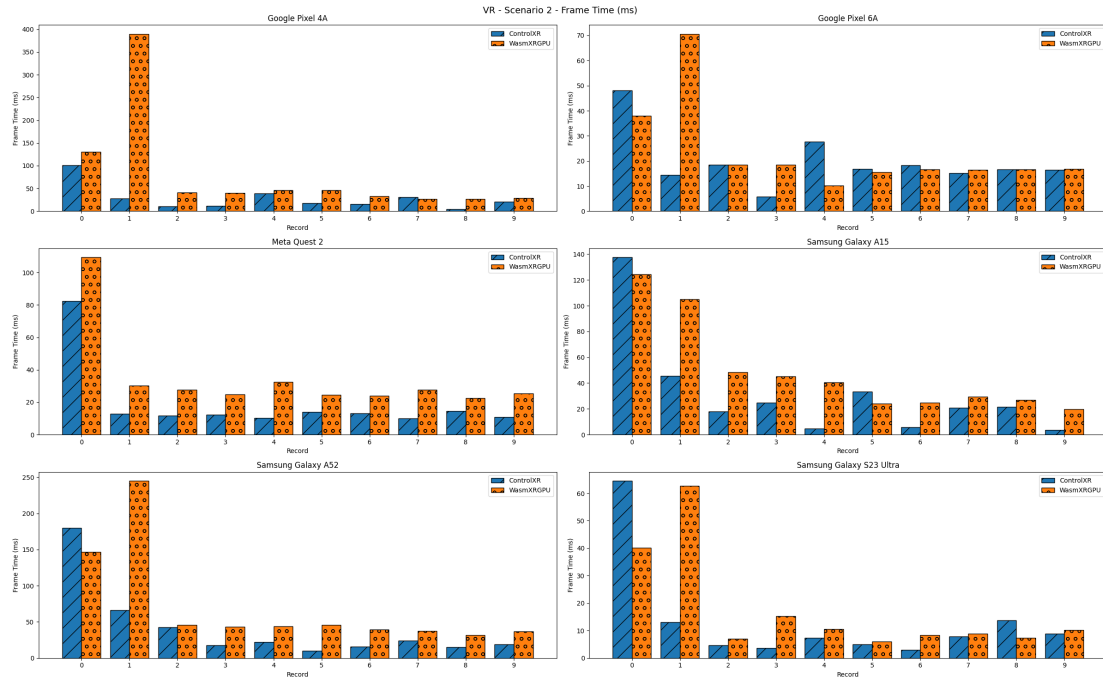


Figure 5.45: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 2, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).

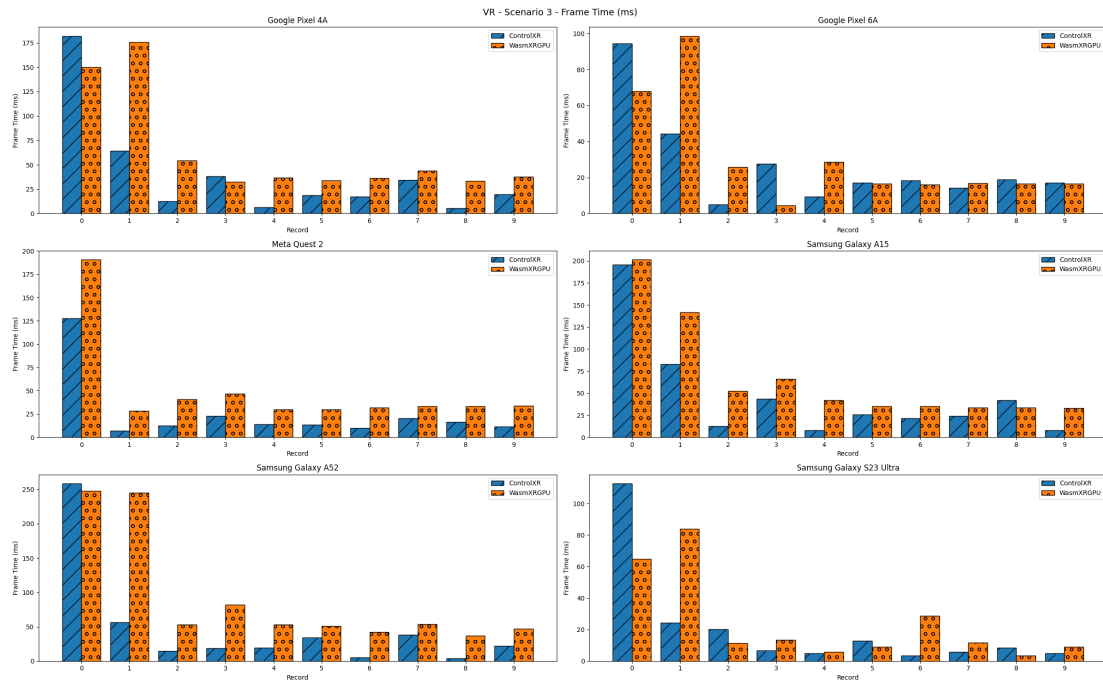


Figure 5.46: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 3, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).

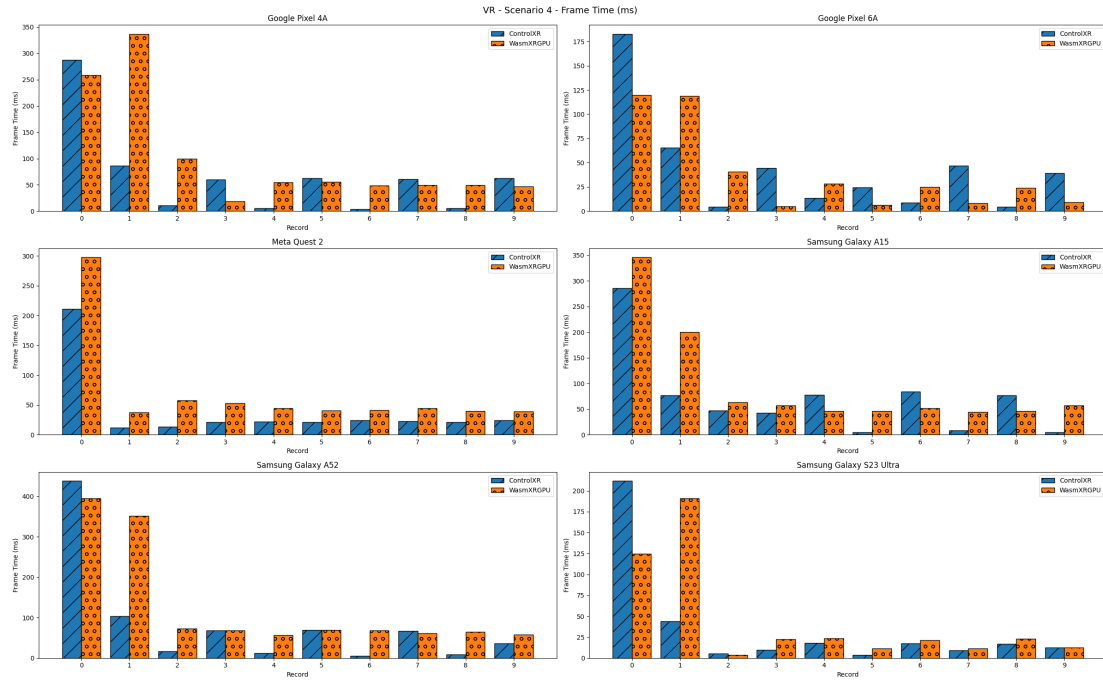


Figure 5.47: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 4, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).

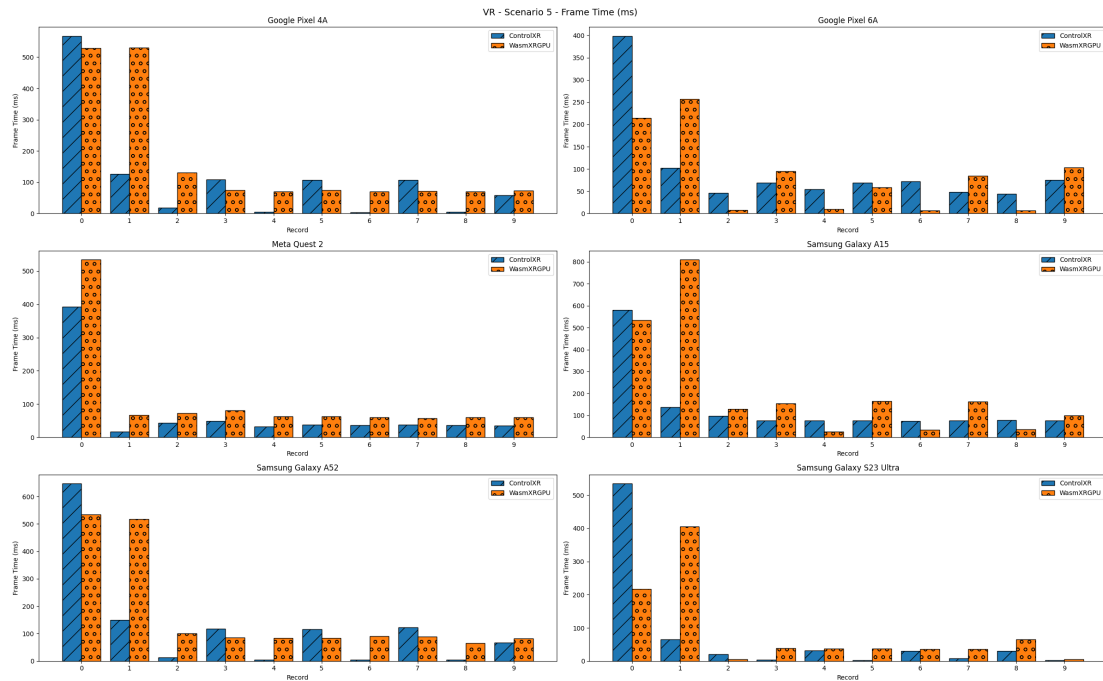


Figure 5.48: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 5, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).

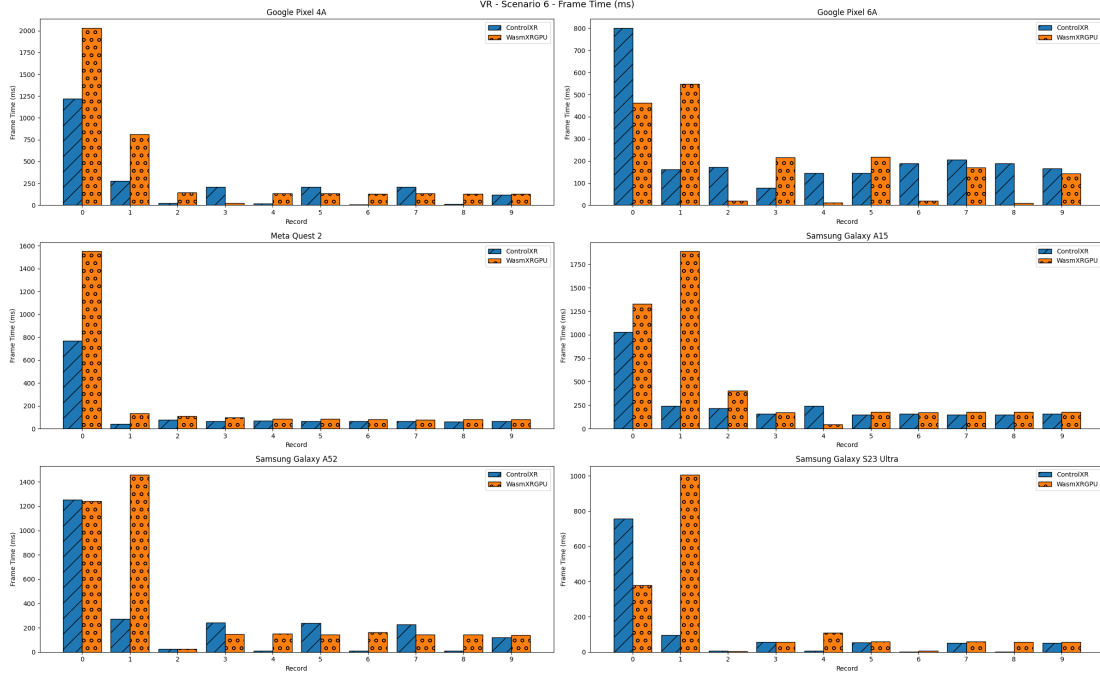


Figure 5.49: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 6, across the devices listed in Table 5.4, in the context of Virtual Reality (VR).

to record performance metric data for that configuration.

Average Performance

The recorded results are averaged over the 500-frame capture window and plotted to visualize the average performance. The captured results are visualized and compared between the two implementations for each device.

1) Average CPU Time

Figure 5.50 illustrates the average CPU time across all tested devices for each scenario. As the scene complexity increases, a clear performance trend emerges: CPU time in WasmXRGPU implementation consistently improves relative to ControlXR in all devices except Google Pixel 4A. The average CPU time values for each device are visualized in a heatmap form which is depicted in Figure 5.56.

2) Average GPU Time

Figure 5.51 illustrates the average GPU time across all tested devices for each scenario. As scene complexity increases, a clear performance trend emerges: contrary to the improvements observed in CPU time, the GPU time in the WasmXRGPU implementation increases significantly compared to ControlXR in all devices. The average GPU time values for each device are visualized in a heatmap form which

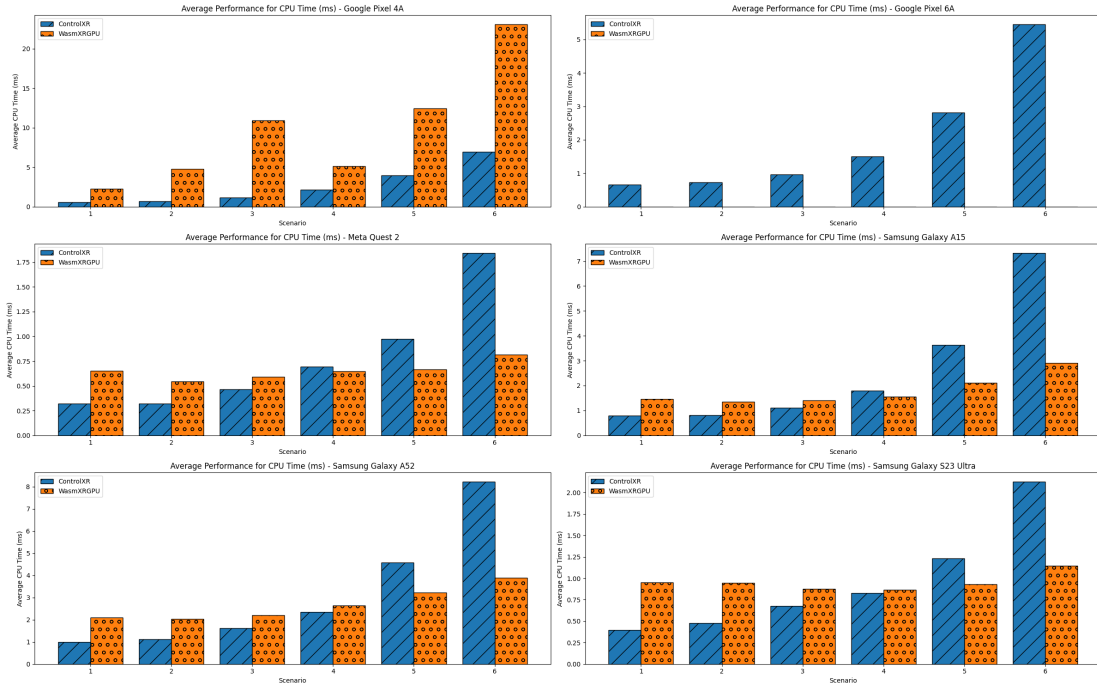


Figure 5.50: Comparison of Average CPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

is depicted in Figure 5.55.

3) Average Framerate Per Second (FPS)

Figure 5.52 illustrates the average FPS across all tested devices for each scenario. As scene complexity increases, both implementations exhibit a gradual decline in FPS, which aligns with expected performance trends. However, the Samsung Galaxy S23 Ultra maintains consistent performance across all scenarios, showing no noticeable FPS degradation. The average FPS values for each device are visualized in a heatmap form which is depicted in Figure 5.56.

4) Average Frame Time

Figure 5.53 illustrates the average frame time across all tested devices for each scenario. As scene complexity increases, both implementations exhibit a gradual rise in frame time, which is consistent with expected performance behavior. Similar to the FPS results, the Samsung Galaxy S23 Ultra maintains consistent performance across all scenarios, showing no significant variation in frame time. The average frame time values for each device are visualized in a heatmap form which is depicted in Figure 5.57.

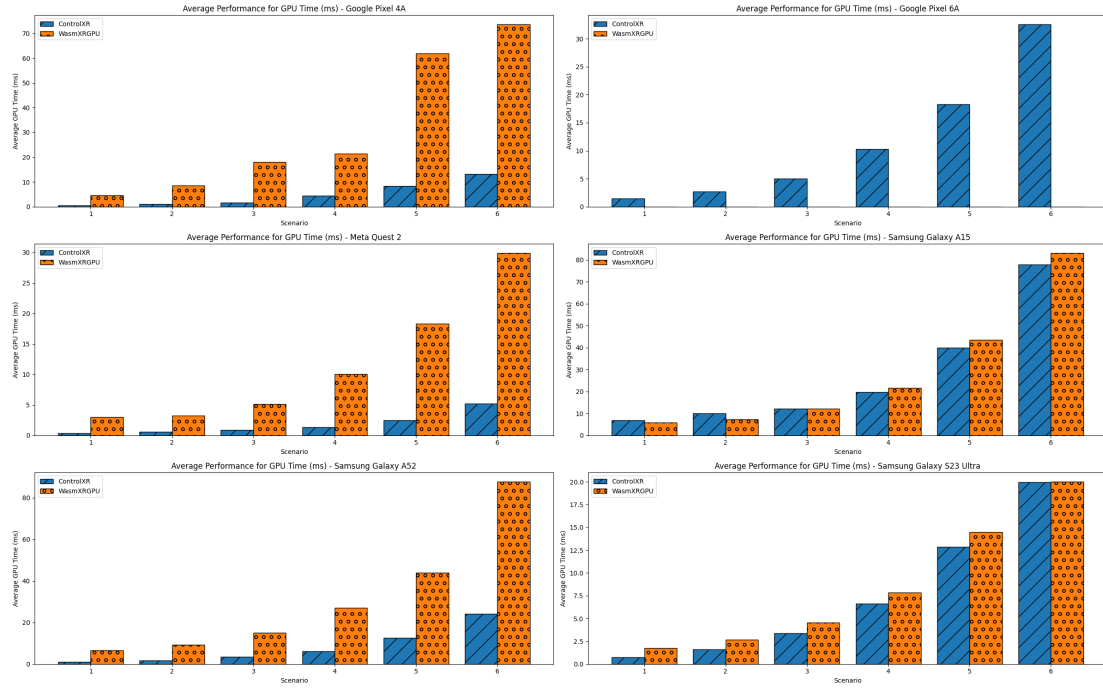


Figure 5.51: Comparison of Average GPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

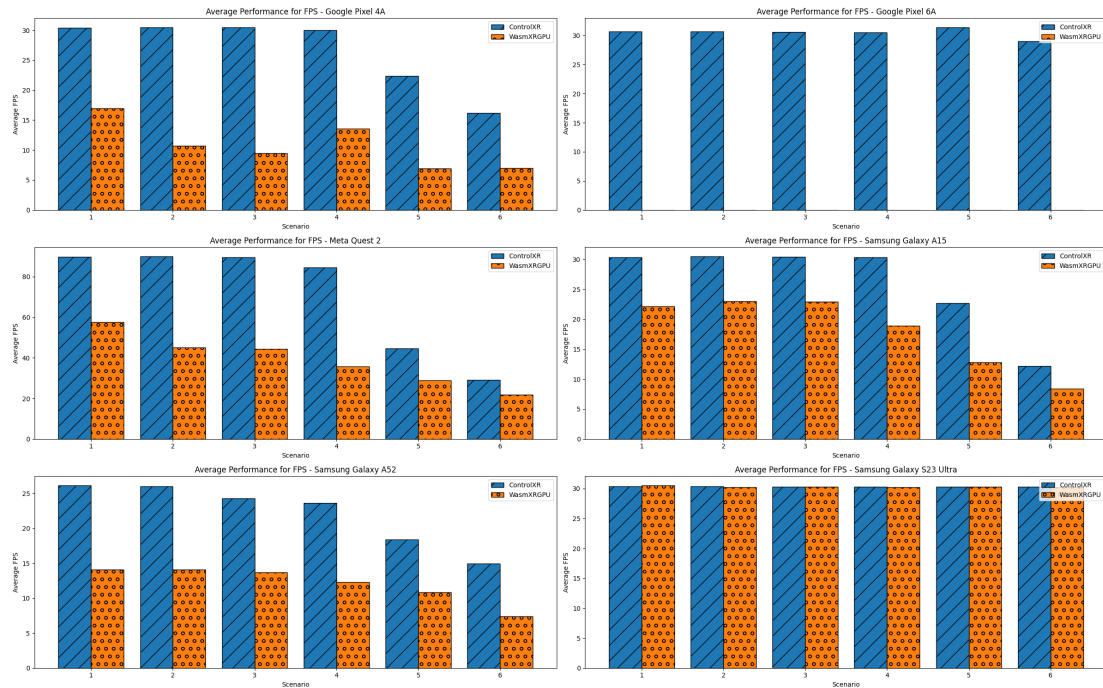


Figure 5.52: Comparison of Average FPS between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.

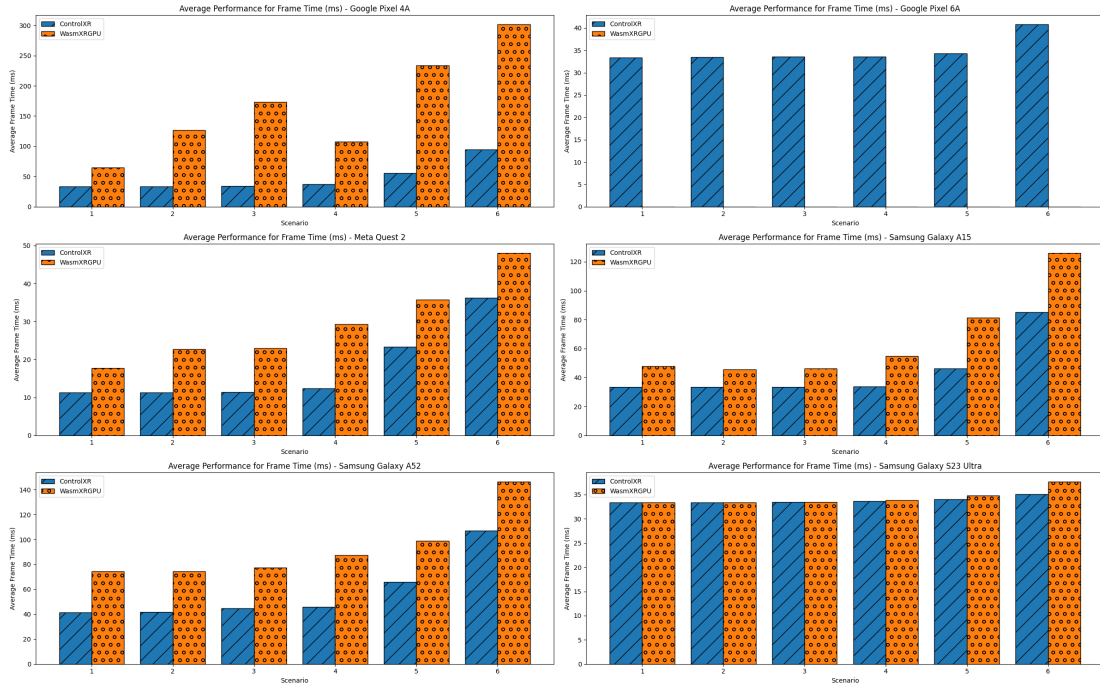


Figure 5.53: Comparison of Average Frame Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

Per-Frame Performance

The average values provided a general overview of overall performance. Unlike in the VR sessions, no clear anomalies were observed in the AR sessions. However, to explore the possibility of subtler irregularities and to better understand the distribution of performance metrics, scatter plots were used to visualize per-frame data.

The following plots present per-frame data for key performance metrics across different scenarios and devices. This detailed analysis helps to reveal performance variability and offers insights into any potential anomalies. For brevity, this section focuses on the FPS and Frame Time metrics, as they are directly correlated and provide a clear indication of runtime performance. Visualizations of all other metrics are included in the Appendix A.3.

1) Scatter Plot

This section will visualize all the recorded data using scatter plots. The first 10 records are omitted from the visualization, as they may skew the data representation. These initial 10 records correspond to the initialization phase of the experiment.

Figures 5.58 , 5.60 , 5.62 , 5.64 , 5.66, 5.68 present scatter plots of FPS

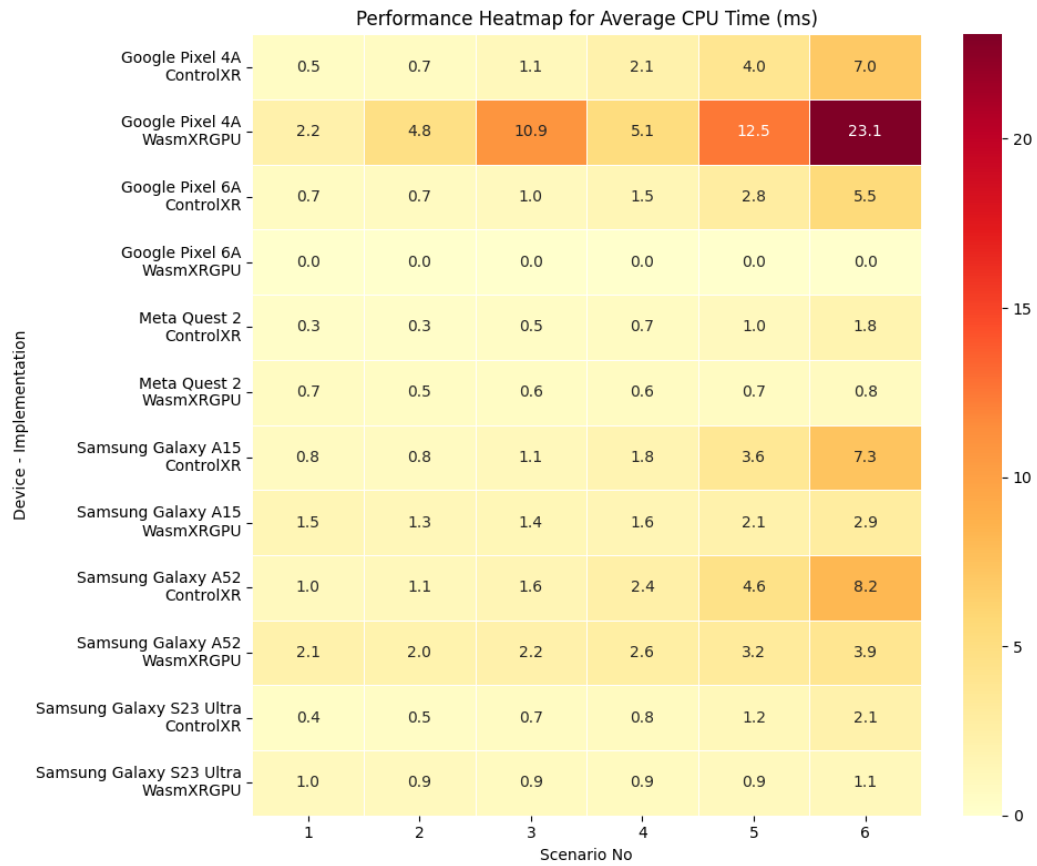


Figure 5.54: Heatmap Visualization of Average CPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower the color, better.

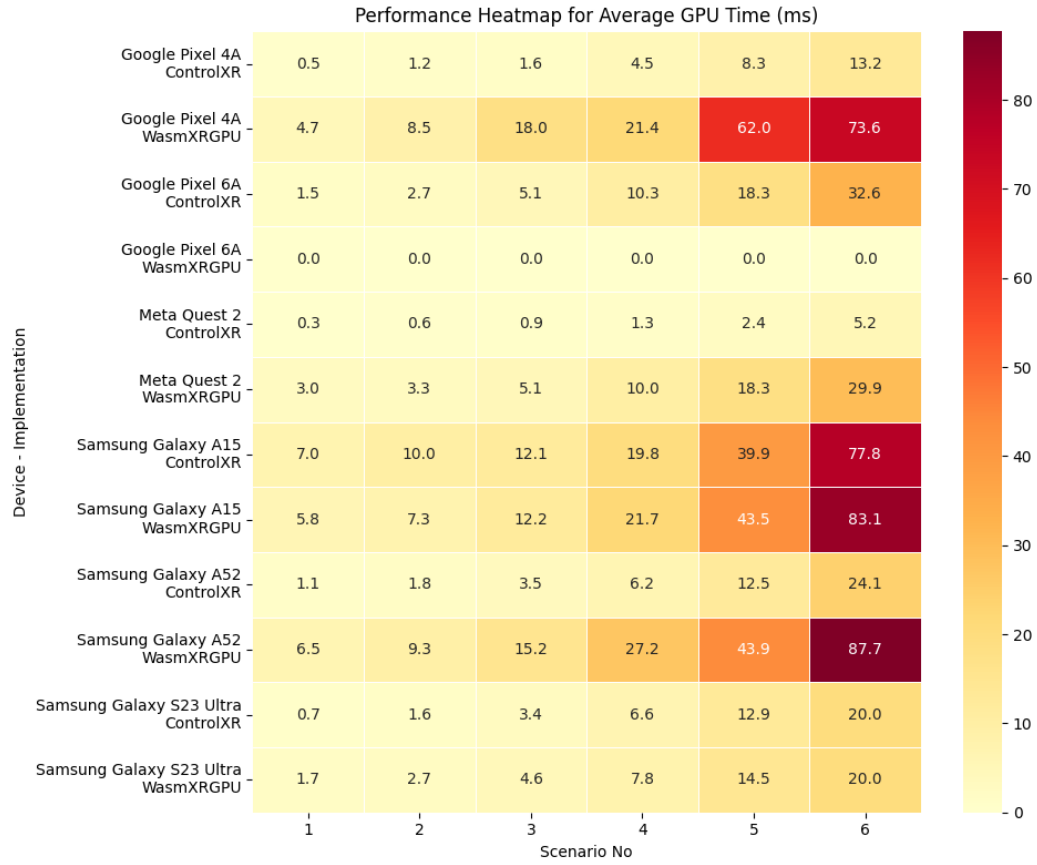


Figure 5.55: Heatmap Visualization of Average GPU Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower the color, is better.

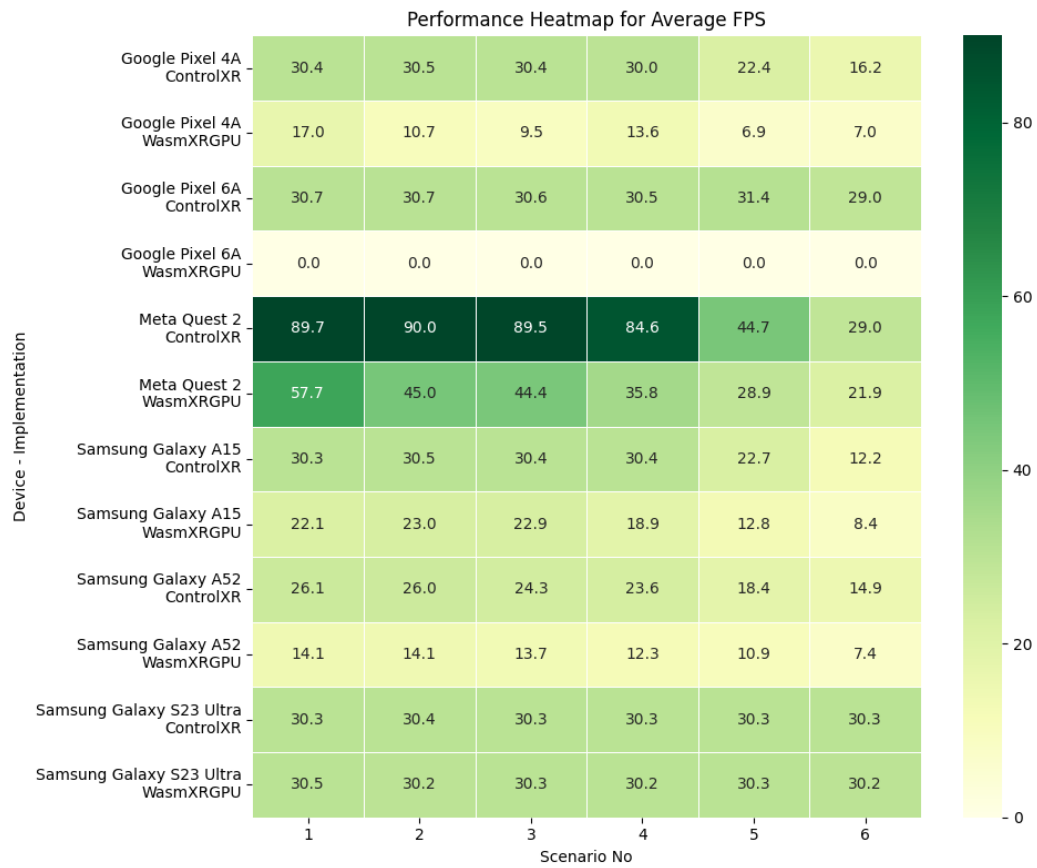


Figure 5.56: Heatmap Visualization of Average FPS between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher the color, is better.

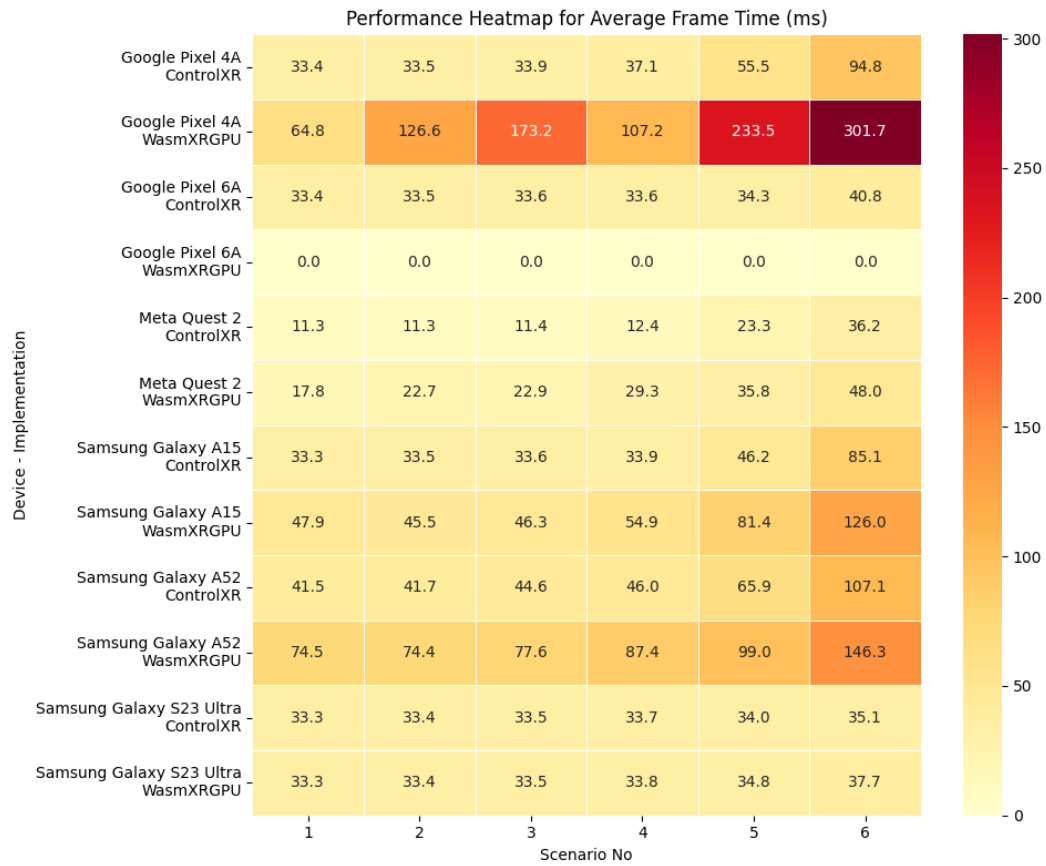


Figure 5.57: Heatmap Visualization of Average Frame Time (ms) between ControlXR and WasmXRGPU with respect to the defined scenarios in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower the color, better.

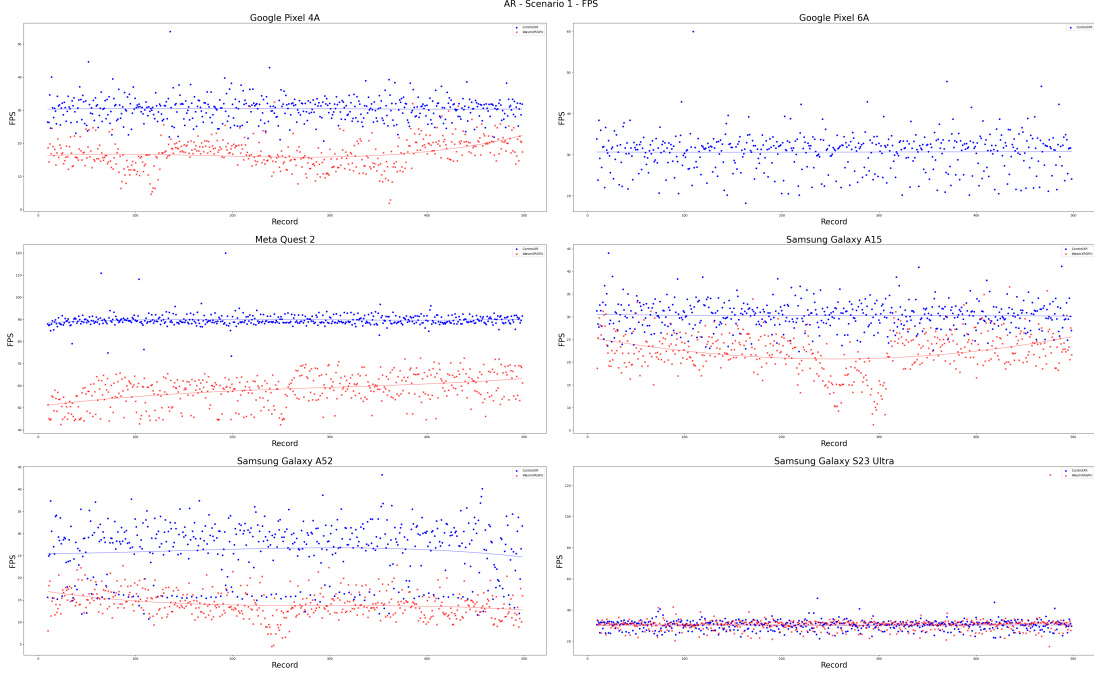


Figure 5.58: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.

across all scenarios, while Figures 5.59, 5.61, 5.63, 5.65, 5.67, 5.69 display the corresponding scatter plots for Frame Time.

Across all plots, no significant anomalies were observed beyond the expected spread and fluctuation of metric values. However, in Scenario 4, the Meta Quest 2 exhibits a distinct distribution pattern in the WasmXRGPU implementation, where the metric data is spread across two separate ranges.

For this reason, violin plot visualizations for the AR mode are not presented in this section. However, they have been included in the Appendix A.4 for those interested.

Session Load Time

In this section, we analyze the time required to load the very first frame by examining the frame time of the first recorded entry. To compare the initial frame time with subsequent frames, the first 10 recorded frame times are visualized using bar plots. These visualizations are shown in Figure 5.70 through Figure 5.75, corresponding to each scenario. No consistent or distinguishable patterns were observed across different devices and scenarios. However, in most cases, ControlXR exhibited better initial performance than WasmXRGPU, similar to the results observed in VR mode.

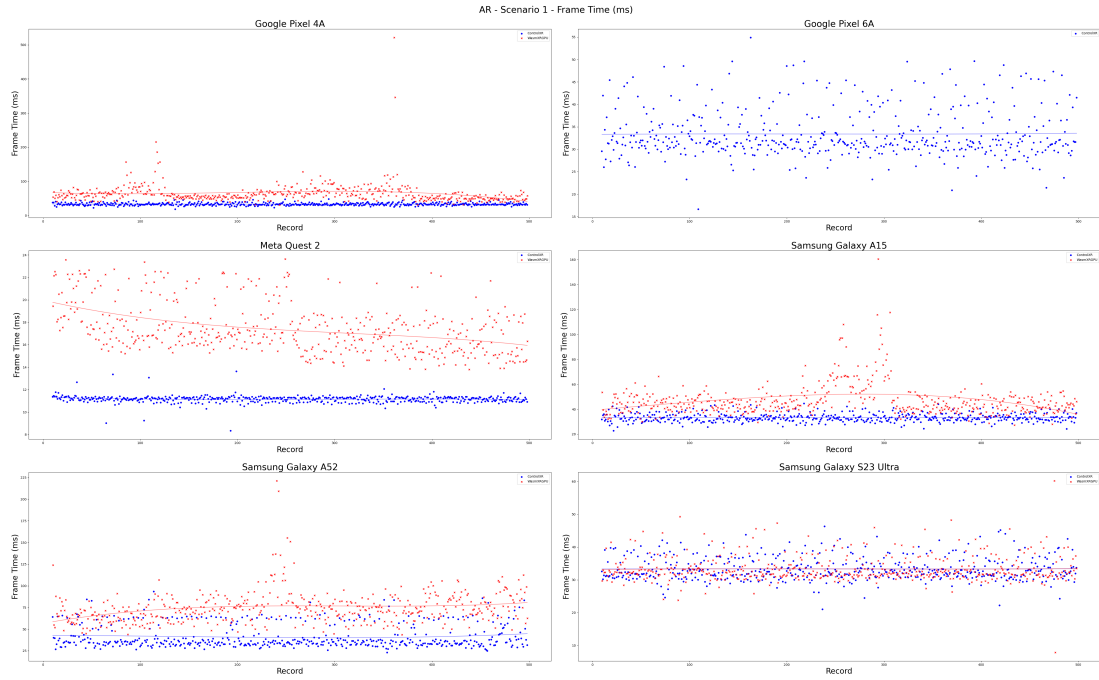


Figure 5.59: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

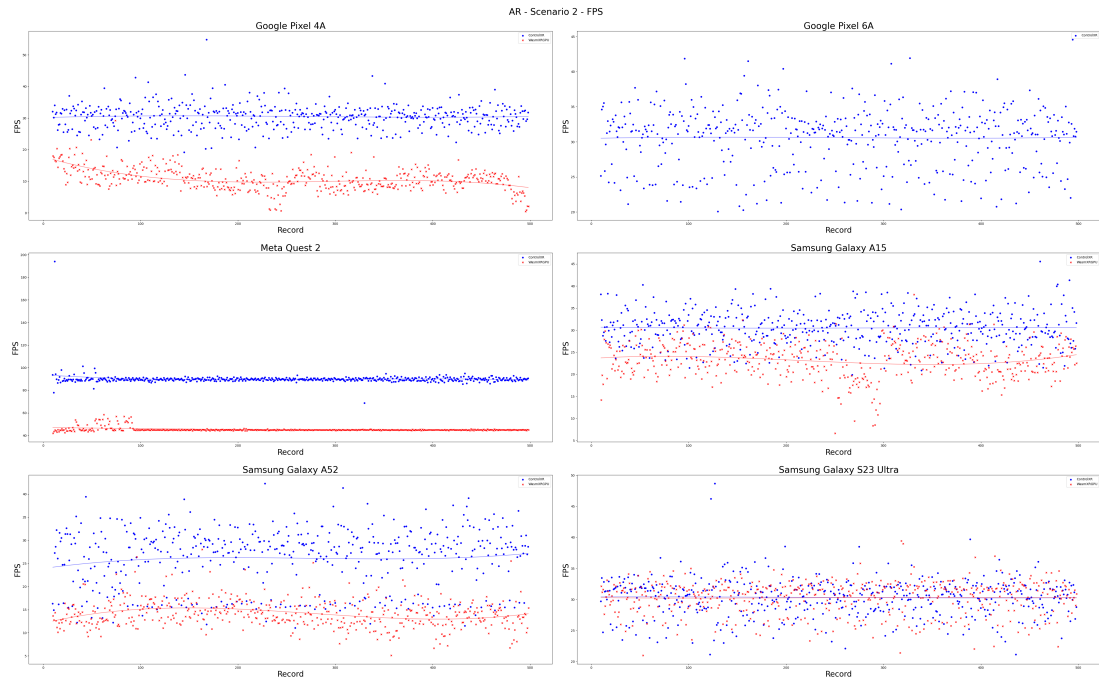


Figure 5.60: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.

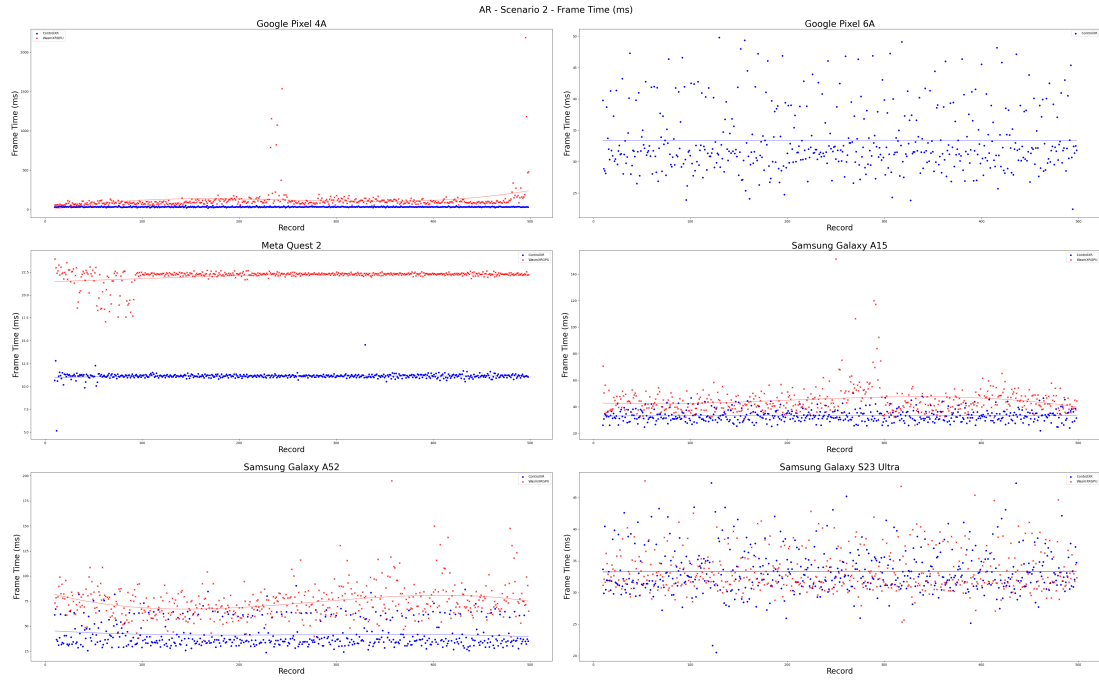


Figure 5.61: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

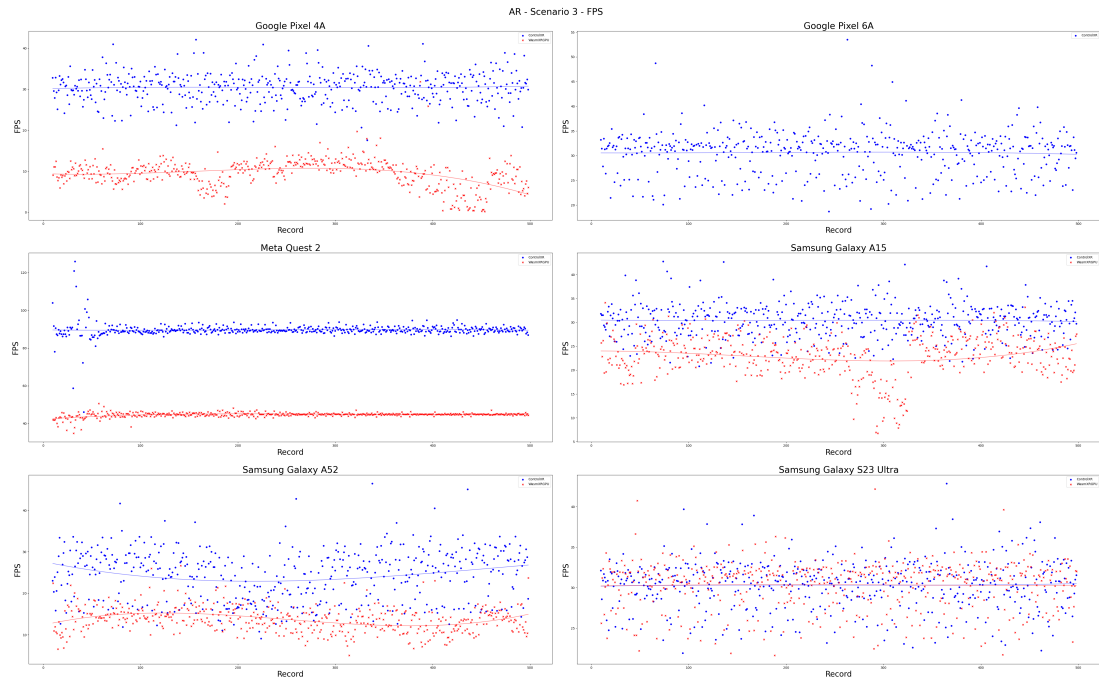


Figure 5.62: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.

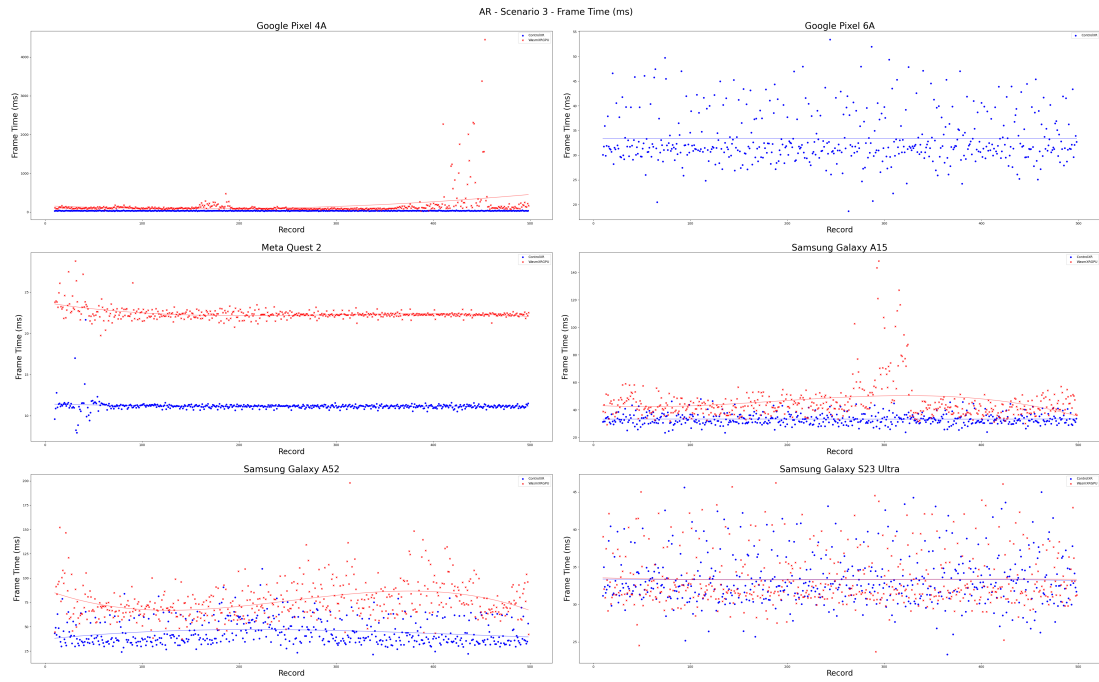


Figure 5.63: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

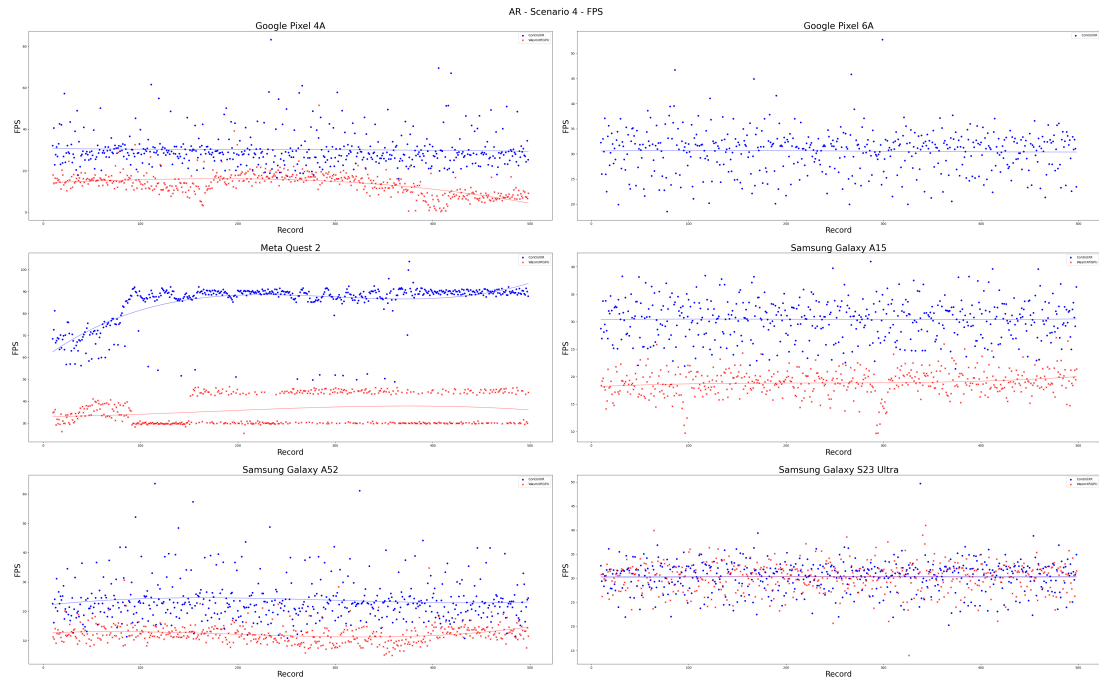


Figure 5.64: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.

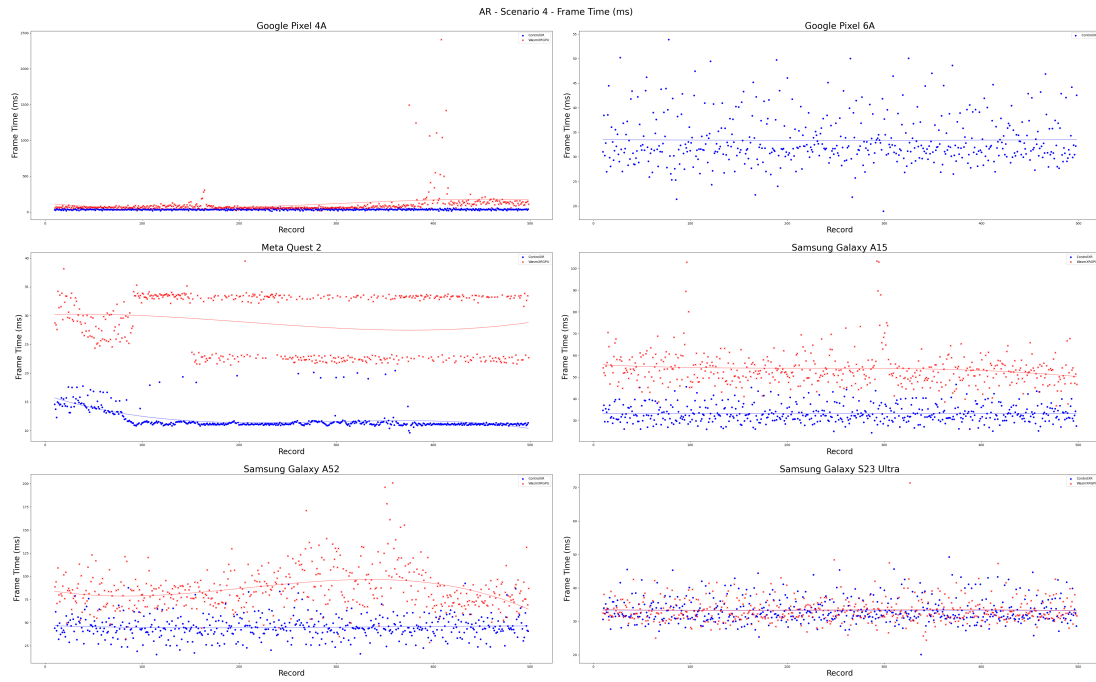


Figure 5.65: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

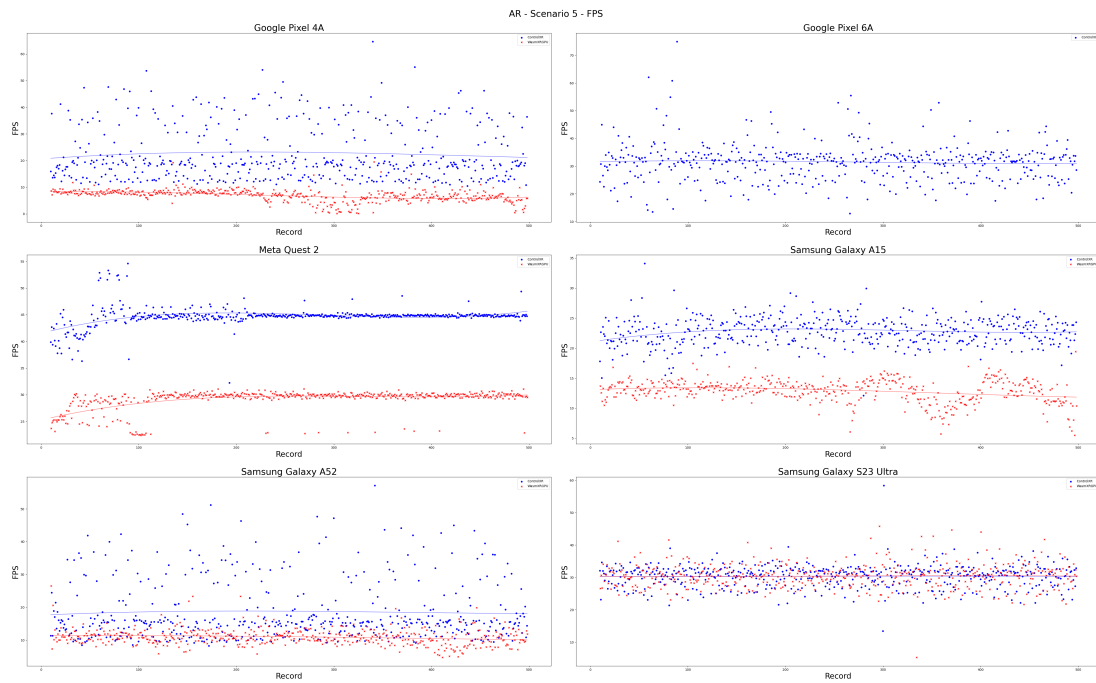


Figure 5.66: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.

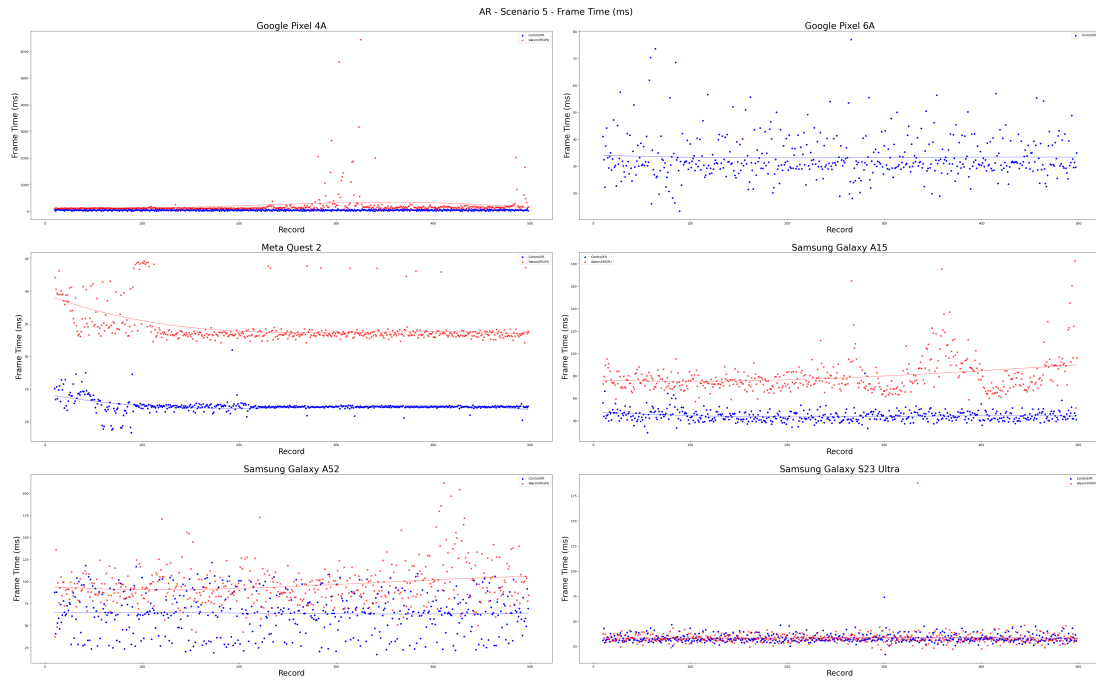


Figure 5.67: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

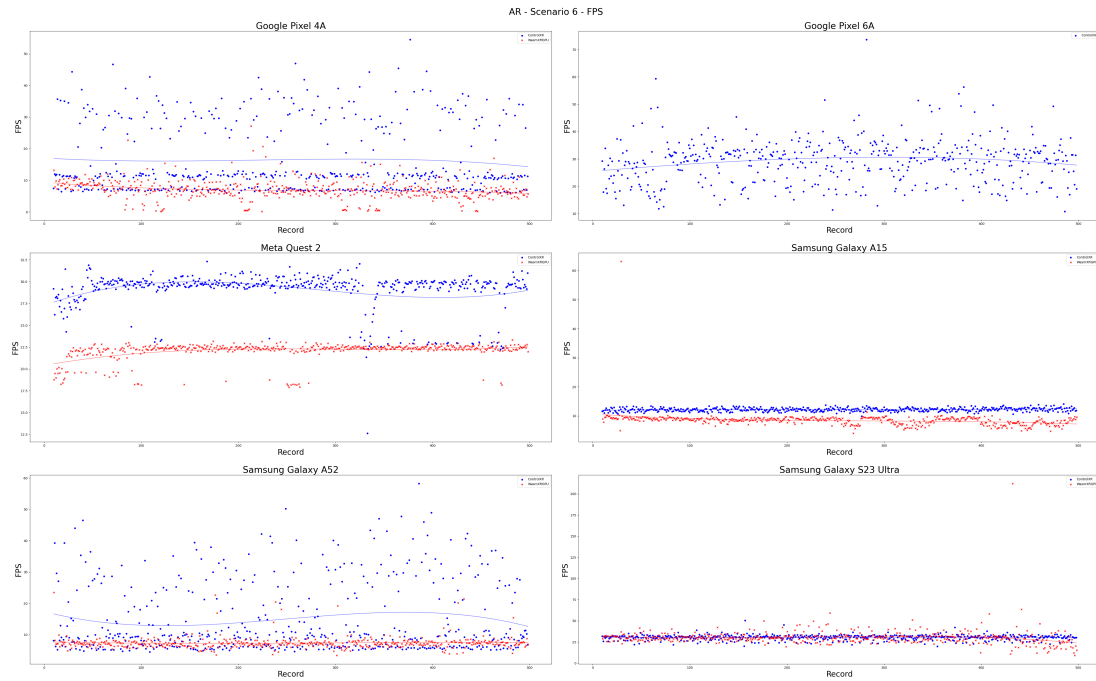


Figure 5.68: Distribution of FPS between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Higher is better.

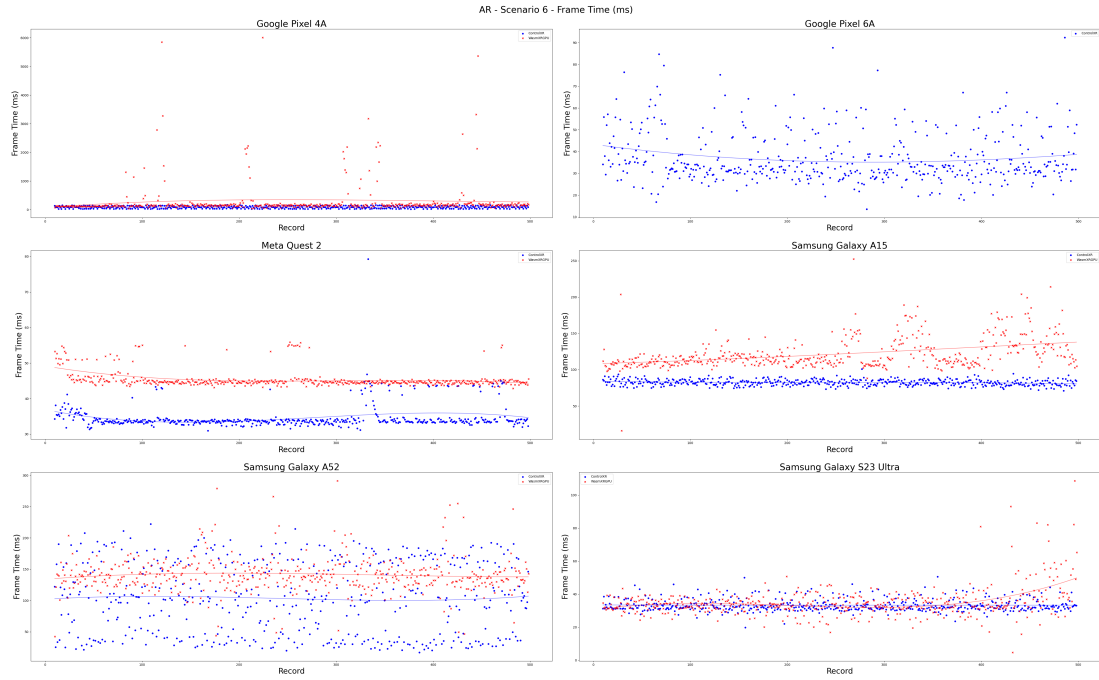


Figure 5.69: Distribution of Frame Time (ms) between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

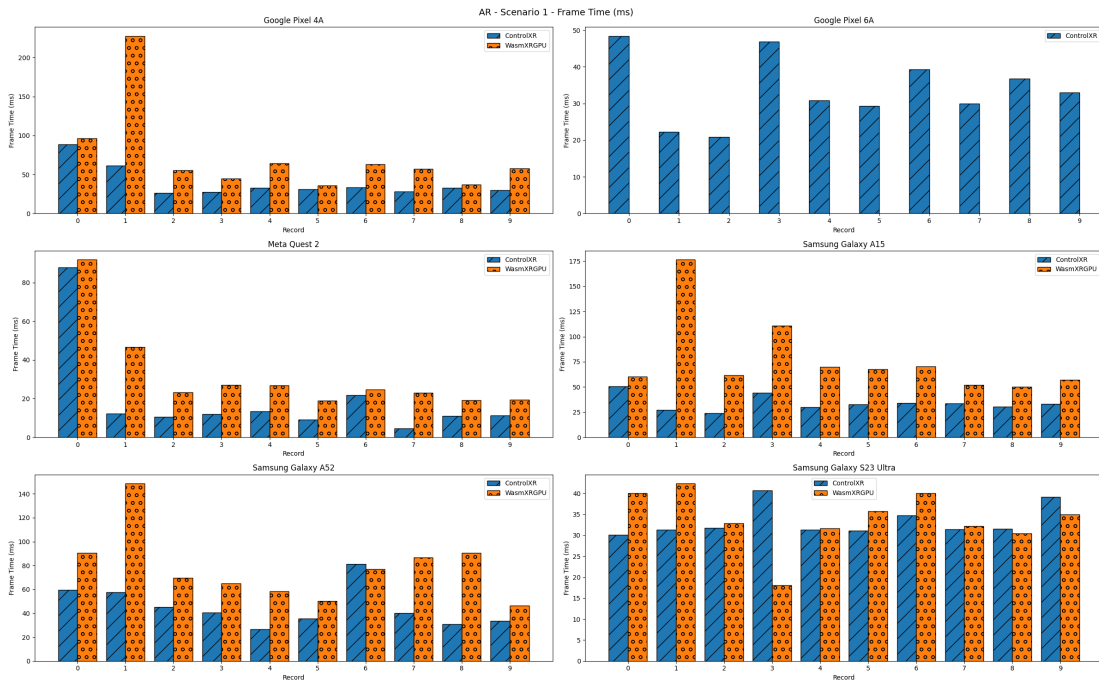


Figure 5.70: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 1, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

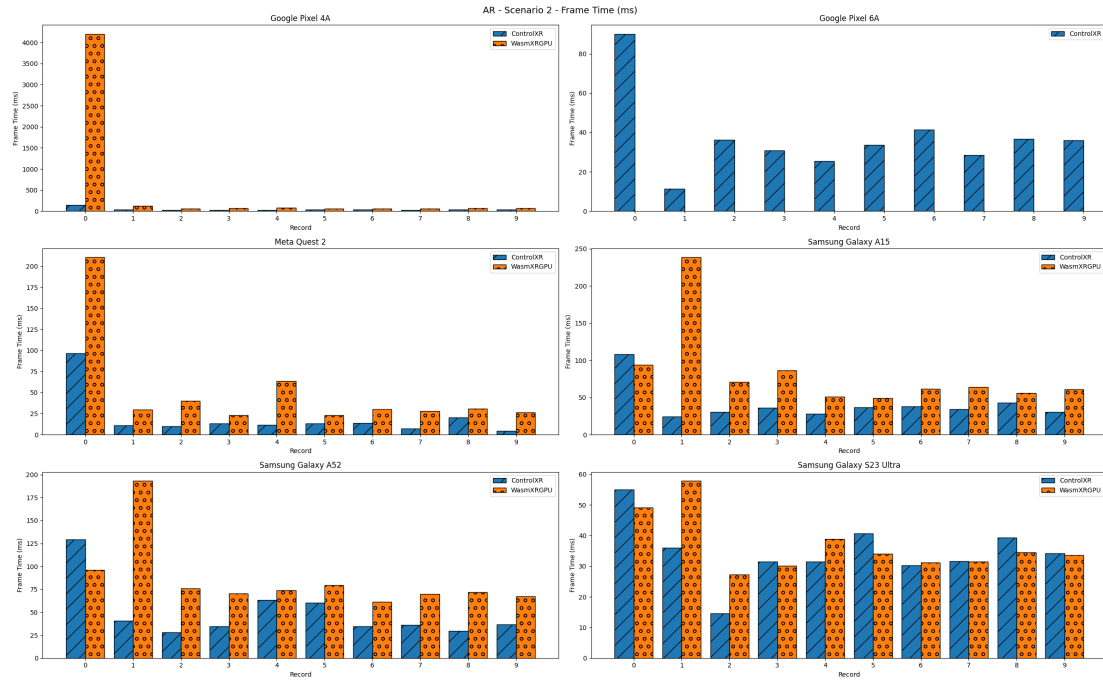


Figure 5.71: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 2, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

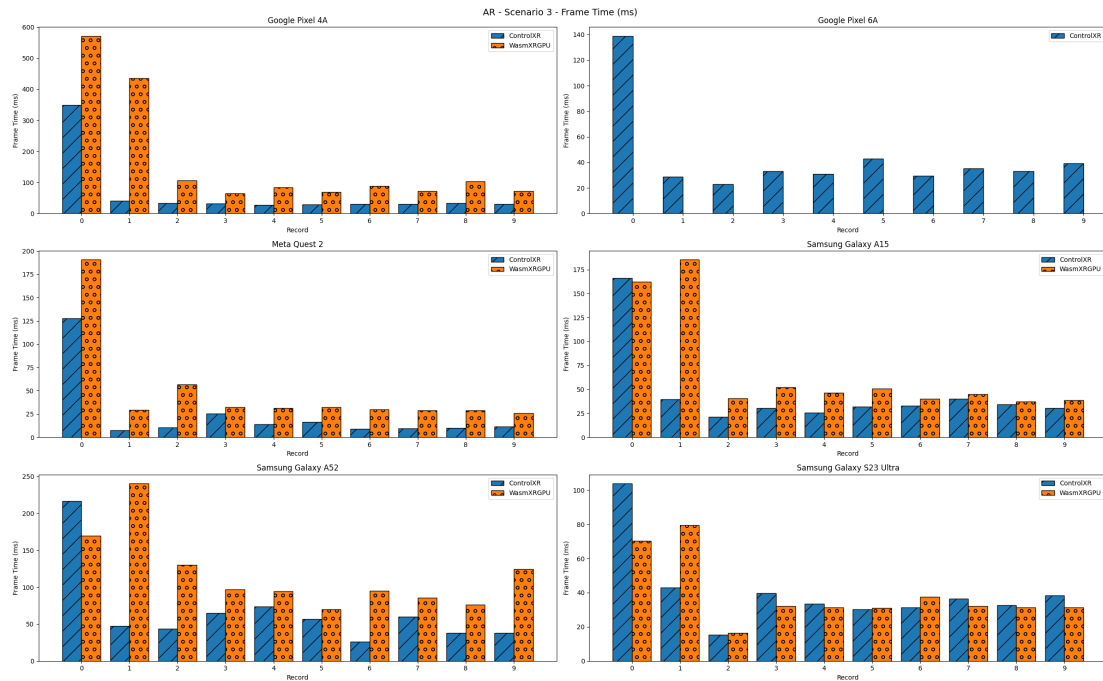


Figure 5.72: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 3, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

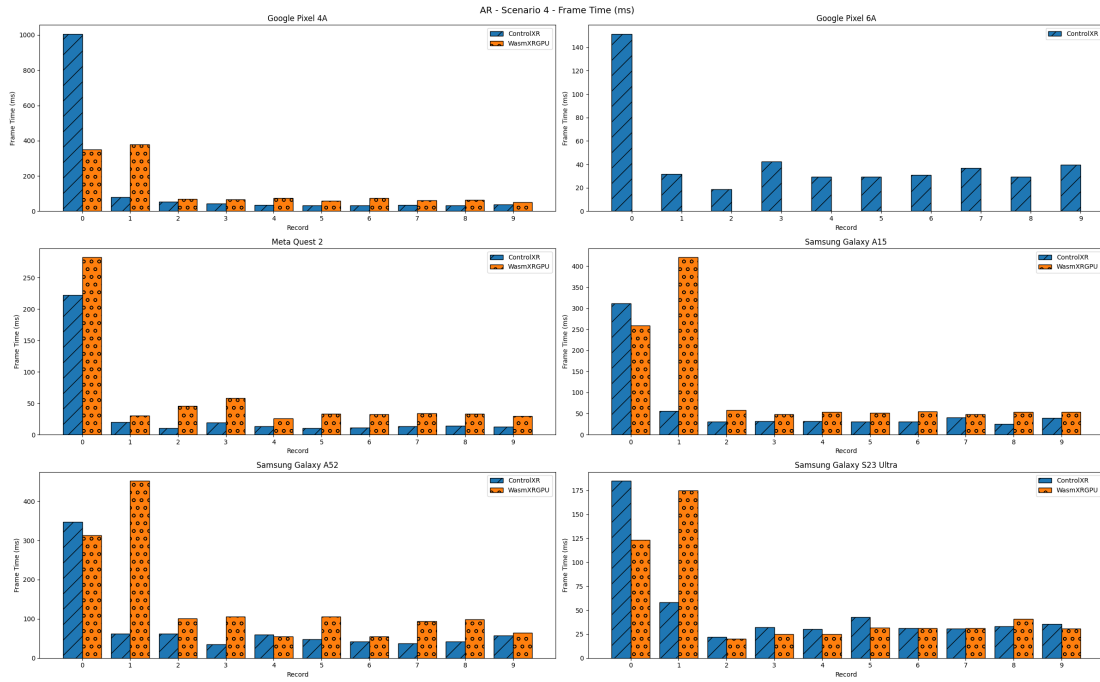


Figure 5.73: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 4, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

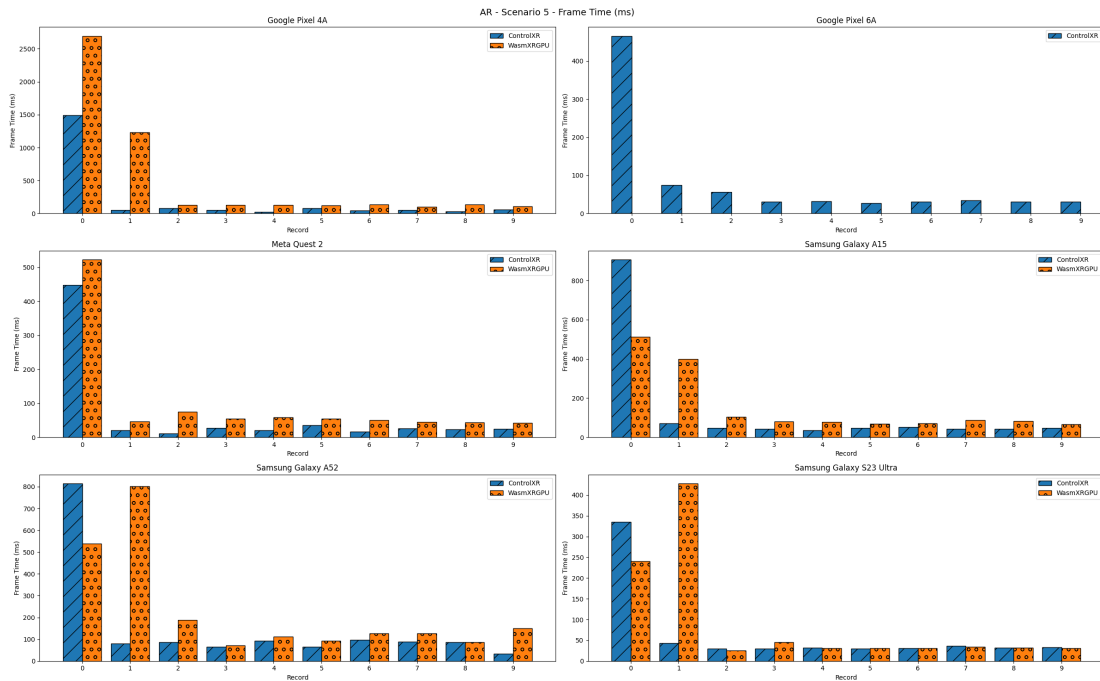


Figure 5.74: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 5, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

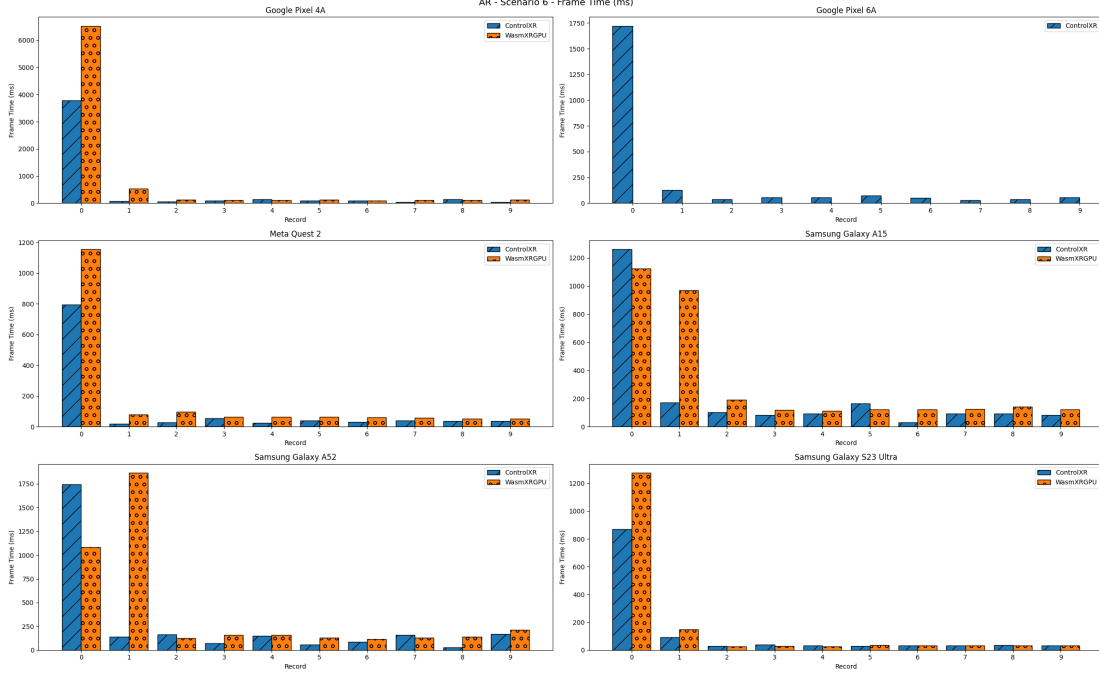


Figure 5.75: Distribution of the first 10 Frame Time (ms) values for ControlXR and WasmXRGPU in Scenario 6, across the devices listed in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

5.3 ControlXR and WasmXRGPU Experiment – Qualitative Analysis

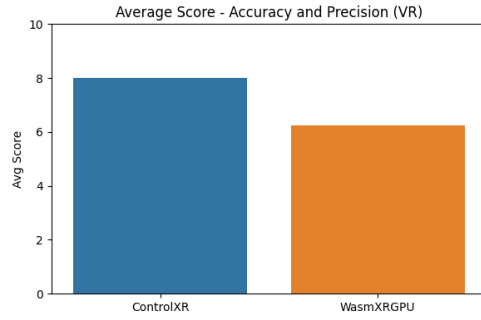
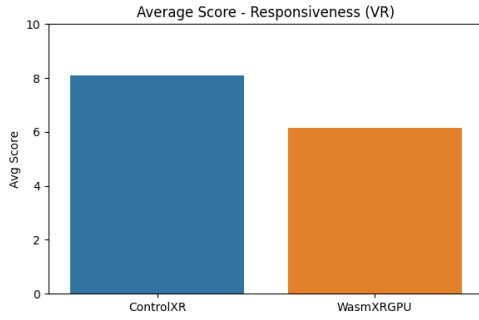
As discussed in Section 3.3.2, hosted implementations of both ControlXR and WasmXRGPU were made available to end users in order to gather qualitative insights into their perceived performance and user experience. Participants interacted with the systems and provided feedback based on a structured set of questions outlined earlier in the document.

5.3.1 Virtual Reality (VR)

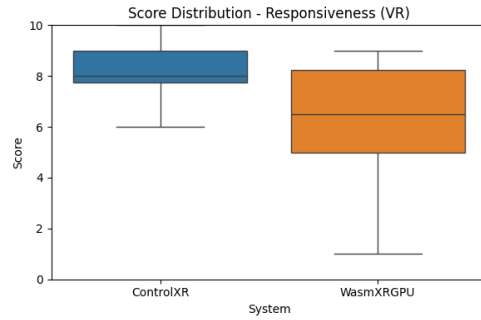
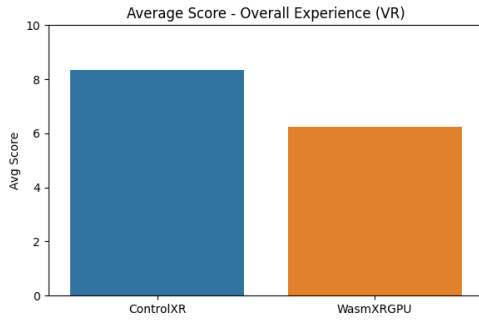
The results of the user feedback are presented in Figures 5.76 and 5.77. Across all evaluated metrics, ControlXR consistently received higher average scores compared to WasmXRGPU. Additionally, the score distribution for WasmXRGPU exhibited greater variability, indicating a broader range of user experiences and perceptions. Their final judgment on overall preference is tabulated in 5.5.

5.3.2 Augmented Reality (AR)

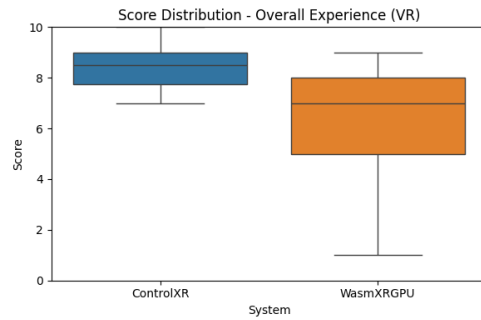
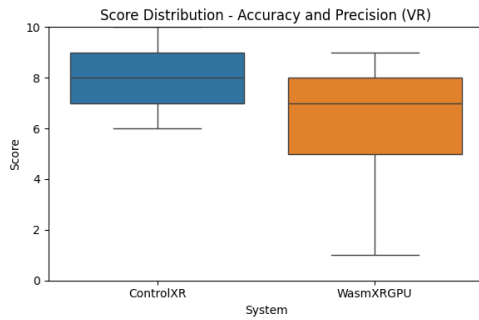
The results of the user feedback are presented in Figures 5.78 and 5.79. Similar to VR, across all evaluated metrics, ControlXR consistently received higher av-



(a) Average Scores for Responsiveness. (b) Average Scores for Accuracy and Precision. Higher is better.



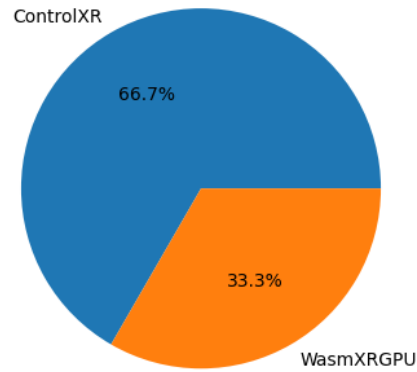
(c) Average Scores for Overall Experience. (d) Score Distribution for Responsiveness. Higher is better.



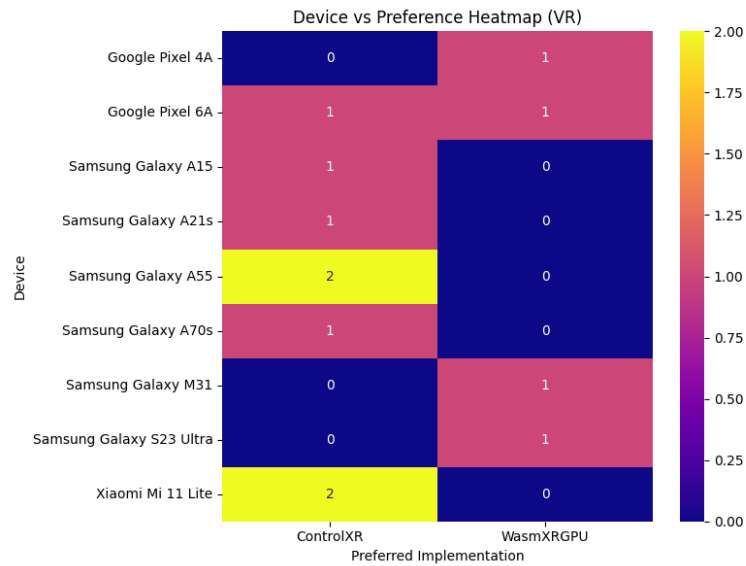
(e) Score Distribution for Accuracy and Precision. (f) Score Distribution for Overall Experience. Higher is better.

Figure 5.76: Qualitative evaluation of ControlXR and WasmXRGPU in VR mode across multiple criteria.

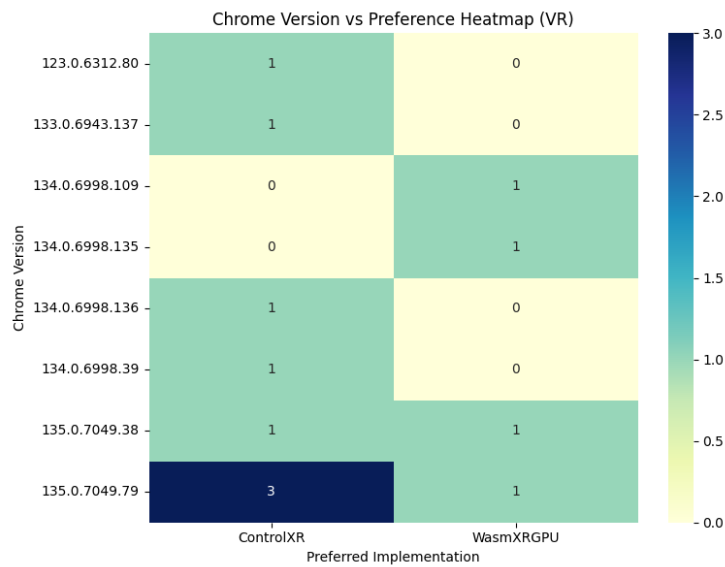
Preferred Implementation (VR)



(a) Preference Pie Chart



(b) Device vs Preference Heatmap



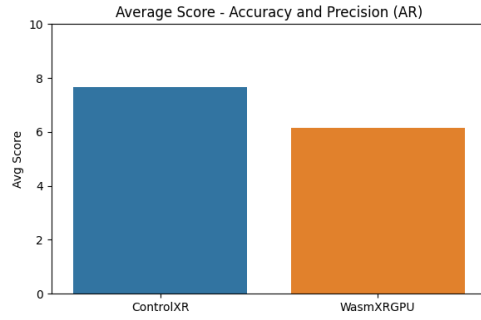
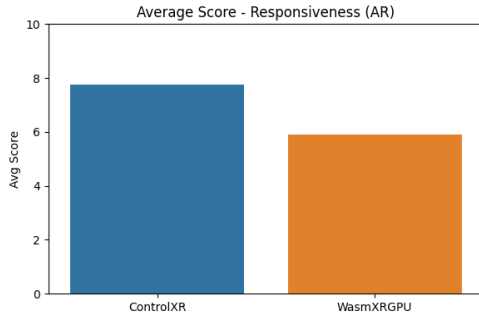
(c) Chrome Version vs Preference Heatmap

Figure 5.77: Participant preference of ControlXR and WasmXRGPU in VR mode.

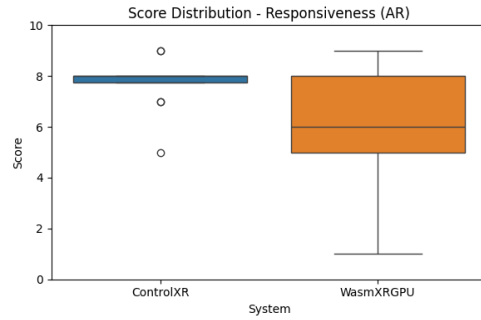
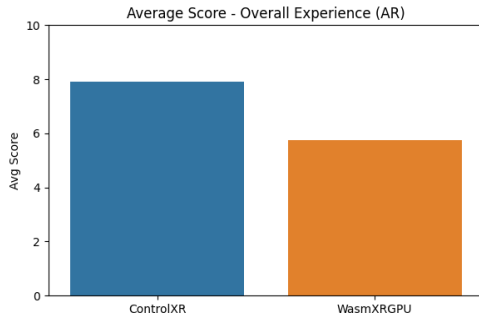
Device	Preference	Justification
Google Pixel 6A	WasmXRGPU	Overall better performance
Xiaomi Mi 11 Lite	ControlXR	Good
Samsung Galaxy A15	ControlXR	WasmXRGPU not working
Samsung Galaxy M31	WasmXRGPU	Better than the other
Xiaomi Mi 11 Lite	ControlXR	Smoothnes
Samsung Galaxy S23 Ultra	WasmXRGPU	It felt smoother and precise
Google Pixel 6A	ControlXR	Average smoothness of controlxr than wasm also in high complexity scenarios
Samsung Galaxy A21s	ControlXR	I preffered ControlXR because it felt easier to work with and overall a better experience.
Samsung Galaxy A70s	ControlXR	Smoothness
Samsung Galaxy A55	ControlXR	WasmXRGPU didn't work on my device.
Samsung Galaxy A55	ControlXR	ControlXR feels better.
Google Pixel 4A	WasmXRGPU	ControlXR's objects looked more incomplete and the perofmrance was bit laggy compared to WasmXRGPU

Table 5.5: Qualitative evaluation partipant preference and justification - Virtual Reality (VR)

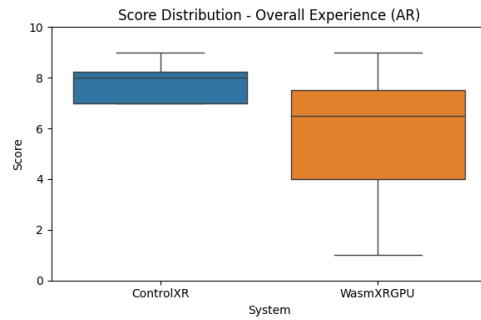
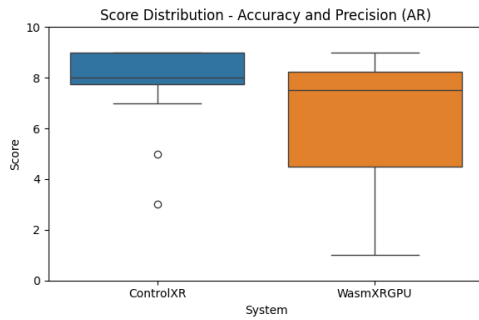
erage scores compared to WasmXRGPU. Additionally, the score distribution for WasmXRGPU exhibited greater variability, indicating a broader range of user experiences and perceptions. Their final judgment on overall preference is tabulated in 5.6.



(a) Average Scores for Responsiveness. (b) Average Scores for Accuracy and Precision. Higher is better.



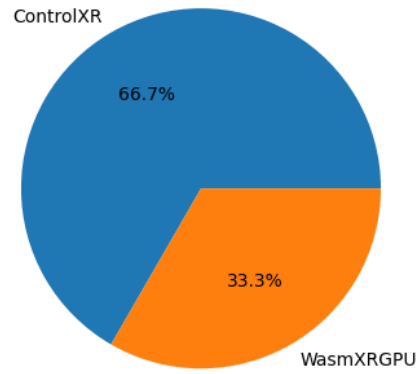
(c) Average Scores for Overall Experience. (d) Score Distribution for Responsiveness. Higher is better.



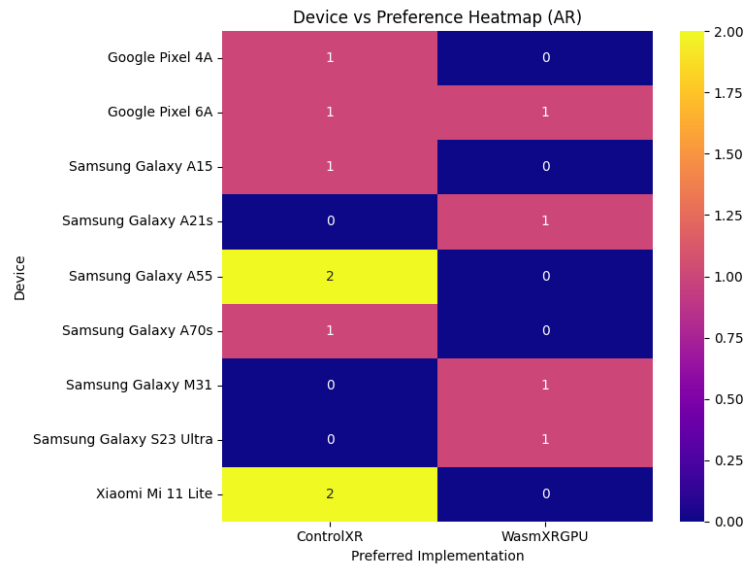
(e) Score Distribution for Accuracy and Precision. (f) Score Distribution for Overall Experience. Higher is better.

Figure 5.78: Qualitative evaluation of ControlXR and WasmXRGPU in VR mode across multiple criteria.

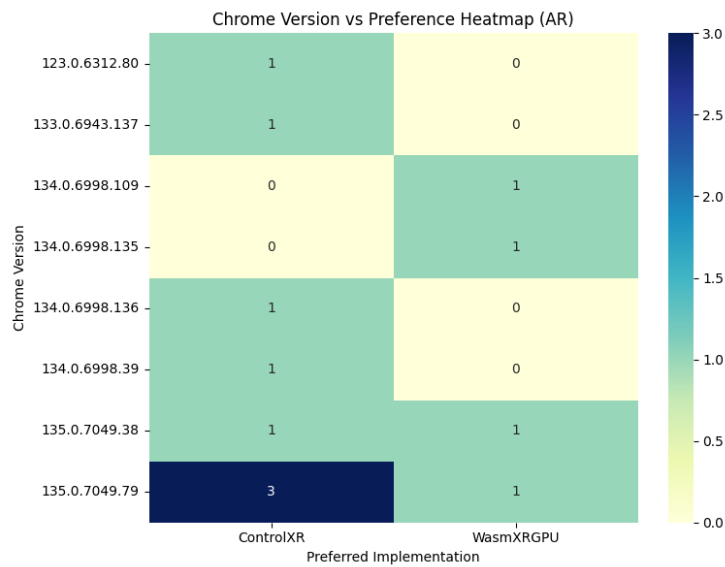
Preferred Implementation (AR)



(a) Preference Pie Chart



(b) Device vs Preference Heatmap



(c) Chrome Version vs Preference Heatmap

Figure 5.79: Participant preference of ControlXR and WasmXRGPU in AR mode.

Device	Preference	Justification
Google Pixel 6A	WasmXRGPU	Better overall performance
Xiaomi Mi 11 Lite	ControlXR	Second link felt laggy
Samsung Galaxy A15	ControlXR	WasmXRGPU is not working
Samsung Galaxy M31	WasmXRGPU	Worked better than the other
Xiaomi Mi 11 Lite	ControlXR	Main reason is responsiveness
Samsung Galaxy S23 Ultra	WasmXRGPU	It had better performance and felt smoother.
Google Pixel 6A	ControlXR	Performance and stability was good in controlXR than WasmXRGPU
Samsung Galaxy A21s	WasmXRGPU	I prefer WasmXRGPU because it offered smoother performance and felt more responsive during use.
Samsung Galaxy A70s	ControlXR	Smoothness
Samsung Galaxy A55	ControlXR	WasmXRGPU didn't work on my device.
Samsung Galaxy A55	ControlXR	ControlXR was better overall.
Google Pixel 4A	ControlXR	ControlXR had more stable performance compared to WasmXRGPU

Table 5.6: Qualitative evaluation participant preference and justification - Augmented Reality (AR)

Chapter 6

Critical Evaluation of Results

In this chapter, the results presented in Chapter 5 are critically evaluated to determine whether the use of WebGPU and WebAssembly in the context of web-based XR applications leads to performance improvements.

6.1 WebGPU-WebGL Interoperability Experiment

The purpose of this experiment was to determine whether using two graphics APIs results in significant overhead. This approach is necessary because the WebXR Device API does not yet have official, built-in support for WebGPU.

6.1.1 Using a Single High Poly 3D Model

The results presented in Section 5.1.1 indicate that there are no significant overheads between the three groups: WebGL-only, WebGPU-only, and WebGPU-WebGL interoperability.

Regarding the Average WebGL Time and Average WebGPU Time measures shown in Figures 5.3 and 5.4, it is evident that in the WebGPU-WebGL interoperability method, both WebGPU and WebGL times are recorded. The results suggest that the performance is on par with the respective standalone applications for each API. This outcome might appear counterintuitive at first glance, as WebGPU is responsible for the majority of the computations, while WebGL mainly handles the *blitFramebuffer* operation.

This seeming anomaly could be attributed to command stalling between WebGL and WebGPU due to the asynchronous nature of their execution. However, due to the asynchronous behavior of both APIs, pinpointing the exact execution order or delays remains a challenge.

One-way ANOVA Test

Although visually there are no significant overheads, statistical validation can provide further assurance. To evaluate whether any significant differences exist, a one-way ANOVA test was conducted.

The performance comparison for the various rendering methods was performed using this statistical test, analyzing metrics such as Average Frame Time (ms), Average JavaScript Time (ms), and Average FPS. The results of the ANOVA test are summarized in Table 6.1. The F-statistics for the metrics are 0.01, 0.25, and 0.01, respectively, and the corresponding p-values are 0.99, 0.78, and 0.99. These high p-values suggest that there are no statistically significant differences in performance across the evaluated rendering methods for these metrics.

It is important to note that the Average WebGL Time (ms) and Average WebGPU Time (ms) were excluded from the analysis. This is because these metrics are specific to the respective methods: the WebGL-only method does not produce WebGPU metrics, and the WebGPU-only method does not generate WebGL metrics. As a result, these metrics were not considered in the ANOVA test.

Based on the ANOVA results, we can conclude that there is no significant difference in performance between the rendering methods with respect to Average Frame Time, JavaScript Time, and FPS. This suggests that the different rendering methods perform similarly across these metrics.

Metric	F-statistic	p-value
Average Frame Time (ms)	0.0107	0.9894
Average JS Time (ms)	0.2496	0.7831
Average FPS	0.0121	0.9880

Table 6.1: ANOVA results for performance metrics when multiple instances of single high poly 3D model are used.

6.1.2 Using Two 3D Models

Metric	F-statistic	p-value
Average Frame Time (ms)	2.9357	0.0840
Average JS Time (ms)	0.0040	0.9960
Average FPS	0.9274	0.4171

Table 6.2: ANOVA results for performance metrics when multiple instances of two 3D models are used.

The results presented in Section 5.1.2 indicate that there are no significant overheads between the three groups: WebGL-only, WebGPU-only, and WebGPU-

WebGL interoperability. However, further analysis reveals notable differences between the WebGL-only method and the other two methods. Specifically, WebGPU’s performance appears to be significantly reduced when compared to WebGL, likely due to the high number of draw calls involved. On the other hand, the WebGPU-WebGL interoperability approach maintains performance similar to that of the WebGPU-only method, suggesting that the overhead introduced by interoperability does not contribute to significant performance degradation.

Moreover, the results shown in Table 6.2 indicate that the p-value for Average Frame Time (ms) is very close to the 0.05 threshold, hinting at a potential performance difference. However, since the p-value does not exceed this threshold, we cannot confidently conclude that the differences are statistically significant. This outcome may be attributed to the fact that both the WebGPU-only method and the WebGPU-WebGL interoperability method produced similar results, which likely balanced out any potential differences in performance.

Despite the lack of clear statistical significance, the data clearly suggest that an increase in the number of draw calls and CPU-GPU communication negatively impacts WebGPU performance. This aligns with the findings from Bi et al. (2024), which proposed methods to merge draw calls and reduce communication between the CPU and GPU. However, it is important to note that their study did not explicitly address the relationship between scene complexity and the number of draw calls, which may further explain the observed performance degradation in more complex scenes.

6.2 ControlXR and WasmXRGPU Experiment - Quantitative Evaluation

With the conclusion that the interoperability approach does not introduce significant overheads, the study was able to proceed with its main experiment. The remainder of this section is divided into two parts, discussing the VR and AR modes of the experiment.

6.2.1 Virtual Reality (VR)

CPU Time (ms) Figure 5.12 visualizes the average CPU times. As observed, when the scene complexity increases, ControlXR’s CPU time rises more significantly compared to WasmXRGPU. Two main reasons contribute to this behavior:

1. Unlike ControlXR, the majority of WasmXRGPU’s codebase is written in C++. As demonstrated in previous studies, WebAssembly generally outperforms JavaScript in terms of runtime performance. Similar results are observed here as well, although the performance improvement becomes apparent only in scenarios with higher complexity. This leads to the next point.
2. As mentioned in Section 4.4, the use of compute shaders in WebGPU allows many matrix multiplications and inversions to be offloaded from the CPU to the GPU. This is not the case with ControlXR, where such operations are consistently handled on the CPU. As scene complexity increases, so does the number of 3D objects—and consequently, the number of matrix operations. Therefore, it is evident that offloading these tasks to the GPU frees up CPU resources. However, it is also observed that WasmXRGPU performs worse than ControlXR in low-complexity scenarios in terms of CPU time. This is because the number of 3D objects—and thus the matrix operations—is minimal, making the overhead of CPU-GPU communication more prominent and impactful.

GPU Time (ms) Figure 5.13 shows the average GPU times. As previously discussed, a majority of matrix multiplications in WasmXRGPU are performed on the GPU. As a result, the GPU time in WasmXRGPU is generally higher than that of ControlXR across most devices.

However, an interesting exception is observed on the Meta Quest 2, where ControlXR exhibits significantly higher GPU time compared to WasmXRGPU. This observation suggests that dedicated XR devices like the Meta Quest 2 may be better optimized for modern graphics API architectures such as WebGPU.

Framerate Per Second (FPS) Figure 5.14 presents the average FPS measured across all tested devices under the defined scenarios. As previously noted, some devices exhibit anomalous behavior where the FPS increases despite the rising complexity of the rendering scenarios. To investigate potential causes, scatter plots and violin plots were utilized, as shown in Figures 5.20, 5.22, 5.24, 5.26, 5.28, 5.30 and Figures 5.32, 5.34, 5.36, 5.38, 5.40, 5.42 respectively.

In scenarios with lower complexity, both ControlXR and WasmXRGPU demonstrate stable performance, with ControlXR generally achieving higher FPS. However, beginning with scenario 3, ControlXR shows increased instability on most devices. As previously discussed, the FPS values often fall

into two distinct ranges: one with unusually high values that raise the overall average, and another with significantly lower values that bring the average down. The combined effect of these two ranges results in an overall average FPS that is higher than that of WasmXRGPU.

From a user experience perspective, the higher FPS values in ControlXR are not perceived, likely due to frame drops occurring to synchronize with the next rendered frame. This phenomenon appears to be the root cause of the observed anomalies. Additionally, the more stable FPS values of WasmXRGPU on some devices tend to be higher than, or comparable to, the lower range of ControlXR's fluctuating values—indicating that users may actually experience smoother performance with WasmXRGPU. An exception to this trend is observed on the Google Pixel 6A, where WasmXRGPU exhibits a wider distribution of FPS values, suggesting less consistent performance.

Frame Time (ms) Figure 5.15 presents the average frame time measured across all tested devices under the defined scenarios. Since FPS is calculated based on frame time, the anomalies discussed earlier are also reflected here. However, they are not very apparent in Figure 5.15. Therefore, Figures 5.21, 5.23, 5.25, 5.27, 5.29, 5.31 and Figures 5.33, 5.35, 5.37, 5.39, 5.41, 5.43 visualize the scatter plots and the violin plots, respectively, to better identify these anomalies.

Similar to the FPS observations, starting from scenario 3, ControlXR shows unstable frame time values across most devices. In contrast, WasmXRGPU's frame times tend to remain within a consistent range that falls below the higher end of ControlXR's fluctuating range. Since lower frame time indicates better performance, this suggests that WasmXRGPU generally performs better under this reasoning. However, this trend does not hold true for all devices. For example, on the Meta Quest 2, such behavior is not observed, and in that case, WasmXRGPU does not outperform ControlXR.

Session Load Time The recording of performance metrics starts when the user requests the API to start a session using some sort of an event. This is because there's a limitation/restriction imposed by WebXR Device API to prevent a XR session being created without the user consent.

Figures 5.44 through 5.49 visualize the first 10 frame time values of each scenario rendered by all the experimented devices. As previously observed, nearly every graph shows that ControlXR exhibits better performance (i.e., lower frame time values) compared to WasmXRGPU. A possible explanation

for this behavior may lie in the underlying WebAssembly implementation. As discussed in Section 4.4, WasmXRGPU stores WebXR data in the WebAssembly memory heap. This introduces additional memory allocations and WebAssembly-specific configurations that are not present in ControlXR.

Furthermore, WebGPU entails more complex configurations than WebGL. Although the ‘GLRenderer’ and ‘GPURenderer’, shown in Figures 4.4 and 4.5, are created during the webpage load, the actual creation of scenarios and the merging of draw calls involve additional dynamic steps that are triggered when the user initiates a request event.

Even though WasmXRGPU exhibits better load time in the first recorded value of some scenarios, the frame time remains significantly higher compared to ControlXR in the second recorded value.

6.2.2 Augmented Reality (AR)

CPU Time (ms) Figure 5.50 visualizes the average CPU times. As observed, when scene complexity increases, ControlXR’s CPU time rises more significantly compared to WasmXRGPU on all devices except the Google Pixel 4A. As discussed in the VR mode section, the reasoning behind WasmXRGPU outperforming ControlXR is likely the same, since—despite the difference in modes—the rendered scenarios remain identical.

GPU Time (ms) Figure 5.51 shows the average GPU times. As previously discussed, a majority of matrix multiplications in WasmXRGPU are performed on the GPU. As a result, the GPU time in WasmXRGPU is generally higher than that of ControlXR across most devices. However, Samsung Galaxy A15 and Samsung Galaxy S23 Ultra show that GPU time in ControlXR is very similar to that of WasmXRGPU. While it is difficult to pinpoint the exact cause of this behavior from the data alone, it may suggest that these specific devices handle GPU workloads more efficiently, or that the overhead of CPU-based computation in ControlXR is offset by other optimizations or hardware characteristics. Further investigation into GPU utilization patterns or driver-level optimizations on these devices would be needed to draw more concrete conclusions.

Another interesting observation lies within Meta Quest 2. In VR mode, Meta Quest 2 showed the opposite behavior, where WasmXRGPU had better performance in GPU time compared to ControlXR. However, this trend does not continue in AR mode. In fact, WasmXRGPU’s GPU time is significantly higher than that of ControlXR, indicating that GPU compute for matrix op-

erations is indeed taking place. This contradicts the earlier assumption that Meta Quest 2 is inherently more optimized for modern graphics API architectures. A possible explanation could be that this dedicated XR device handles different XR modes—VR and AR—with different system-level optimizations or hardware scheduling strategies. This mode-specific behavior may be affecting how rendering workloads are distributed between the CPU and GPU.

Framerate Per Second (FPS) Figure 5.52 presents the average FPS measured across all tested devices under the defined scenarios. Unlike in VR mode, no visible anomalies are observed in the graph. However, to investigate potential outliers and anomalies, scatter plots were utilized, as shown in Figures 5.58, 5.60, 5.62, 5.64, 5.66, 5.68.

Even after analyzing the scatter plots, no anomalies were detected, indicating that in AR mode, WasmXRGPU does not outperform ControlXR in terms of FPS. However, an interesting observation is that the Samsung Galaxy S23 Ultra maintains a consistent FPS range across all scenarios. Compared to other mobile devices, the Samsung Galaxy S23 Ultra is equipped with superior hardware specifications, which likely contributes to its strong performance, consistently achieving 30 FPS. The scatter plots for this device reveal no distinct ranges that could artificially elevate the average FPS. Throughout the experiment, both ControlXR and WasmXRGPU consistently maintain similar and close FPS values.

Samsung Galaxy A52 and Google Pixel 4A exhibit some outliers in scenarios with higher complexities that may contribute to the average metric value. However, these outliers do not form a distinct distribution range. The presence of these outliers may be due to the WebGL time measuring query being stalled for a period of time.

As for Meta Quest 2, there are some varying distributions in scenario 4 for WasmXRGPU. However, the ranges of WasmXRGPU do not surpass the distribution range of ControlXR, which does not alter the final conclusion regarding their average performance.

Frame Time (ms) Figure 5.53 presents the average frame time measured across all tested devices under the defined scenarios. Figures 5.59, 5.61, 5.63, 5.65, 5.67, 5.69 visualize the scatter plots to identify any potential anomalies.

Similar to the FPS observations, no significant anomalies are found in this metric, aside from some outliers observed for Samsung Galaxy A52 and

Google Pixel 4A, as mentioned earlier. A similar case is observed for Meta Quest 2 as well.

Session Load Time Figures 5.70 through 5.75 visualize the first 10 frame time values of each scenario rendered by all the experimented devices. As previously observed, nearly every graph indicates that ControlXR demonstrates better performance (i.e., lower frame time values) than WasmXRGPU. This aligns with the same observation and reasoning discussed in VR mode.

6.3 ControlXR and WasmXRGPU Experiment - Qualitative Evaluation

Despite some cases where WasmXRGPU outperforms ControlXR in quantitative metrics, the qualitative feedback from participants indicates a clear preference for ControlXR overall. The score distributions, as shown in Figures 5.76, 5.77, 5.78 and 5.79, reveal greater variation in user experiences with WasmXRGPU across both VR and AR modes, suggesting that the effectiveness of the implementation depends heavily on the specific device and participant. While a few participants did rate WasmXRGPU as the better implementation, most found that even if it performed better in one mode, ControlXR was favored in the other. This points to the potential influence of different runtime backends used in each mode.

Tables 5.5 and 5.6 present the participants' justifications for their preferences. Many highlighted factors such as responsiveness and smoothness of performance in their chosen implementation. However, some participants reported that WasmXRGPU did not function at all, which may suggest compatibility issues related to hardware or browser versions.

6.4 Virtual Reality vs. Augmented Reality

The study aimed to use the same scenarios, as defined in Table 5.3, across both VR and AR modes. However, notable differences were observed in the results produced by these two modes. Furthermore, qualitative feedback revealed that some participants preferred different implementations depending on the mode, suggesting discrepancies in the runtime backends of VR and AR. This section critically examines potential factors contributing to these differences.

First, in the case of VR, the application is required to render two separate views—one for each eye—creating a fully immersive experience. Additionally, there is no camera overlay involved in the rendering pipeline.

In contrast, AR does not render two views for each eye. Instead, the virtual content is rendered on a separate layer, particularly when the WebXR Device API is used. Unlike VR, AR includes a camera overlay that displays the real-world environment captured by the device’s camera.

Furthermore, as previously noted, Table 4.1 illustrates that VR and AR modes rely on different backend runtimes or SDKs. This discrepancy introduces variation at the system level that cannot be modified by developers or researchers attempting to make direct comparisons between the two modes.

In both implementations—ControlXR and WasmXRGPU—Figure 5.14 shows that FPS gradually decreases as scene complexity increases in the context of VR. In contrast, Figure 5.52 demonstrates that FPS in AR mode appears to be capped at certain values depending on the device. On mobile devices, for example, the maximum FPS tends to be close to 30, whereas in VR, the maximum FPS varies across devices. This behavior is likely influenced by the presence of the camera overlay in AR. Since the rendered frames must remain synchronized with the live camera feed, exceeding the camera’s refresh rate could result in visual artifacts or latency, degrading the user experience. Therefore, it is plausible that the WebXR API enforces a cap on the maximum FPS in AR mode.

A notable exception to this trend is the Meta Quest 2, which consistently maintains an FPS of approximately 90 in both VR and AR modes. The device’s specifications indicate that its display supports a 90Hz refresh rate, suggesting that its camera subsystem is also capable of matching that framerate, thereby avoiding any need for artificial capping.

Lastly, the observation that devices such as the Google Pixel 4A and Google Pixel 6A perform significantly worse in AR mode compared to VR, while other devices perform better in AR mode, further supports the conclusion that hardware infrastructure plays a substantial role in determining overall performance.

6.5 The Impact of WebGPU–WebGL Interoperability

Although a dedicated experiment concluded that this interoperability method does not introduce significant overhead in general contexts, it was not conducted specifically within the domain of web-based XR. The reason for this limitation lies in the lack of official WebGPU support in the WebXR Device API, making it currently infeasible to implement a fully WebGPU-based XR experiment group for comparison.

This raises the question of whether the chosen interoperability approach may introduce performance overheads or limitations specifically in the context of Web-based XR. While the WebXR Device API specification (*WebXR Device API* 2024) primarily defines how the API interfaces with XR devices and does not explicitly mandate or restrict the use of specific graphics APIs beyond their role in rendering virtual content layers, there may still be GPU-level operations, internal configurations, or hidden resource contention that contribute to performance bottlenecks or inefficiencies. These potential overlaps and constraints remain speculative without deeper low-level access, but their existence cannot be entirely ruled out.

6.6 Findings and Contributions

After critically evaluating the results presented in Chapter 5, the key findings and contributions of the evaluation can be summarized as follows:

1. Due to the lack of native or official WebGPU support in the WebXR Device API, a hybrid rendering approach was introduced in which WebGPU and WebGL are used together through an interoperability method.
2. This WebGPU–WebGL interoperability method enabled the use of WebGPU’s advanced GPU capabilities within immersive web experiences, while relying on WebGL for XR session context management.
3. Performance comparisons between the interoperability method, standalone WebGPU, and standalone WebGL revealed no statistically significant overhead from using the interoperability method, even across scenes of varying geometric complexity.
4. The significance of these performance observations was confirmed using a one-way ANOVA test, ensuring statistical robustness.
5. One of the critical findings in the interoperability evaluation was the impact of draw call count on overall rendering performance. As the number of draw calls increased, WebGPU time increased disproportionately in WebGPU-based approaches.
6. To mitigate this variable, a draw call merging strategy, inspired by Bi et al. (2024), was applied consistently across both ControlXR and WasmXRGPU implementations to isolate the effect of rendering backends.
7. The evaluation was performed in both VR and AR contexts, across multiple Android-based mobile devices, to assess the portability and performance consistency of the proposed approach.

8. In both contexts, WasmXRGPU demonstrated significantly improved CPU performance in complex scenes, as intensive matrix computations were offloaded to the GPU via compute shaders and parallel execution.
9. The observed increase in GPU time for WasmXRGPU was an expected result of this offloading strategy, which highlights the GPU’s effective utilization rather than inefficiency.
10. While FPS and frame time varied depending on the device and context, the WasmXRGPU approach exhibited more consistent frame pacing and lower variability, especially in VR contexts.
11. These variations are likely influenced by differences in XR runtime backends used by the devices (e.g., ARCore for AR and Google Cardboard for VR), as shown in Table 4.1. Such backend differences lead to inconsistent scheduling and rendering behavior across platforms.
12. It was also observed that the ControlXR implementation, which relies solely on WebGL, had less stable performance in complex scenes, especially in the VR context in some devices.
13. Overall, WasmXRGPU did not consistently outperform ControlXR across all metrics. This may be attributed to the additional overhead introduced by the WebGPU–WebGL interoperability method. However, this cannot be definitively concluded, as a purely WebGPU-based implementation of WasmXRGPU could not be developed due to the current lack of native WebGPU support in the WebXR Device API. Therefore, a direct comparison between a WebGPU-only and the hybrid WebGPU–WebGL implementation in WebXR context remains infeasible.

Chapter 7

Conclusion

This study set out to investigate the performance of web-based Extended Reality (XR) applications developed using emerging web technologies—namely WebAssembly and WebGPU. The motivation stemmed from the increasing need to push the boundaries of what web applications can achieve in immersive environments, and to explore whether browser-based XR can compete with or complement native XR solutions. This aim was achieved by designing and implementing a novel framework that integrates these technologies, followed by a systematic performance evaluation across multiple devices and scenarios.

The study identified several integration challenges, including performance considerations, differences in graphics API abstraction between WebGL and WebGPU, and the lack of official WebGPU support in the WebXR Device API. These challenges were addressed through an iterative development process that led to the discovery of practical solutions, ultimately resulting in a performant prototype.

During the prototype design phase, multiple integration strategies were explored. One viable approach was to create a wrapper around the WebXR Device API for access from C++ code. Another more practical method involved handling the WebXR Device API via JavaScript and allocating the data in the WebAssembly memory heap, making it directly accessible from C++. This approach minimized communication overhead between JavaScript and WebAssembly contexts.

Following the JavaScript-managed data allocation strategy, the absence of official WebGPU support in the WebXR Device API necessitated an interoperability layer between WebGPU and WebGL. To realize the prototype, high-performance third-party libraries were used for fast 3D model loading. WebGPU compute shaders were leveraged for efficient matrix operations. Compiler-level optimizations were applied to the WebAssembly module, and merged draw calls were introduced to

minimize CPU-GPU communication overhead.

A fully functioning web-based XR application was implemented, supporting both VR and AR modes. The system was designed with modular components to promote flexibility and extensibility.

Performance was evaluated through frame rate analysis, CPU/GPU usage metrics, and comparative benchmarking with ControlXR (a WebGL-based baseline implementation). The results, presented in both tabular and graphical form, highlighted the impact of scene complexity, device hardware, and runtime environments on overall system behavior.

As stated in Section 6.6, performance analysis revealed that the WebAssembly + WebGPU implementation consistently outperformed the WebGL-based version in terms of CPU time, owing to the offloading of intensive matrix operations to the GPU. Although the average frame time lagged behind the traditional WebGL implementation, it remained more stable under increasing scene complexity in the context of VR. However, performance differed markedly across devices—especially between mobile devices and standalone XR headsets, highlighting the critical role of hardware and runtime environments.

7.1 Overall Outcome

The study successfully fulfilled its aim of investigating the performance of web-based XR applications powered by WebAssembly and WebGPU. A functional prototype was developed and evaluated across multiple scenarios, offering insight into the strengths and limitations of the combined use of these emerging technologies in immersive web applications.

The evaluation demonstrated notable improvements in specific metrics, particularly in CPU time, attributed to the offloading of computational tasks to the GPU via WebGPU. However, the WebAssembly and WebGPU-based implementation did not consistently outperform the traditional WebGL-based approach across all performance aspects.

As discussed earlier, the performance limitations observed in the study may stem from the interoperability method used to integrate WebGPU with the WebXR Device API, which necessitated the use of WebGL as a bridge. While this hybrid approach enabled the integration of advanced graphics capabilities, it may have introduced additional overhead affecting overall performance. However, conducting experiments under this hybrid approach also enabled the evaluation of the relationship between scene complexity and draw call count. This, in turn, provided an

opportunity to validate the effectiveness of draw call merging, as proposed by Bi et al. (2024), in reducing CPU-GPU communication overhead. Thus, despite its limitations, the hybrid setup not only served its immediate purpose but also facilitated deeper insights into performance optimization strategies relevant to future WebGPU-based XR systems.

7.2 Limitations

While the study provides meaningful insights into the performance of web-based XR applications utilizing WebAssembly and WebGPU, several limitations must be acknowledged:

- **Lack of Official WebGPU Support in WebXR:** One of the primary limitations was the absence of official WebGPU integration within the WebXR Device API. As a result, the prototype had to rely on WebGL as an intermediary for XR rendering, potentially introducing overhead and impacting the overall performance of the application.
- **Merging Draw Calls:** This technique was employed to enhance WebGPU performance and reduce CPU-GPU communication overhead. However, it becomes less effective in scenes with numerous objects that frequently update or manipulate vertex data directly (as opposed to simple object transformations). In such cases, the merged buffer must be updated often, diminishing the benefits of this approach. Additionally, this study utilized a single large buffer to store all vertex data and issued a unified draw call. While effective in some scenarios, this method may not be universally applicable due to WebGPU's limitations on maximum buffer size (*"WebGPU Specification - Limits"* 2025), which vary based on the device's hardware specifications. Therefore, special care must be taken to handle such constraints and avoid runtime errors.
- **Interoperability Complexity:** The approach depended on complex interoperability between JavaScript, WebAssembly, and GPU APIs. Although functional, the communication overhead and resource management between these layers might not represent the most efficient solution in future implementations once native integration of WebGPU into WebXR is possible.
- **Limited Device Compatibility and Runtime Support:** The experiments were conducted on a limited range of devices and browsers, some of which had incomplete or inconsistent support for emerging standards like WebGPU. This constrained the ability to generalize the performance ob-

servations across a broader array of platforms and user environments. For example, due to the closed ecosystem of iOS and Apple, iPhones do not support WebXR Device API, and Mozilla Firefox’s WebXR Device API and WebGPU are not supported by default.

- **Evaluation Constraints in AR Context:** Camera frame rate limitations and API constraints in AR environments prevented a pure separation between hardware-imposed caps (e.g., 30 FPS on mobile devices) and the evaluation of rendering capabilities. This limitation made it difficult to assess the true potential of WebGPU-based rendering, especially in the context of AR performance.
- **Prototype-Specific Optimizations:** The study’s optimizations and design decisions were specifically tailored to the prototype system. While effective within this context, they may not apply universally to all XR applications or frameworks. One such case is the handling of matrix operations. Offloading matrix calculations to the GPU is effective for complex scenes, but for simpler scenarios, this may not provide the same benefits. Furthermore, WebGPU’s high parallelism introduces challenges when working with parent-child relationships in scene trees. For instance, calculating the child’s model matrix relies on the parent’s model matrix, but the parallel execution of GPU threads does not guarantee the necessary order, potentially resulting in race conditions and incorrect results.

7.3 Future Directions

While the current study provides a foundation for implementing and evaluating web-based XR applications using WebAssembly and WebGPU, several avenues remain open for future exploration and improvement:

- **Native WebGPU Support in WebXR:** One of the most significant improvements would come from the integration of native WebGPU support within the WebXR Device API. Future developments in the WebXR standard may allow WebGPU to be used directly for rendering, removing the need for WebGL as an intermediary and thus eliminating the associated overhead. This would enable more efficient XR rendering with better performance and lower latency.
- **Cross-Platform Compatibility:** Expanding the range of devices and browsers used in testing would allow for a broader understanding of how WebGPU-based XR applications perform across different environments. As

WebGPU support continues to evolve, it would be crucial to assess performance on a wider variety of platforms, including mobile devices, standalone XR headsets, and desktops. Ensuring cross-platform compatibility is essential for the widespread adoption of web-based XR applications.

- **Addressing Performance Bottlenecks in AR:** Future work could focus on further isolating AR performance from hardware-imposed limitations, such as camera frame rates. One potential direction would be to develop better techniques for decoupling camera performance from rendering tasks, allowing for more accurate evaluations of the rendering capabilities in AR environments.
- **Refining Optimizations for Scene Complexity:** As seen in this study, certain optimizations, such as offloading matrix operations to the GPU, are effective only in complex scenes. Future research could investigate other optimizations tailored to less complex scenes, or even dynamic optimizations that adjust based on scene complexity. Additionally, further exploration of the challenges related to parallel execution and thread synchronization in WebGPU could lead to new solutions for handling parent-child relationships in scene trees more effectively.
- **CPU Parallelism Using Pthreads or Web Workers:** Another promising direction for future work is the use of CPU parallelism through Pthreads or Web Workers, which is supported by Emscripten. It provides built-in support for parallel execution using Web Workers (MozDevNet 2023) (in the case of JavaScript) and Pthreads (for multi-threaded C/C++ code). By leveraging these technologies, it's possible to offload computationally intensive tasks to multiple threads or workers, allowing for better parallelism and more efficient CPU usage. This could lead to significant performance improvements, especially for tasks like physics simulations, AI processing, or other heavy calculations in XR applications. However, using Pthreads or Web Workers in a web-based environment introduces challenges such as managing thread synchronization and ensuring compatibility across different browsers, as not all platforms provide complete support for multi-threading in the same way. Future work could investigate how to efficiently implement and optimize these parallel processing techniques in the context of WebGPU and WebAssembly-based XR applications, potentially improving responsiveness and performance in both AR and VR scenarios.
- **Real-Time Collaborative Experiences:** With the increasing demand for multiplayer or collaborative XR experiences, future studies could explore

the potential of WebGPU and WebAssembly in enabling real-time collaborative applications. This would require addressing issues such as latency, synchronization, and data sharing between users, while ensuring smooth performance across a wide range of devices.

These directions collectively highlight the evolving landscape of web-based immersive technologies and point toward a future where WebGPU and WebAssembly may be seamlessly integrated into WebXR workflows. As browser vendors continue to advance support for these technologies, future research and development efforts will be better positioned to achieve true native-level performance on the web. By addressing the current limitations and exploring these promising avenues, subsequent studies can build upon this foundation to deliver richer, more efficient XR experiences directly through the browser.

References

8th Wall (2024). Last accessed on 2024-06-19.

URL: <https://www.8thwall.com/docs/home/intro/>

Ammann, M., Drabble, A., Ingensand, J. & Chapuis, B. (2022), ‘Maplibre-
rs: Toward portable map renderers’, *The International Archives of the
Photogrammetry, Remote Sensing and Spatial Information Sciences* **XLVIII-
4/W1-2022**, 35–42.

URL: <https://isprs-archives.copernicus.org/articles/XLVIII-4-W1-2022/35/2022/>

Asm.js Specification (2011). Last accessed on 2024-06-03.

URL: <http://asmjs.org/spec/latest/>

Azuma, R. T. (1997), ‘A Survey of Augmented Reality’, *Presence: Teleoperators
and Virtual Environments* **6**(4), 355–385.

URL: <https://doi.org/10.1162/pres.1997.6.4.355>

Bi, W., Ma, Y., Han, Y., Chen, Y., Tian, D. & Du, J. (2024), Fusionrender:
Harnessing webgpu’s power for enhanced graphics performance on web browsers,
in ‘Proceedings of the ACM on Web Conference 2024’, WWW ’24, Association
for Computing Machinery, New York, NY, USA, p. 2890–2901.

URL: <https://doi.org/10.1145/3589334.3645395>

Bi, W., Ma, Y., Tian, D., Yang, Q., Zhang, M. & Jing, X. (2023), Demystifying
mobile extended reality in web browsers: How far can we go?, *in* ‘Proceedings
of the ACM Web Conference 2023’, WWW ’23, Association for Computing
Machinery, New York, NY, USA, p. 2960–2969.

URL: <https://doi.org/10.1145/3543507.3583329>

BlitFramebuffer - MDN Web Docs (2024). Last accessed on 2024-11-06.

URL: <https://developer.mozilla.org/en-US/docs/Web/API/WebGL2RenderingContext/blitFramebuffer>

Carmigniani, J. & Furht, B. (2011), *Augmented Reality: An Overview*, Springer

New York, New York, NY, pp. 3–46.

URL: https://doi.org/10.1007/978-1-4614-0064-6_1

Chickerur, S., Balannavar, S., Hongekar, P., Prerna, A. & Jituri, S. (2024), ‘Webgl vs. webgpu: A performance analysis for web 3.0’, *Procedia Computer Science* **233**, 919–928. 5th International Conference on Innovative Data Communication Technologies and Application (ICIDCA 2024).

URL: <https://www.sciencedirect.com/science/article/pii/S1877050924006410>

”Dawn, a WebGPU implementation” (2025). Last accessed on 2025-04-11.

URL: <https://dawn.googlesource.com/dawn>

DirectX by Microsoft (1995). Last accessed on 2024-06-03.

URL: <https://learn.microsoft.com/en-us/windows/win32/getting-started-with-directx-graphics>

Du, R., Turner, E., Dzitsiuk, M., Prasso, L., Duarte, I., Dourgarian, J., Afonso, J., Pascoal, J., Gladstone, J., Cruces, N., Izadi, S., Kowdle, A., Tsotsos, K. & Kim, D. (2020), Depthlab: Real-time 3d interaction with depth maps for mobile augmented reality, in ‘Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology’, UIST ’20, Association for Computing Machinery, New York, NY, USA, p. 829–843.

URL: <https://doi.org/10.1145/3379337.3415881>

Emscripten (2015). Last accessed on 2024-06-03.

URL: <https://emscripten.org/>

Erazo, C. & Demir, I. (2023), ‘Hydrocompute: An open-source web-based computational library for hydrology and environmental sciences’.

”fastgltf - Documentation” (2025). Last accessed on 2025-04-06.

URL: <https://fastgltf.readthedocs.io/latest/>

Ferrão, J., Dias, P., Santos, B. S. & Oliveira, M. (2023), ‘Environment-aware rendering and interaction in web-based augmented reality’, *Journal of Imaging* **9**(3).

URL: <https://www.mdpi.com/2313-433X/9/3/63>

Hamzaturrazak, M., Jonemaro, E. M. A. & Pinandito, A. (2023), Performance analysis of 3d rendering method on web-based augmented reality application using webgl and opengl shading language, in ‘Proceedings of the 8th International Conference on Sustainable Information Engineering and Technology’, SIET ’23, Association for Computing Machinery, New York, NY, USA, p. 637–643.

URL: <https://doi.org/10.1145/3626641.3626949>

- Kharroubi, A., Billen, R. & Poux, F. (2020), ‘Marker-less mobile augmented reality application for massive 3d point clouds and semantics’, *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* **XLIII-B2-2020**, 255–261.
URL: <https://isprs-archives.copernicus.org/articles/XLIII-B2-2020/255/2020/>
- Khomtchouk, B. B. (2021), ‘Webassembly enables low latency interoperable augmented and virtual reality software’.
- Kligge, M. (2024), ‘Comparison of webgl and webgpu as alternatives for implementing gpgpu computing in the browser’, *Abschlussbericht FEP 2023/2024* p. 13.
- Krupitzer, C. (2018), A framework for engineering reusable self-adaptive systems, PhD thesis.
- Lee, D., Shim, W., Lee, M., Lee, S., Jung, K.-D. & Kwon, S. (2021), ‘Performance evaluation of ground ar anchor with webxr device api’, *Applied Sciences* **11**(17).
URL: <https://www.mdpi.com/2076-3417/11/17/7877>
- Lee, K. (2012), ‘Augmented reality in education and training’, *TechTrends* **56**.
- Liu, K., Wu, N. & Han, B. (2023), Demystifying web-based mobile extended reality accelerated by webassembly, in ‘Proceedings of the 2023 ACM on Internet Measurement Conference’, IMC ’23, Association for Computing Machinery, New York, NY, USA, p. 145–153.
URL: <https://doi.org/10.1145/3618257.3624833>
- “loaders.gl - A collection of loaders modules for Geospatial and 3D visualization use cases” (2025). Last accessed on 2025-04-06.
URL: <https://loaders.gl/>
- MacIntyre, B. & Smith, T. F. (2018), Thoughts on the future of webxr and the immersive web, in ‘2018 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)’, pp. 338–342.
- Magnum Engine* (2022). Last accessed on 2024-06-03.
URL: <https://magnum.graphics/>
- McNally, K. F. & Koviland, H. (2024), ‘A web-based augmented reality system’, *EAI Endorsed Transactions on Scalable Information Systems* .
URL: <https://publications.eai.eu/index.php/sis/article/view/5481>

Mendoza-Ramírez, C. E., Tudon-Martinez, J. C., Félix-Herrán, L. C., Lozoya-Santos, J. d. J. & Vargas-Martínez, A. (2023), ‘Augmented reality: Survey’, *Applied Sciences* **13**(18).

URL: <https://www.mdpi.com/2076-3417/13/18/10491>

Metal by Apple (2014). Last accessed on 2024-06-03.

URL: <https://developer.apple.com/documentation/metal>

MozDevNet (2023), ‘Using web workers - web apis: Mdn’. Last accessed on 2024-06-03.

URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

Nam, D., Lee, D., Lee, S. & chul Kwon, S. (2019), Performance comparison of 3d file formats on a mobile web browser.

URL: <https://api.semanticscholar.org/CorpusID:235468359>

Nam, H., Lee, M. & Park, N. (2024), ‘Image processing acceleration using webgpu and webassembly’, *The Transactions of the Korea Information Processing Society* **13**(10), 574–578.

Neelakantam, S. & Pant, T. (2017), *Introduction to VR and WebVR*, Apress, Berkeley, CA, pp. 1–4.

URL: https://doi.org/10.1007/978-1-4842-2710-7_1

Odume, B. W., Okodugha, P. E. & Madu, I. (2024), ‘Leveraging webassembly and webgpu for efficient integration of ai models into web applications’. Available at SSRN: <https://ssrn.com/abstract=5078445> or <http://dx.doi.org/10.2139/ssrn.5078445>.

OffScreenCanvas - MDN Web Docs (2024). Last accessed on 2024-11-06.

URL: <https://developer.mozilla.org/en-US/docs/Web/API/OffscreenCanvas>

OpenCV (2000). Last accessed on 2024-06-03.

URL: <https://opencv.org/>

OpenGL (1997). Last accessed on 2024-06-03.

URL: <https://opengl.org/>

Qiao, X., Ren, P., Dustdar, S., Liu, L., Ma, H. & Chen, J. (2019), ‘Web ar: A promising future for mobile augmented reality—state of the art, challenges, and insights’, *Proceedings of the IEEE* **107**(4), 651–666.

Three.js (2010). Last accessed on 2024-06-03.

URL: <https://threejs.org/>

- Toasa G, R. M., Baldeón Egas, P., Saltos, M., Perreño, M. & Quevedo, W. (2019), *Performance Evaluation of WebGL and WebVR Apps in VR Environments*, pp. 564–575.
- Usta, Z. (2024), ‘Webgpu: A new graphic api for 3d webgis applications’, *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* **XLVIII-4/W9-2024**, 377–382.
URL: <https://isprs-archives.copernicus.org/articles/XLVIII-4-W9-2024/377/2024/>
- Vulkan (2016). Last accessed on 2024-06-03.
URL: <https://www.vulkan.org/>
- W3C (2024). Last accessed on 2024-11-06.
URL: <https://www.w3.org/>
- WebAssembly (2017). Last accessed on 2024-06-03.
URL: <https://webassembly.org/>
- WebGL (2011). Last accessed on 2024-06-03.
URL: <https://www.khronos.org/webgl/>
- WebGPU (2024). Last accessed on 2024-06-03.
URL: <https://www.w3.org/TR/webgpu/>
- ”WebGPU Specification - GPUComputePipeline” (2025). Last accessed on 2025-04-06.
URL: <https://www.w3.org/TR/webgpu/#gpucomputepipeline>
- ”WebGPU Specification - GPURenderBundle” (2025). Last accessed on 2025-04-06.
URL: <https://www.w3.org/TR/webgpu/#gpurenderbundle>
- ”WebGPU Specification - Limits” (2025). Last accessed on 2025-04-17.
URL: <https://www.w3.org/TR/webgpu/#limits>
- WebRTC (2018). Last accessed on 2024-06-03.
URL: <https://webrtc.org/>
- WebVR (2024). Last accessed on 2025-04-11.
URL: <https://webvr.info/>
- WebXR Device API (2024). Last accessed on 2024-06-03.
URL: <https://www.w3.org/TR/webxr/>

"WebXR Device API - Chrome Hardware Support" (2025). Last accessed on 2025-04-06.

URL: <https://immersiveweb.dev/chrome-support.html>

"WebXR/WebGPU Binding Module" (2025). Last accessed on 2025-04-11.

URL: <https://immersive-web.github.io/WebXR-WebGPU-Binding/>

Wonderland Engine (2024). Last accessed on 2024-06-14.

URL: <https://wonderlandengine.com/>

Zhao, Q. (2009), 'A survey on virtual reality', *Science in China Series F: Information Sciences* **52**(3), 348–400.

Appendix A

Appendix

A.1 Scatter Plots of Measured Metrics in VR

A.1.1 CPU Time (ms)

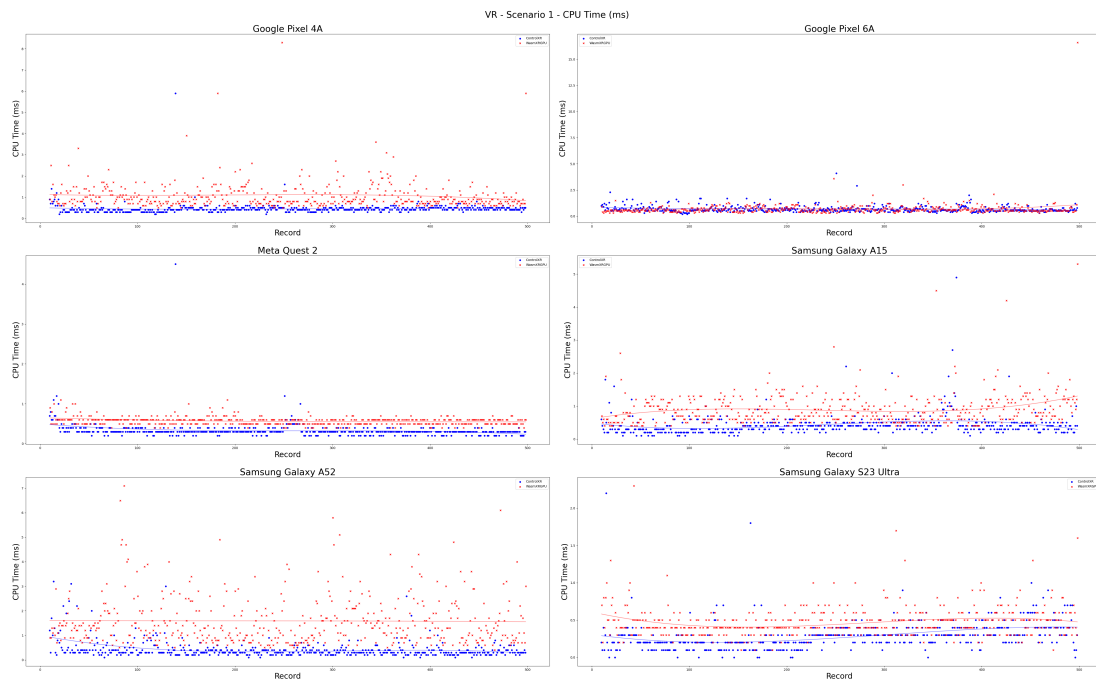


Figure A.1: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

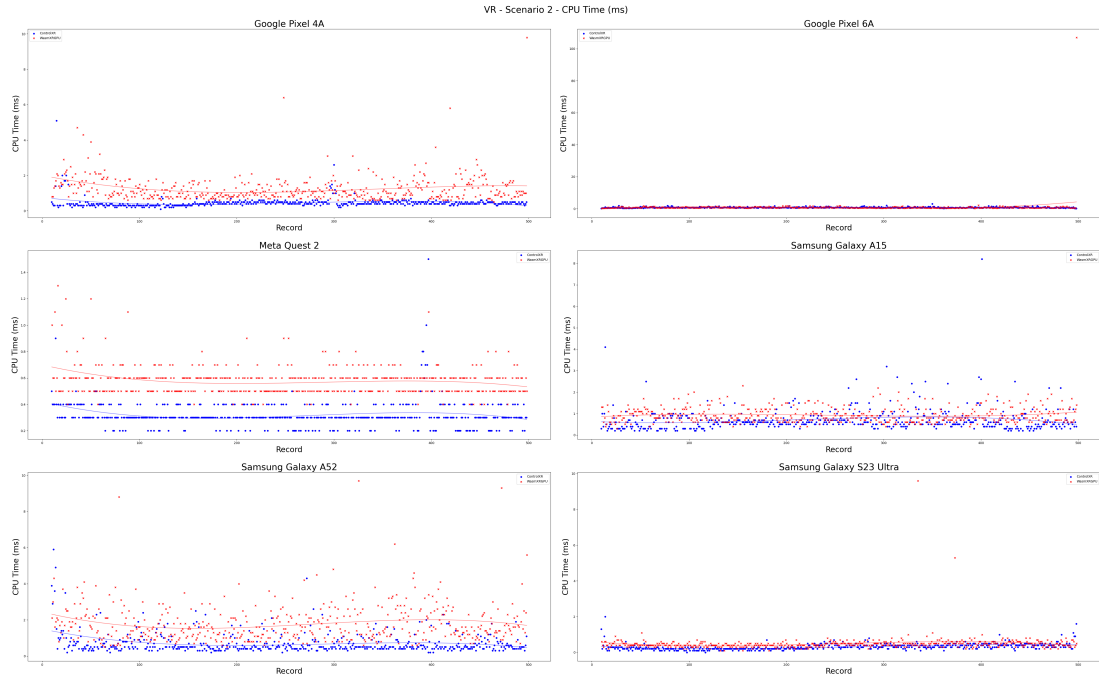


Figure A.2: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

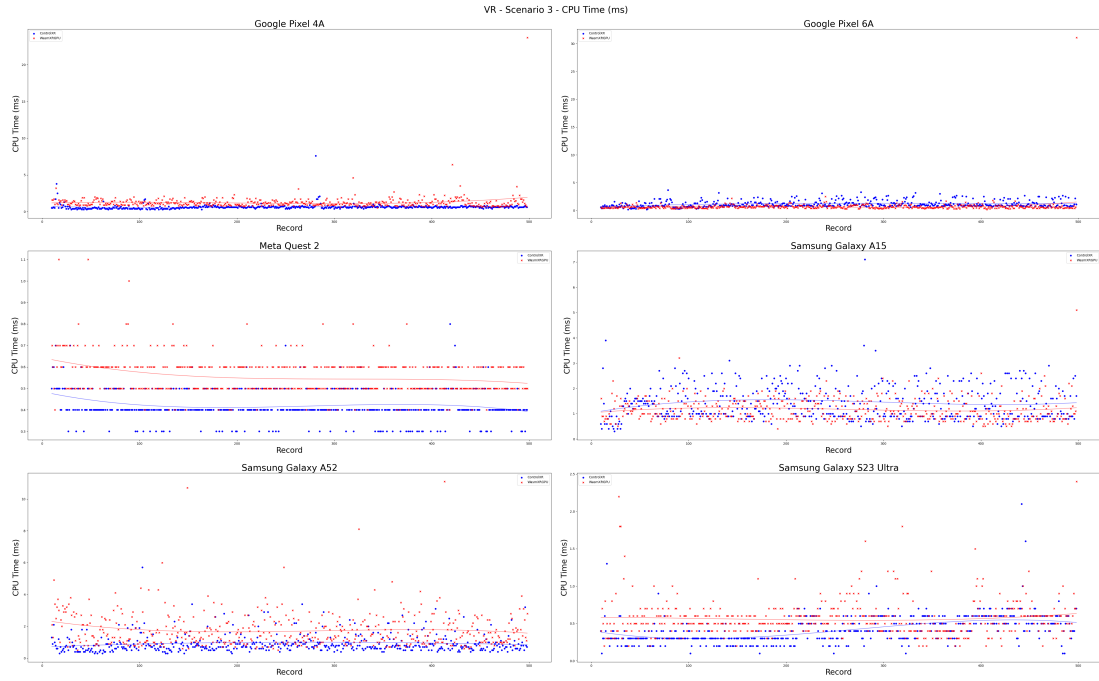


Figure A.3: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

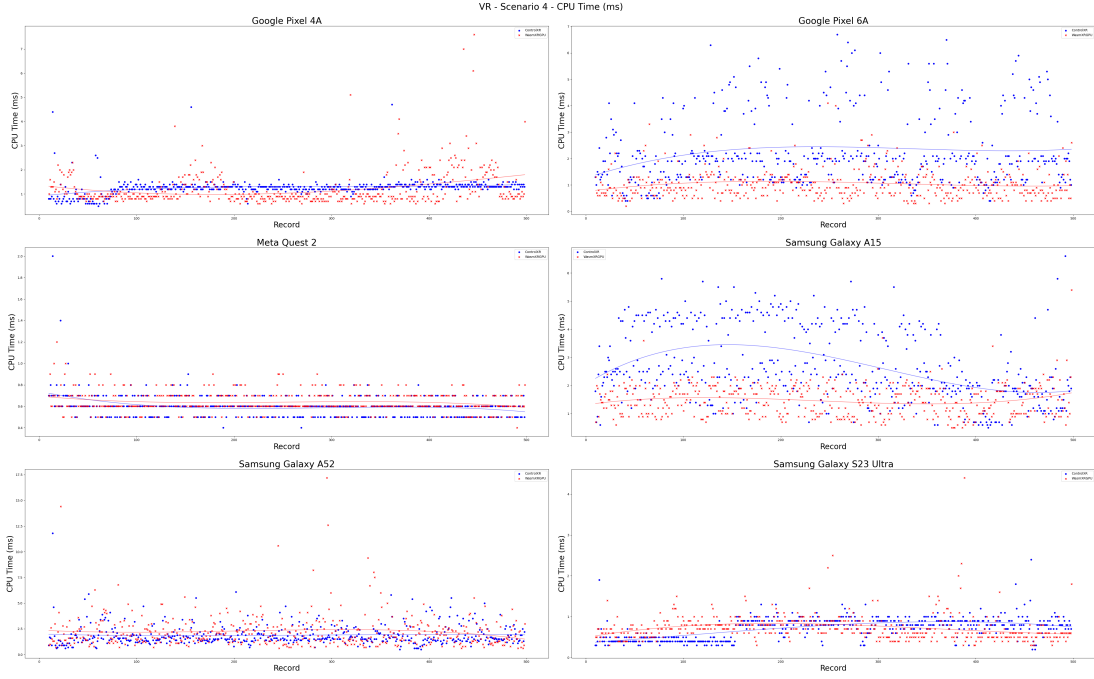


Figure A.4: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

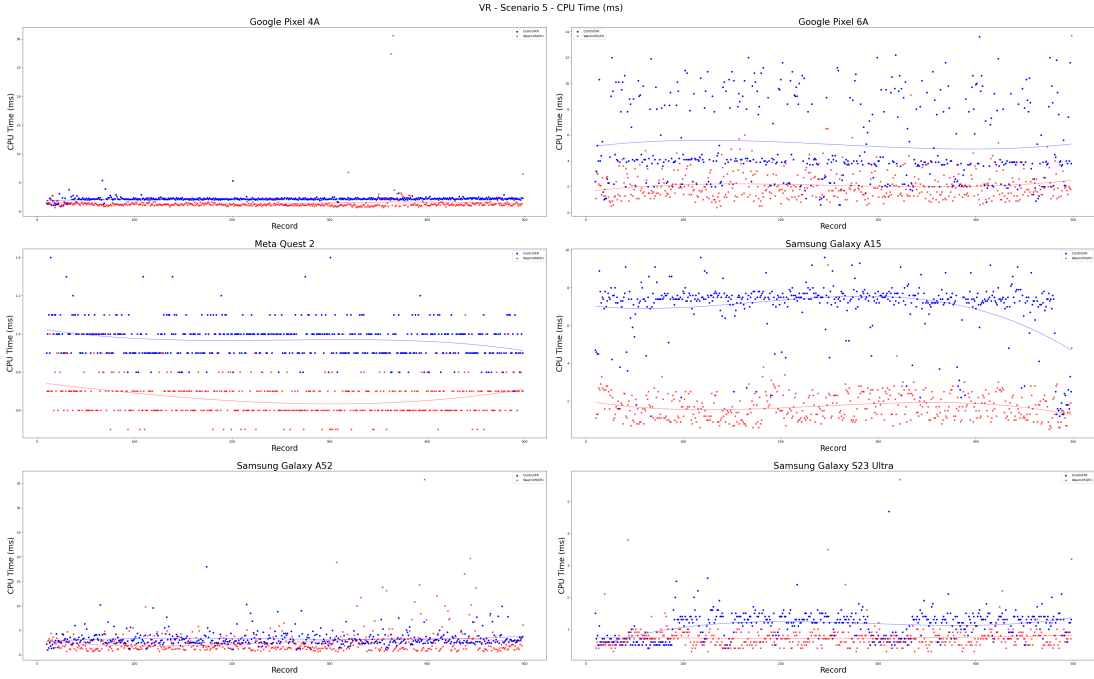


Figure A.5: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

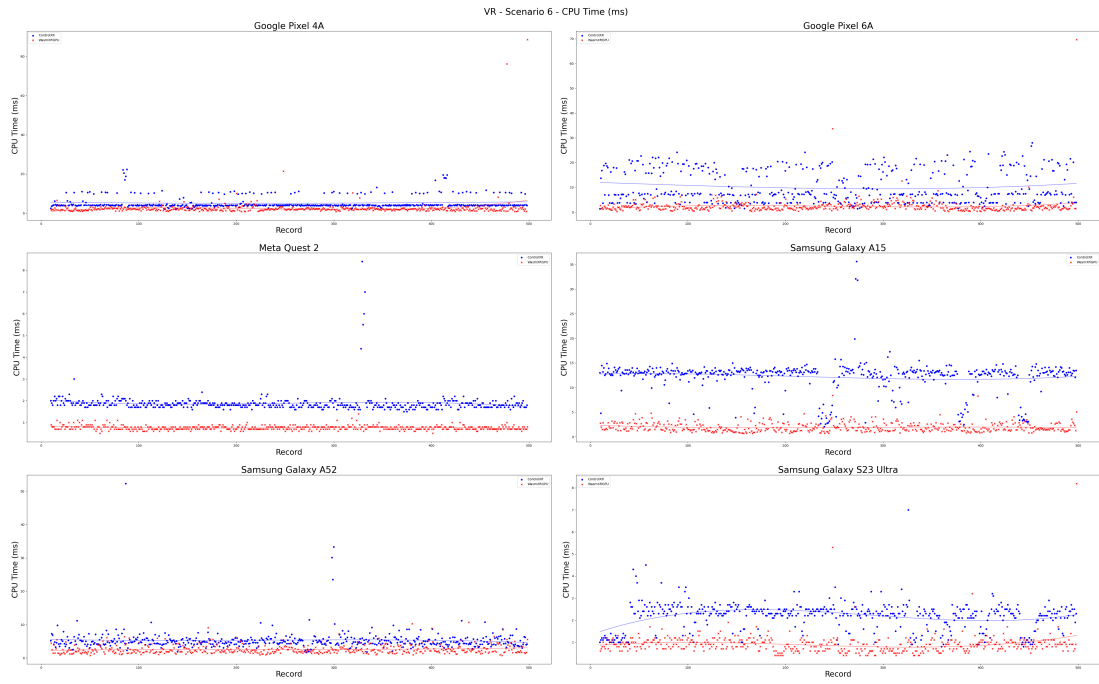


Figure A.6: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

A.1.2 GPU Time (ms)

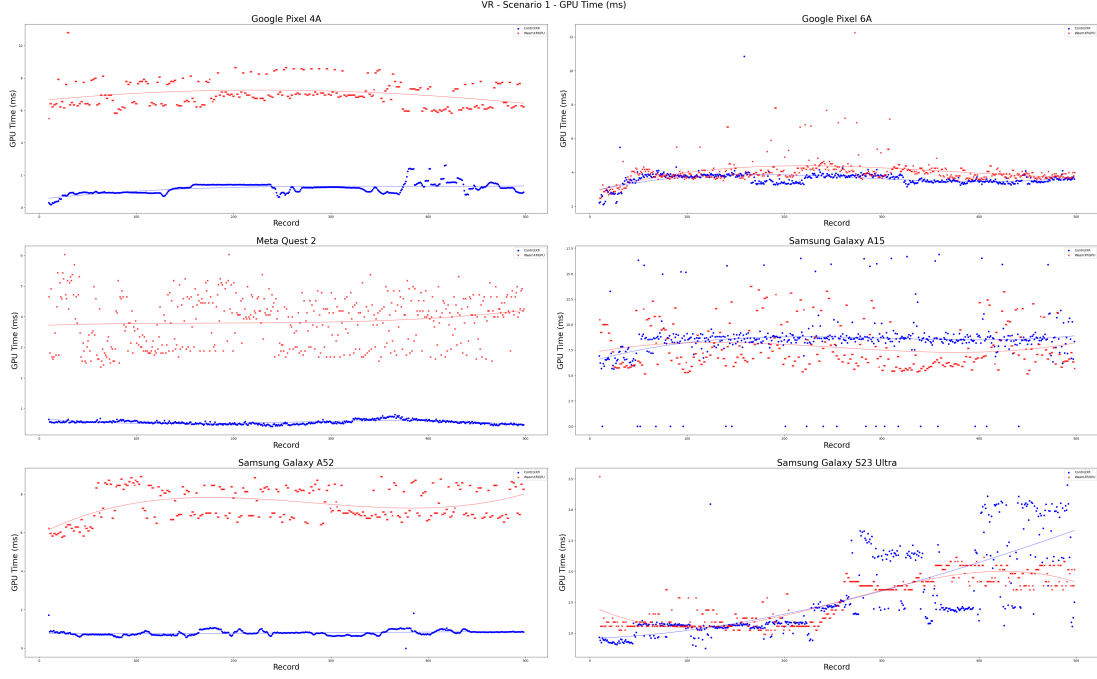


Figure A.7: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

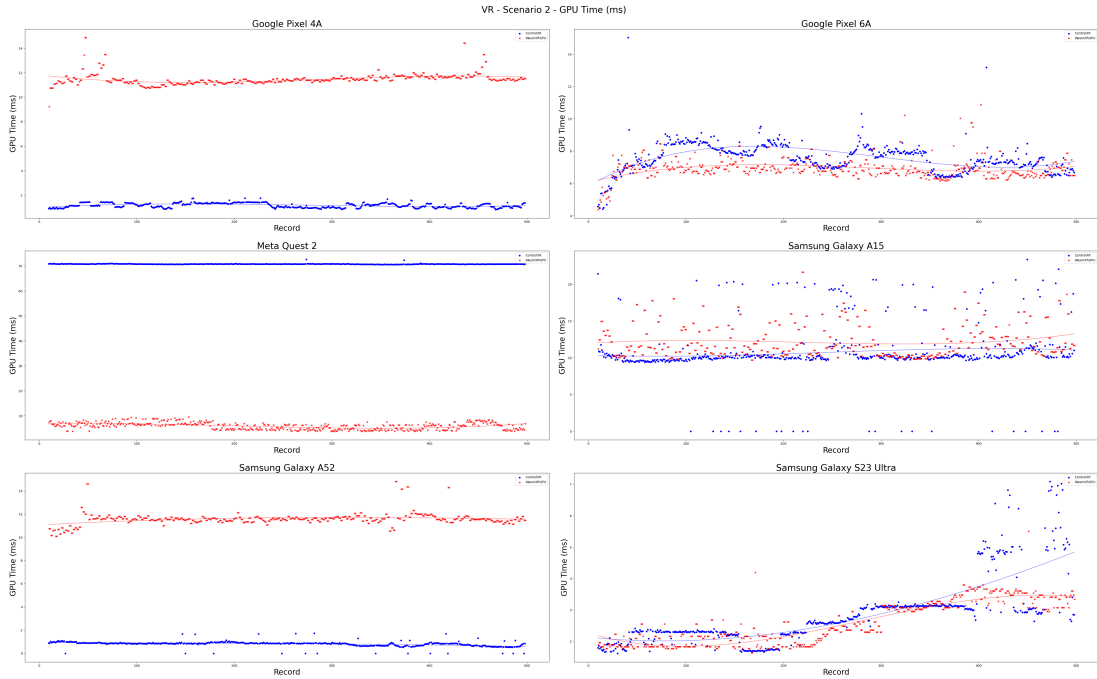


Figure A.8: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

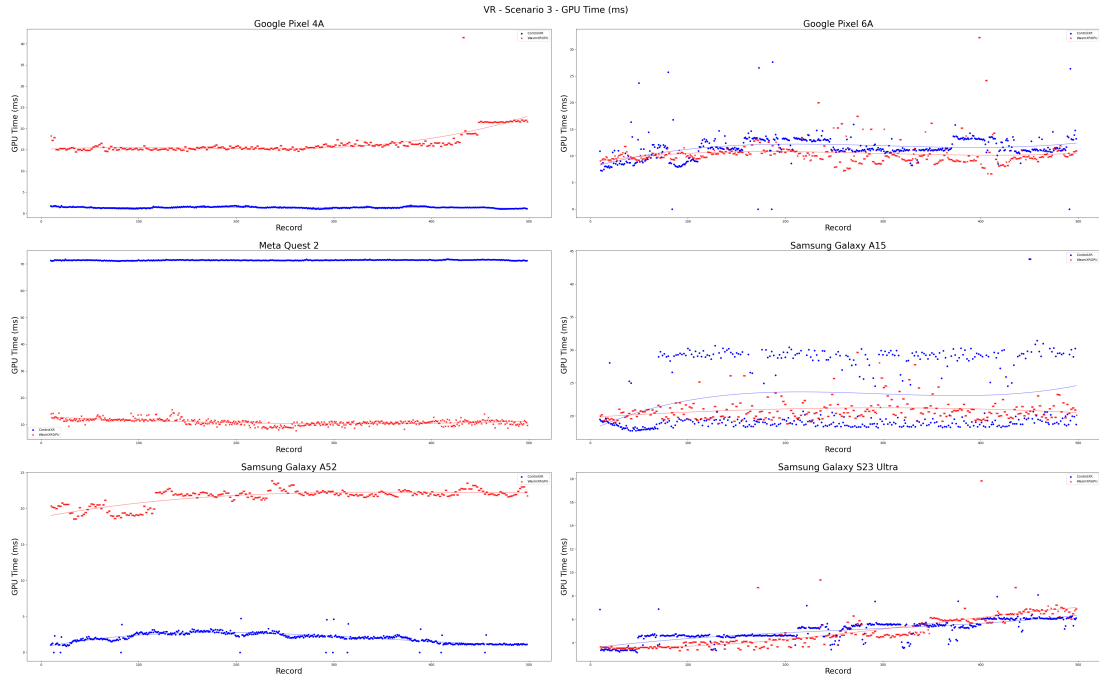


Figure A.9: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

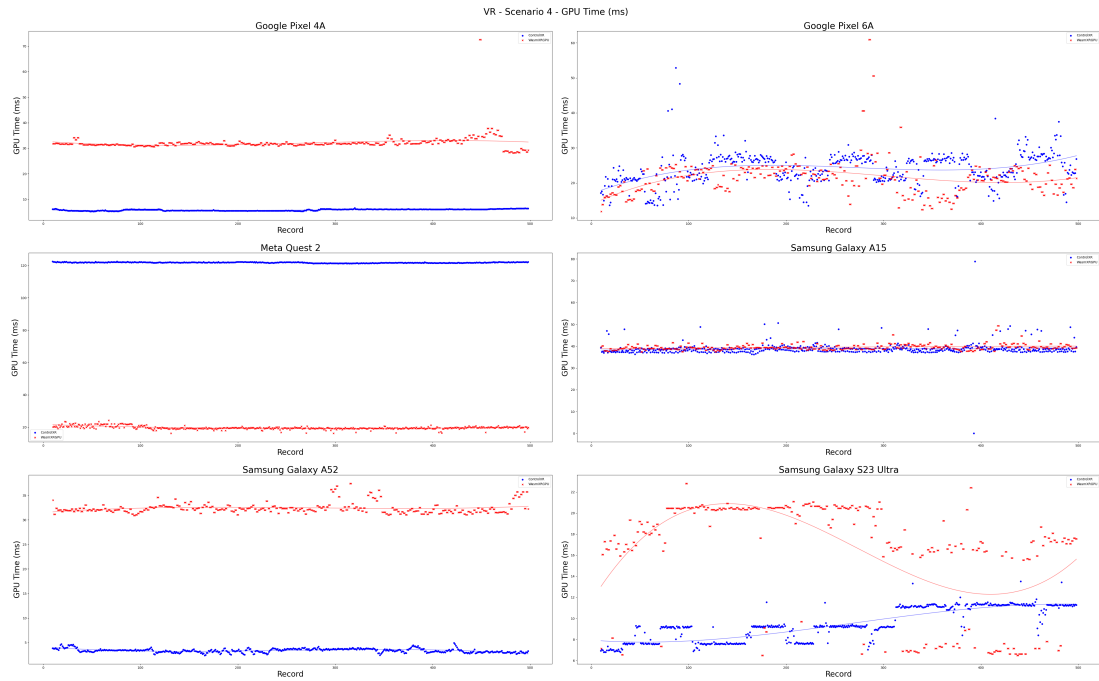


Figure A.10: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

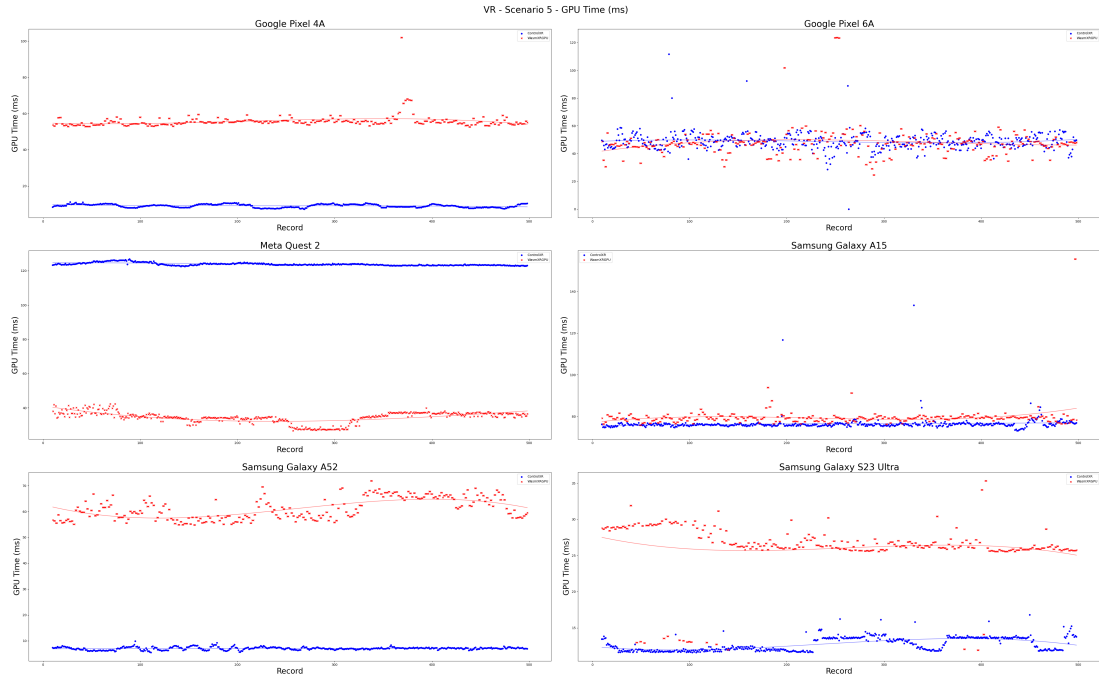


Figure A.11: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

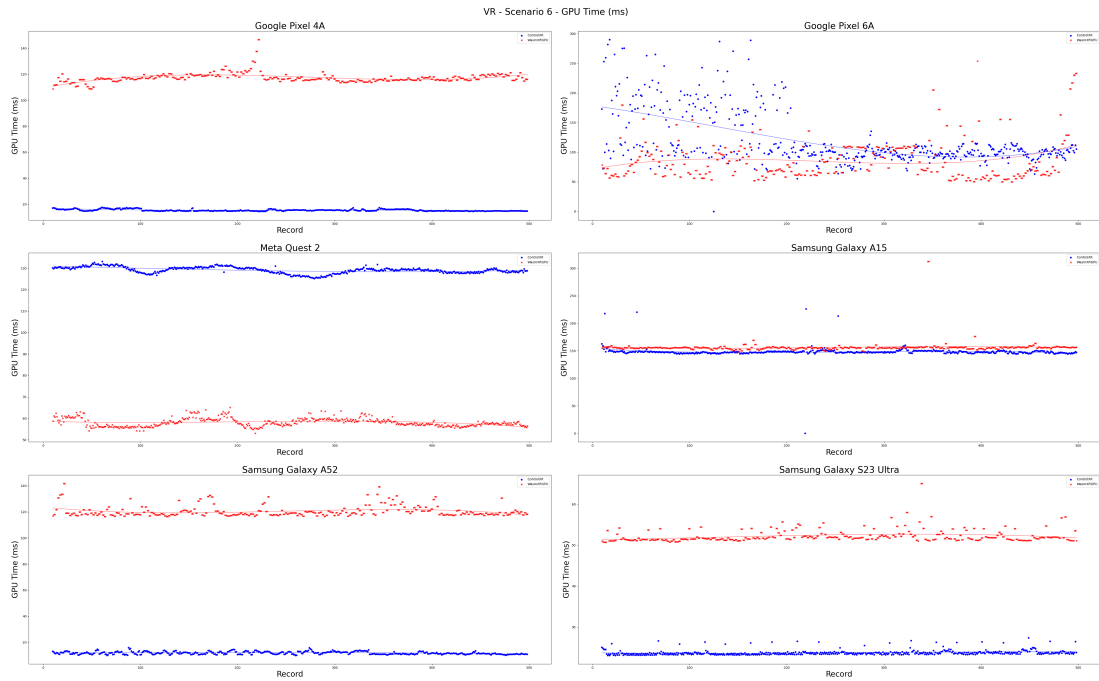


Figure A.12: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Virtual Reality (VR). Lower is better.

A.2 Violin Plots of Measured Devices in VR

A.2.1 Google Pixel 4A

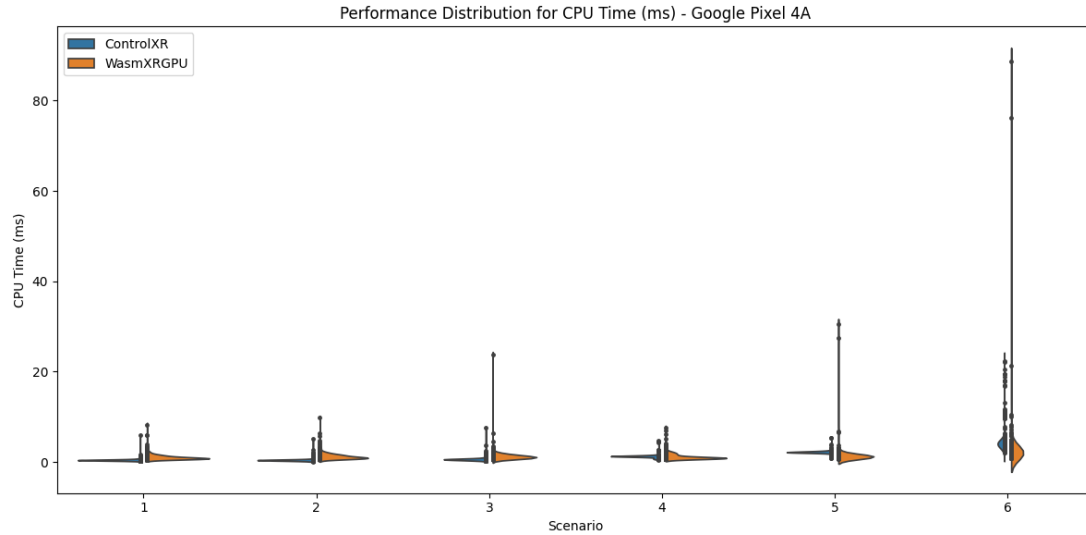


Figure A.13: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Virtual Reality (VR)

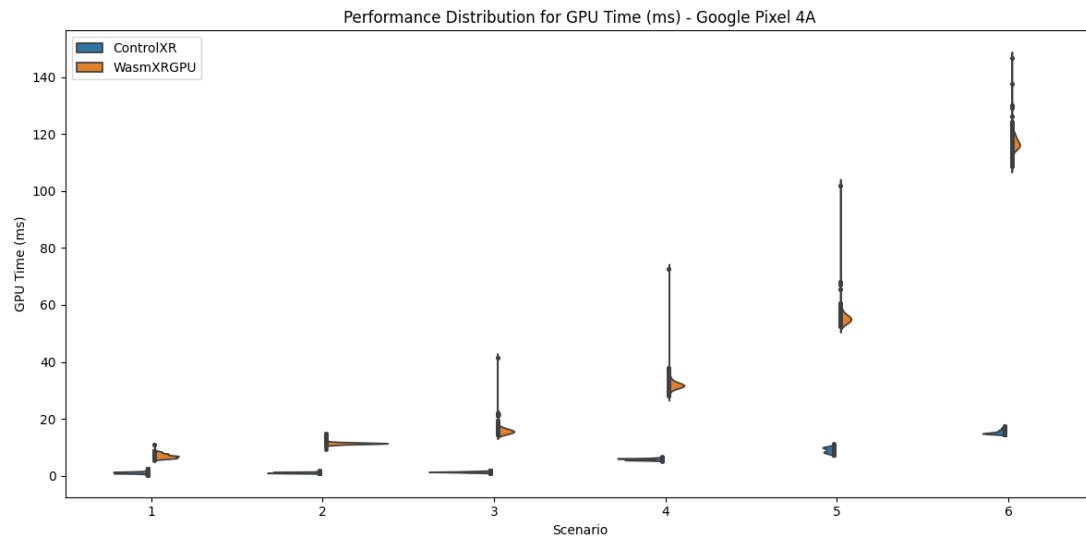


Figure A.14: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Virtual Reality (VR)

A.2.2 Google Pixel 6A

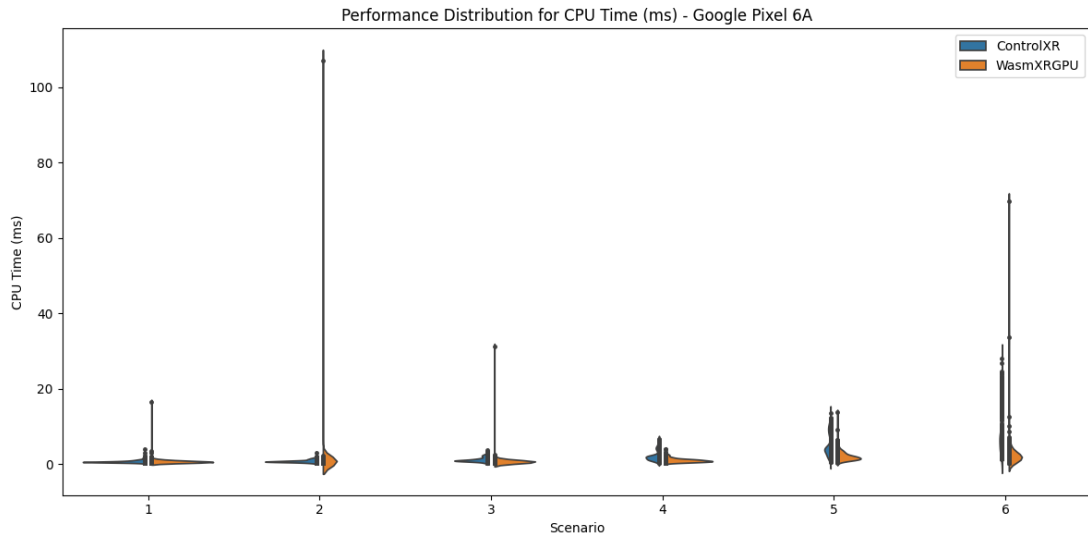


Figure A.15: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Virtual Reality (VR)

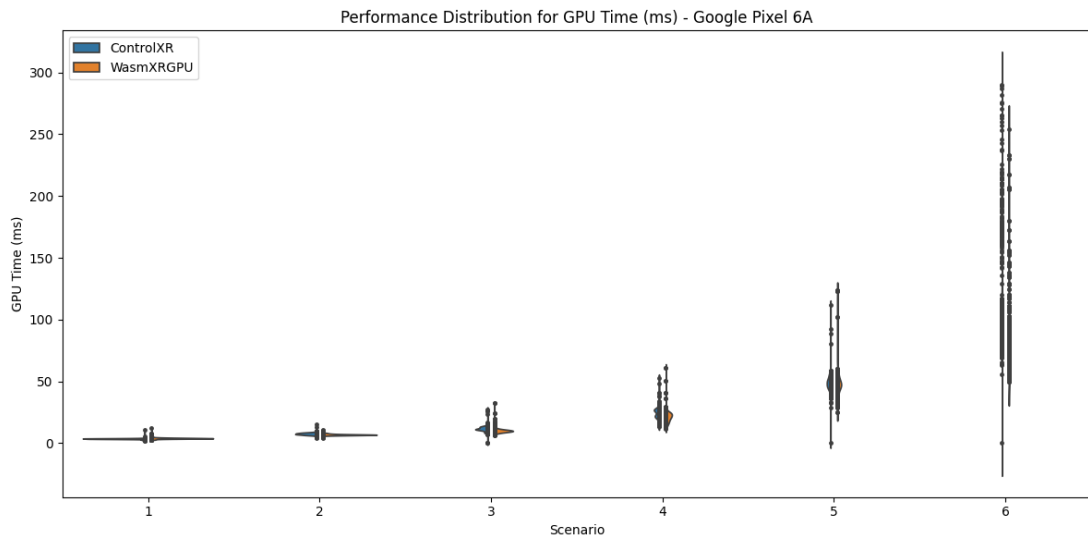


Figure A.16: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Virtual Reality (VR)

A.2.3 Samsung Galaxy A15

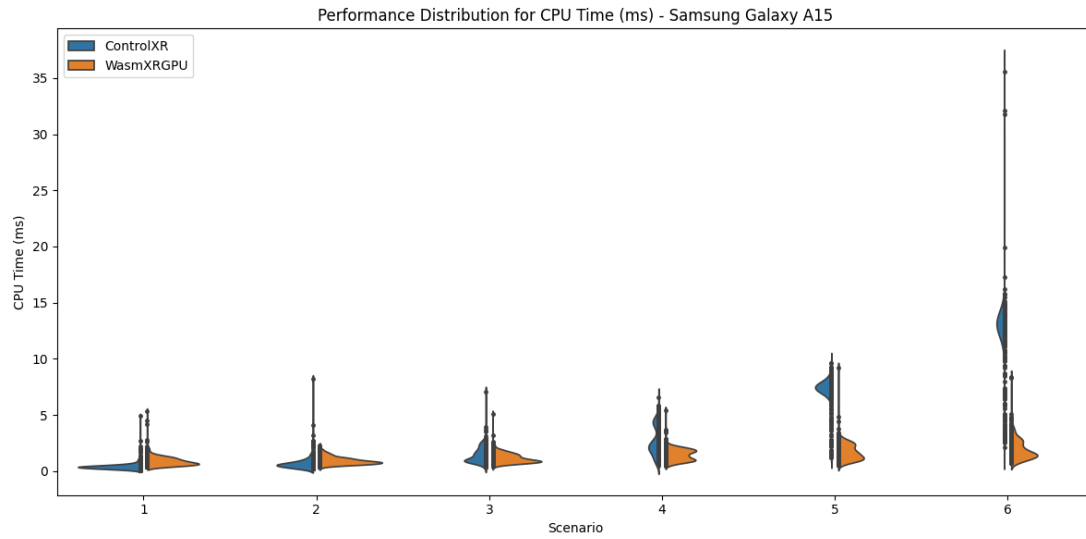


Figure A.17: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Virtual Reality (VR)

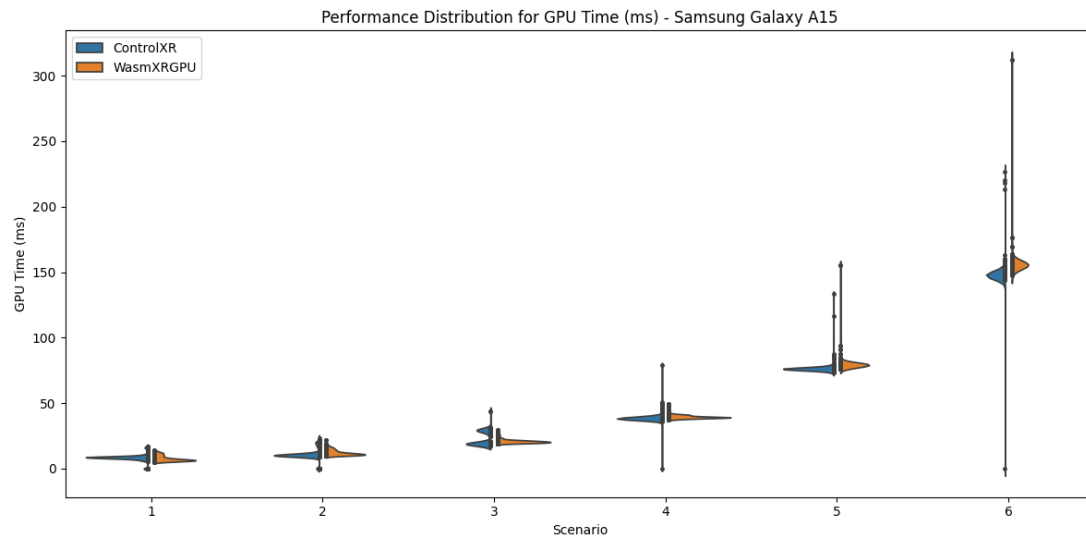


Figure A.18: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Virtual Reality (VR)

A.2.4 Samsung Galaxy A52

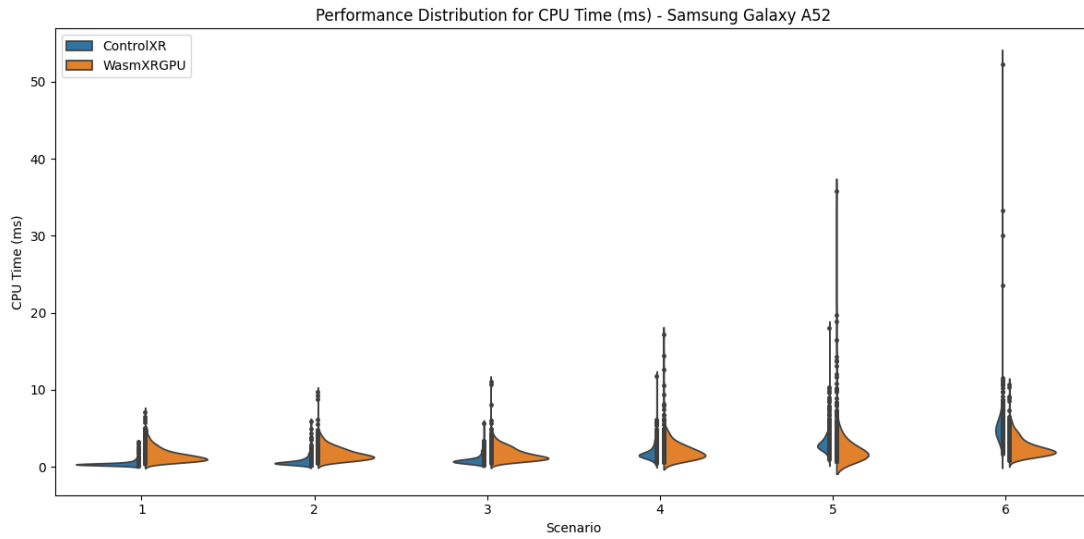


Figure A.19: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Virtual Reality (VR)

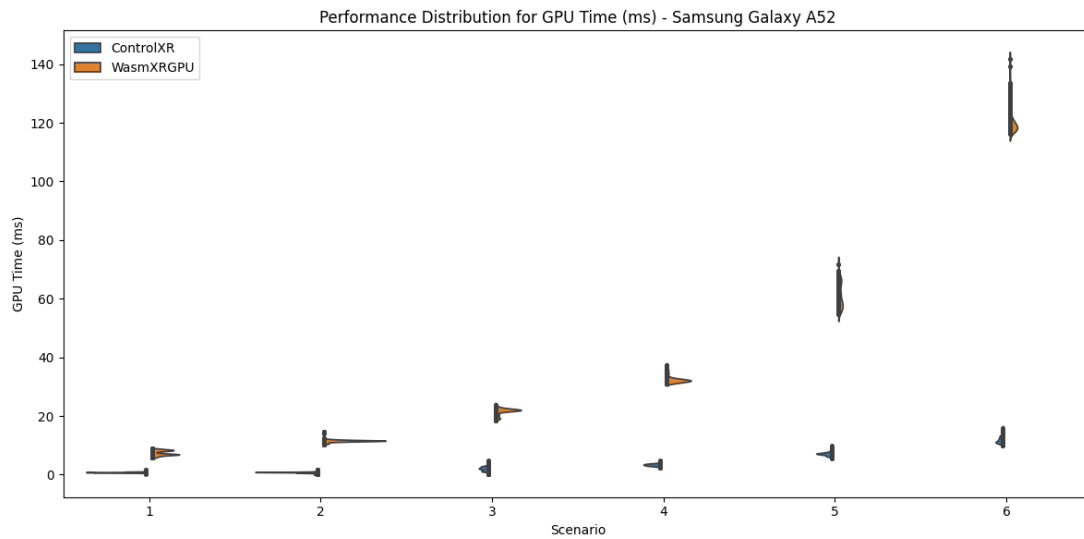


Figure A.20: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Virtual Reality (VR)

A.2.5 Samsung Galaxy S23 Ultra

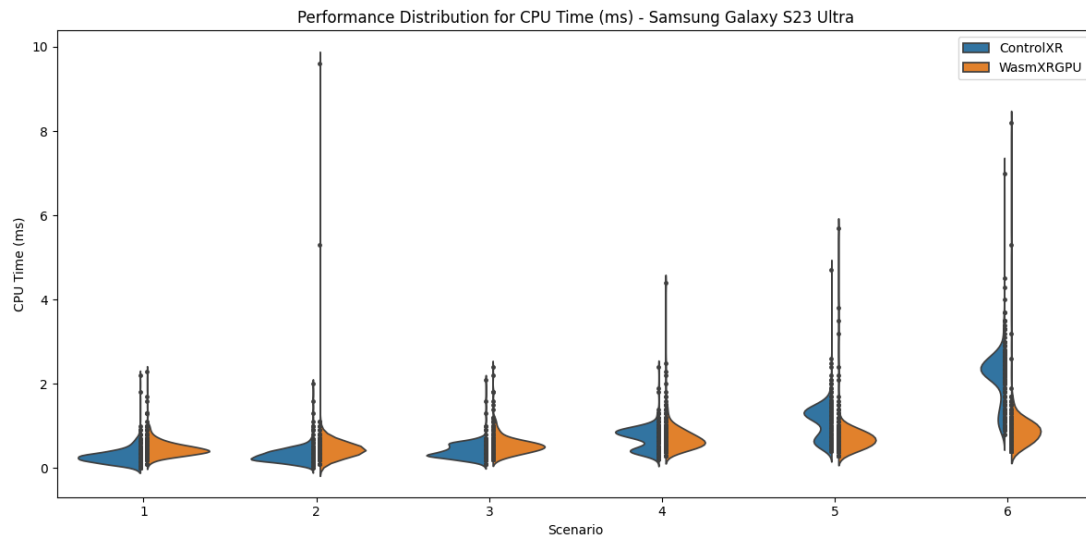


Figure A.21: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Virtual Reality (VR)

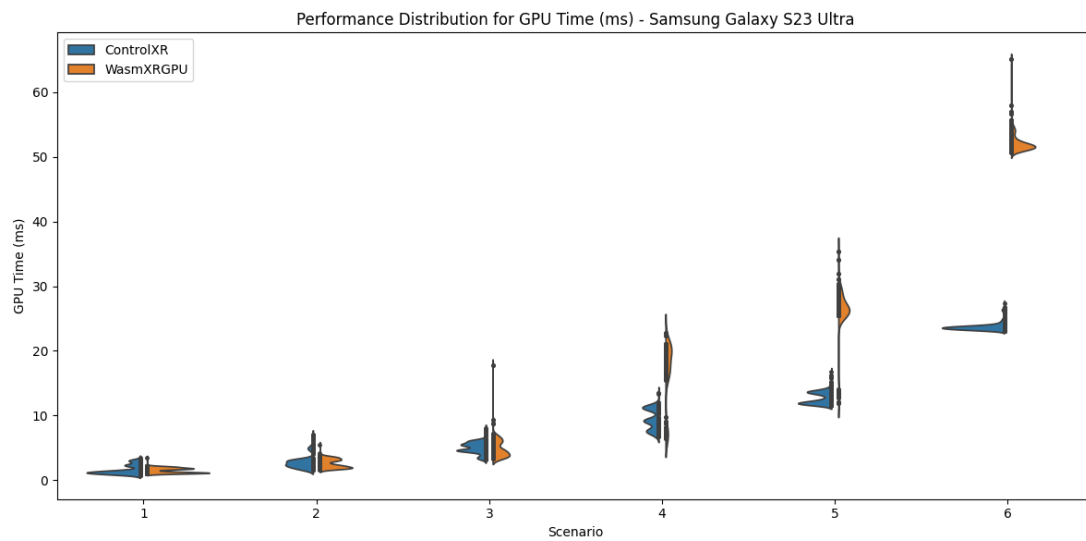


Figure A.22: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Virtual Reality (VR)

A.2.6 Meta Quest 2

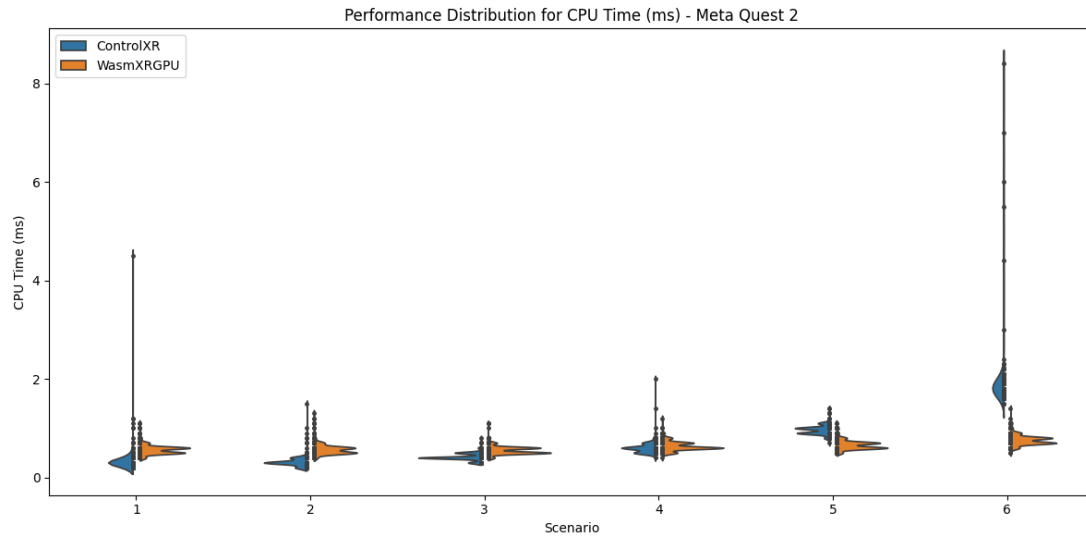


Figure A.23: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Virtual Reality (VR)

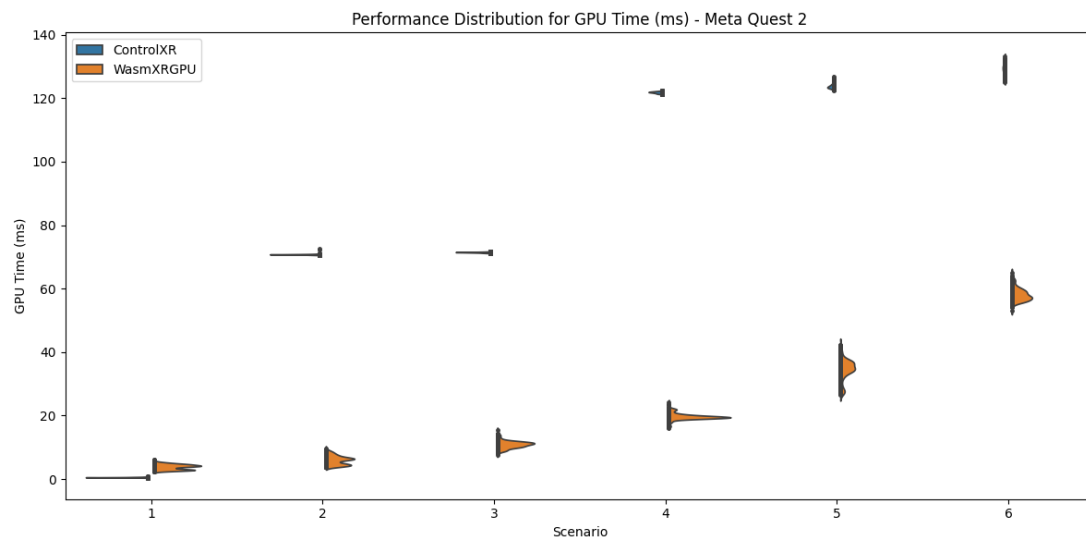


Figure A.24: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Virtual Reality (VR)

A.3 Scatter Plots of Measured Metrics in AR

A.3.1 CPU Time (ms)

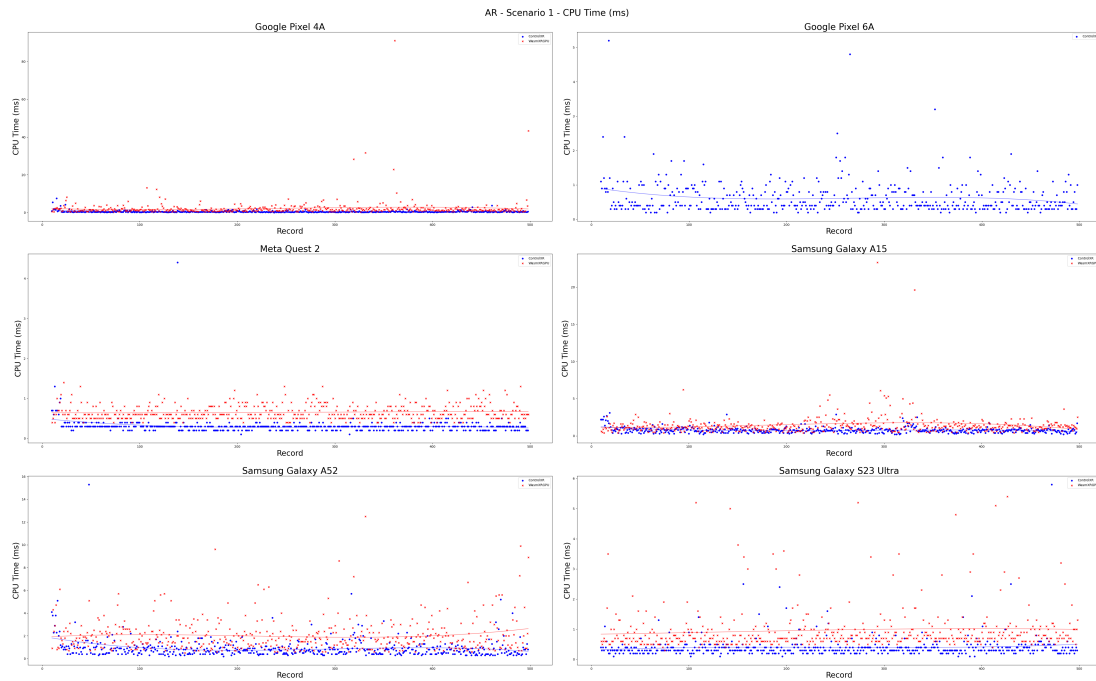


Figure A.25: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

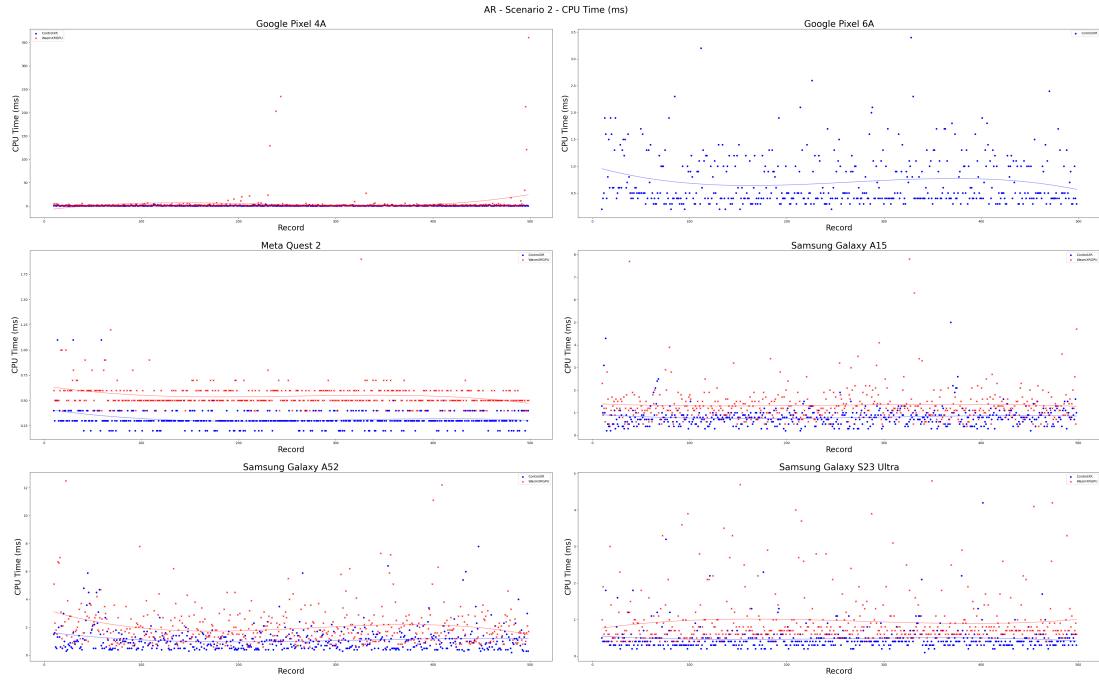


Figure A.26: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

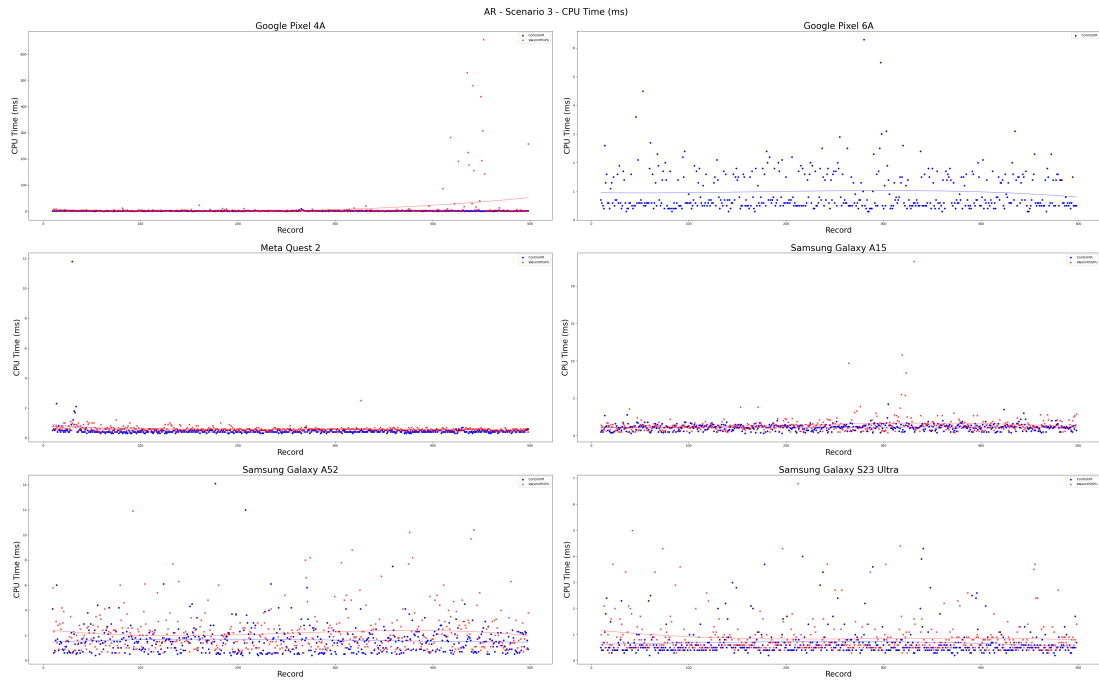


Figure A.27: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

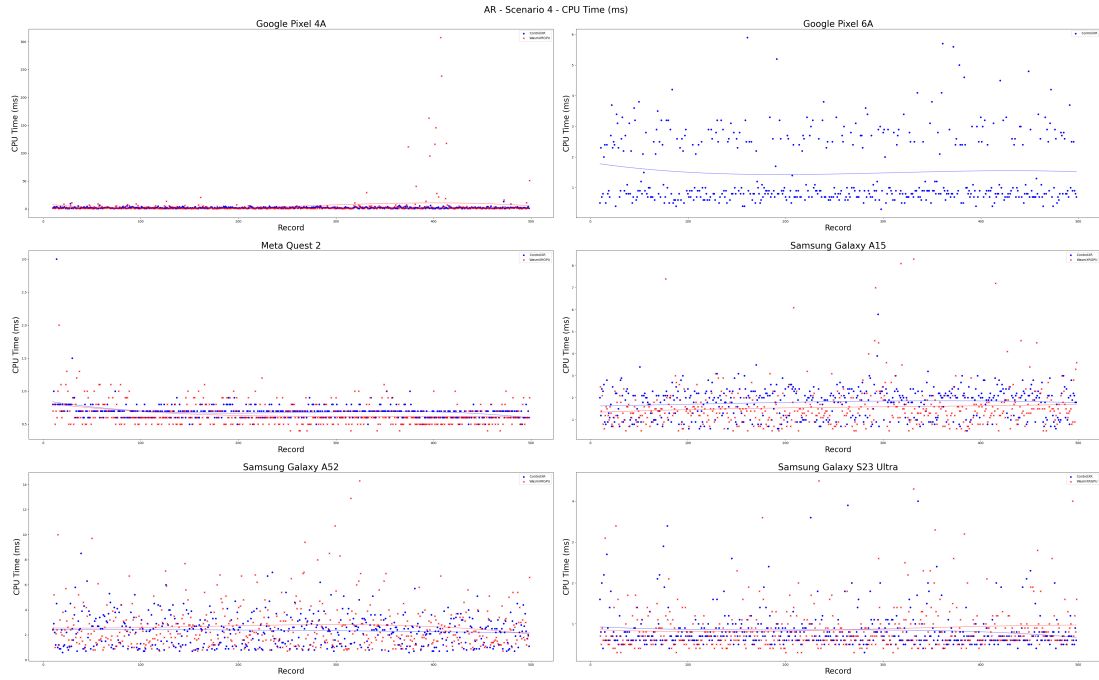


Figure A.28: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

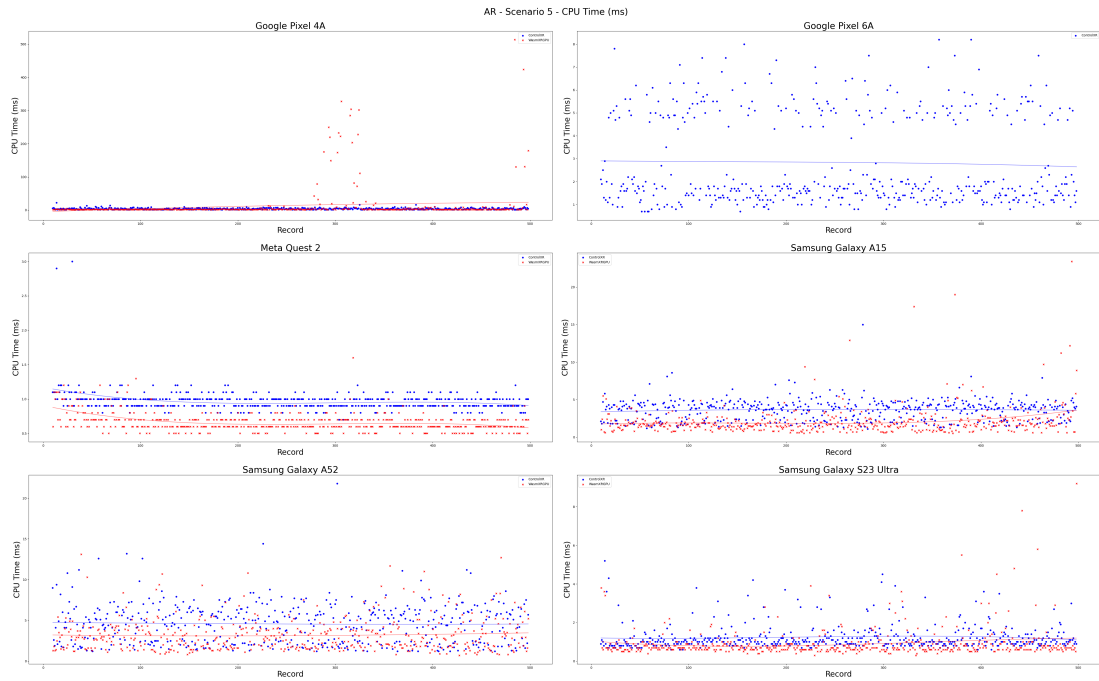


Figure A.29: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

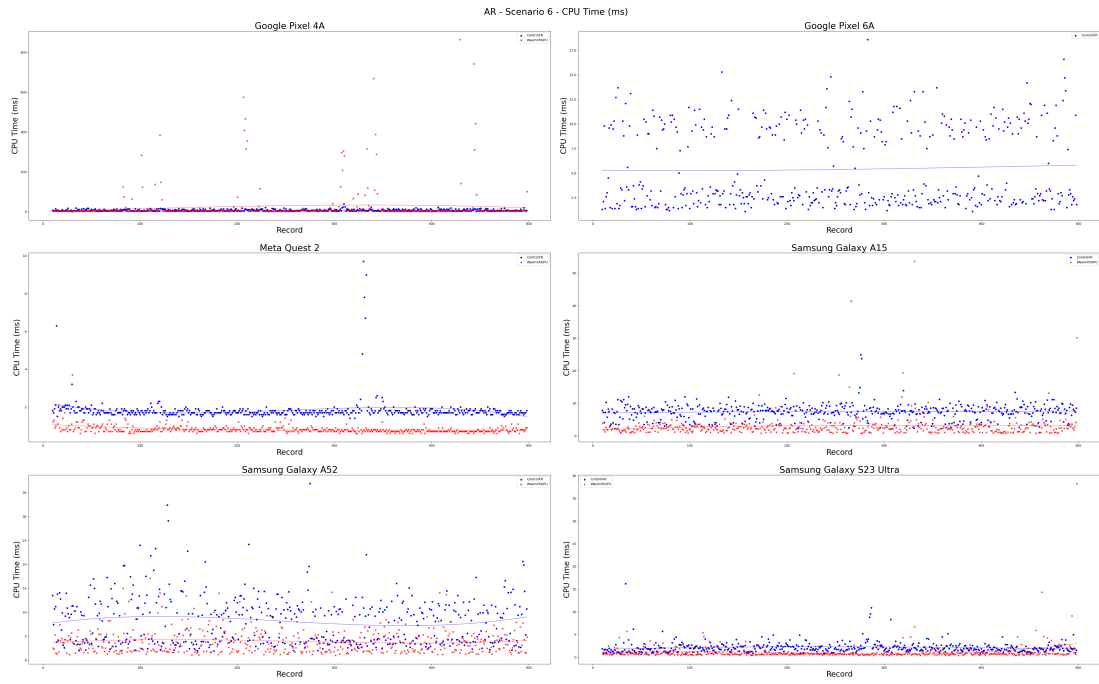


Figure A.30: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

A.3.2 GPU Time (ms)

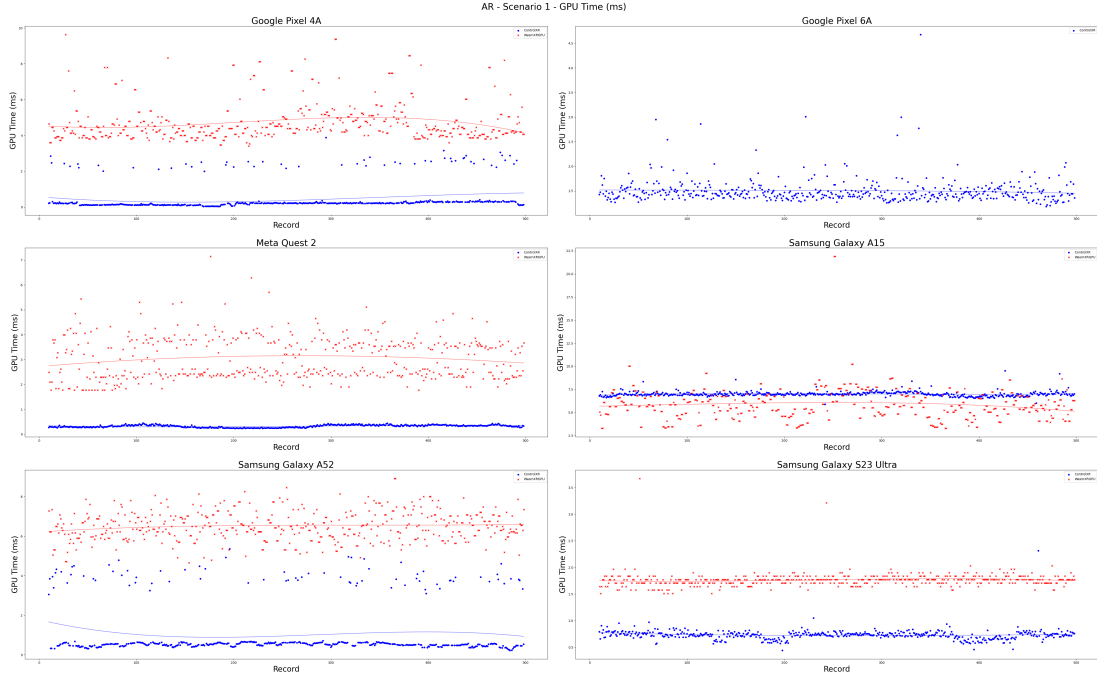


Figure A.31: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 1 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

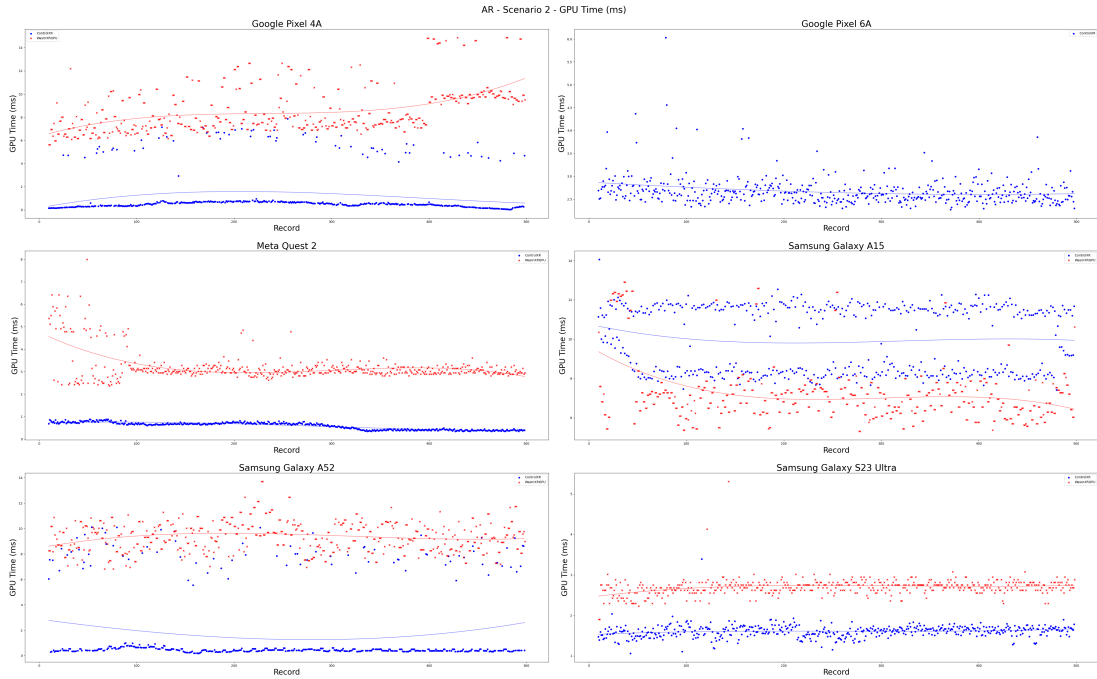


Figure A.32: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 2 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

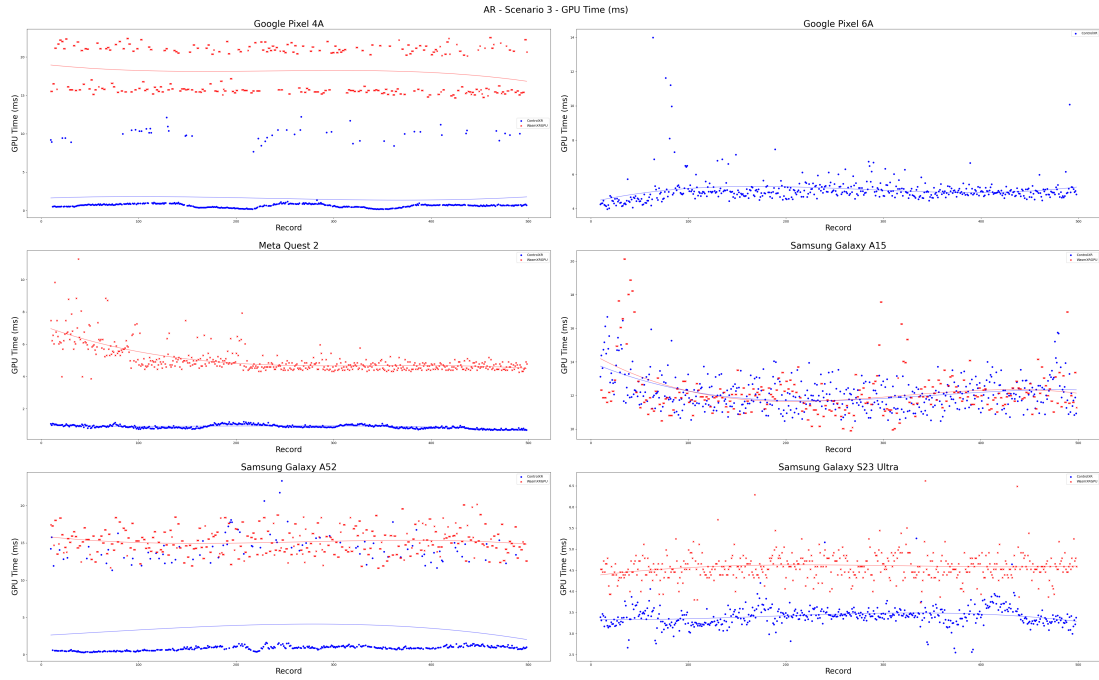


Figure A.33: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 3 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

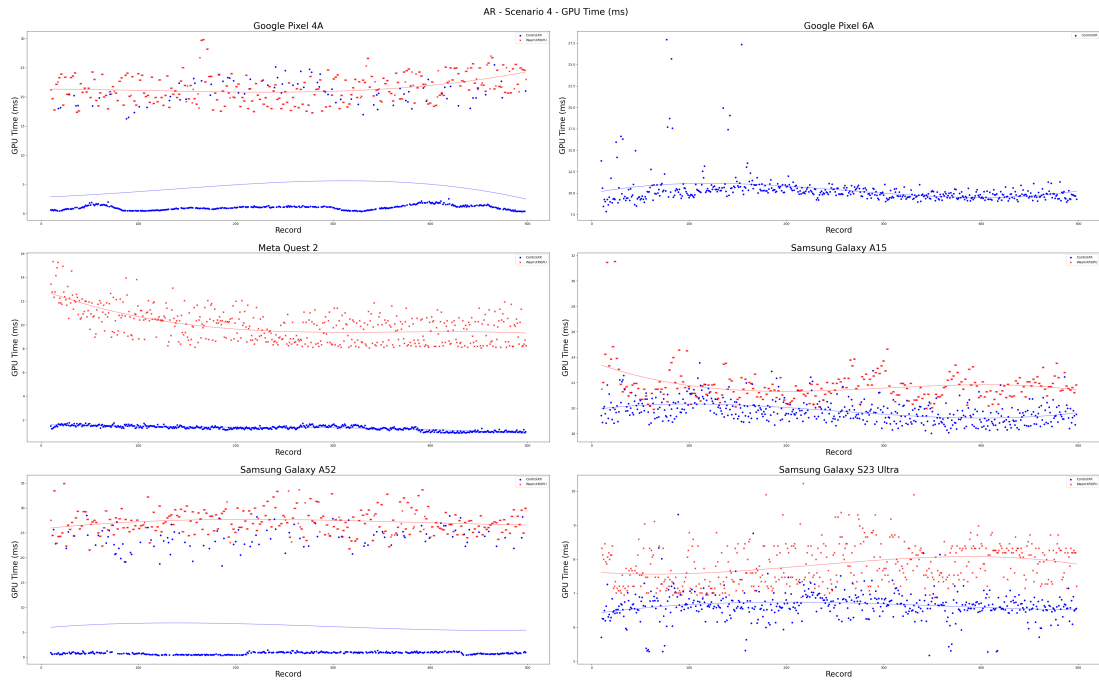


Figure A.34: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 4 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

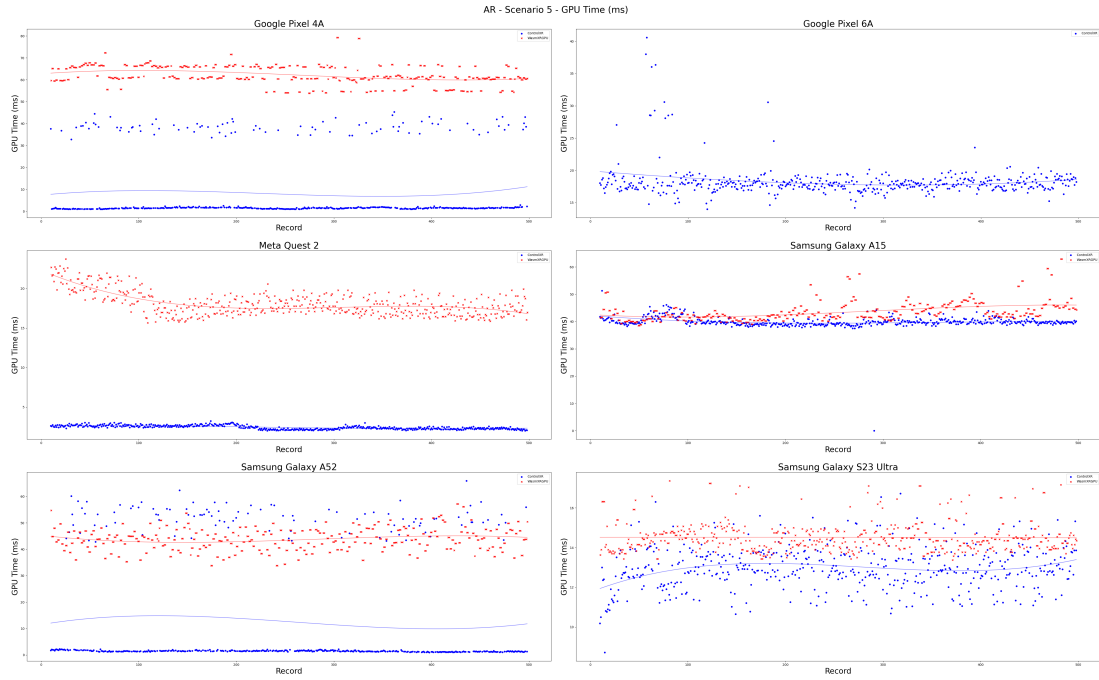


Figure A.35: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 5 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

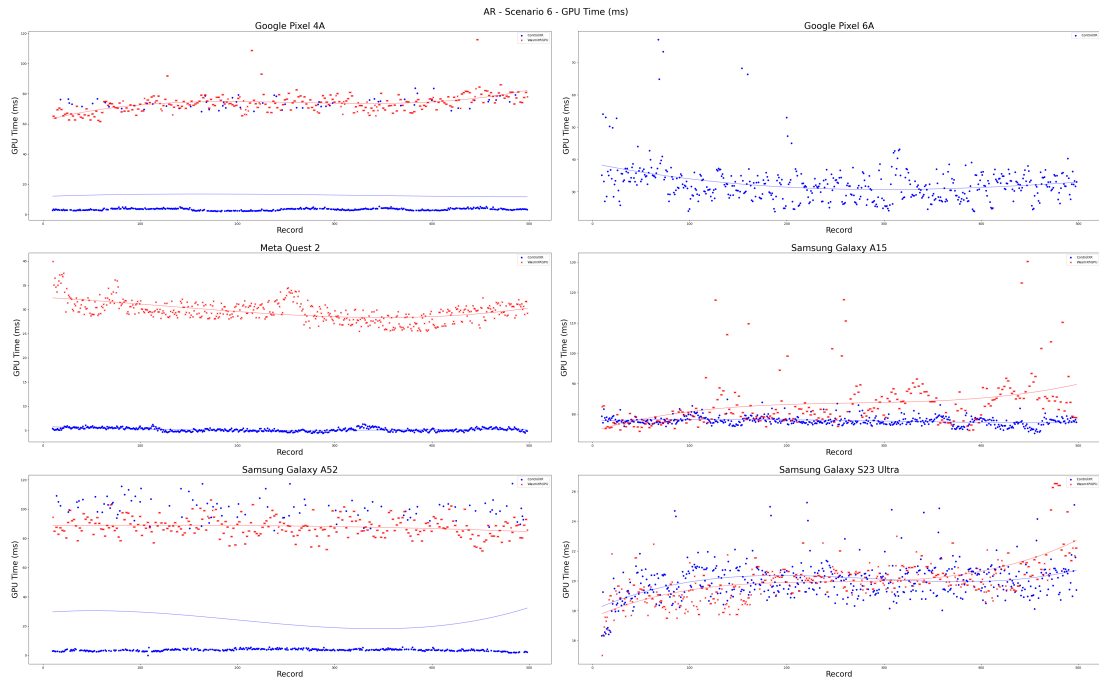


Figure A.36: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenario 6 in Table 5.3 and the devices in Table 5.4, in the context of Augmented Reality (AR). Lower is better.

A.4 Violin Plots of Measured Devices in AR

A.4.1 Google Pixel 4A

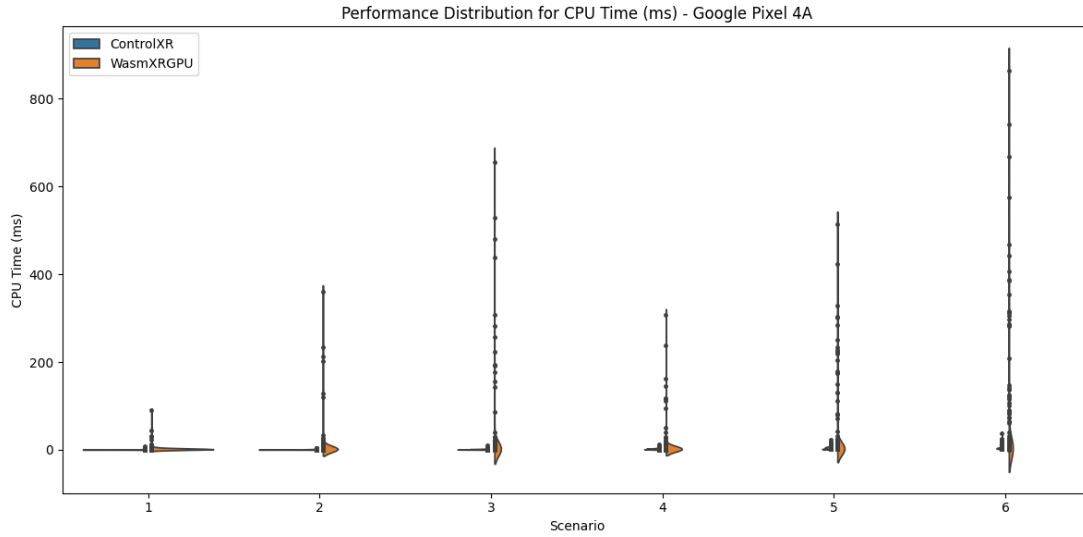


Figure A.37: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Augmented Reality (AR)

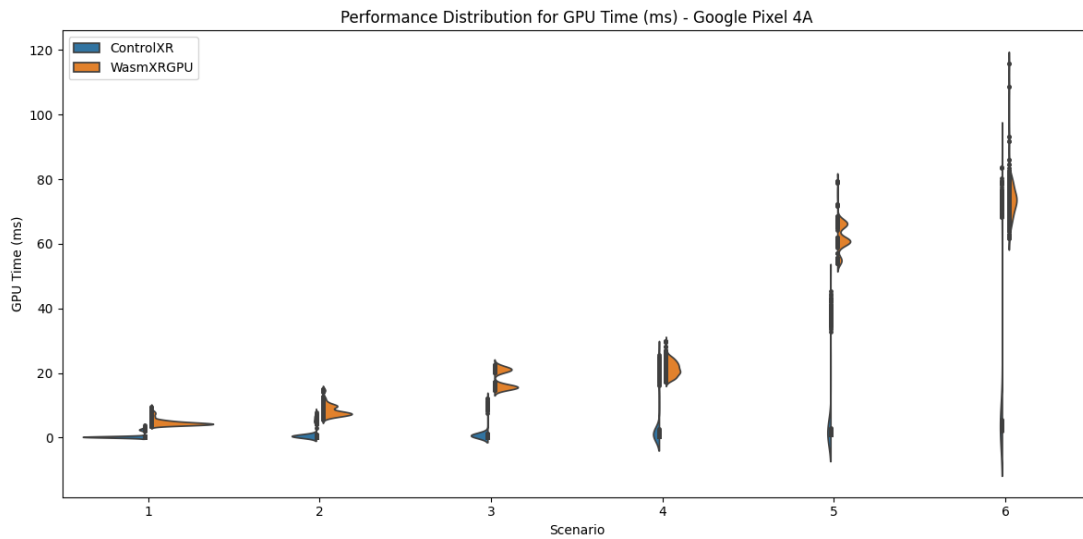


Figure A.38: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 4A, in the context of Augmented Reality (AR)

A.4.2 Google Pixel 6A

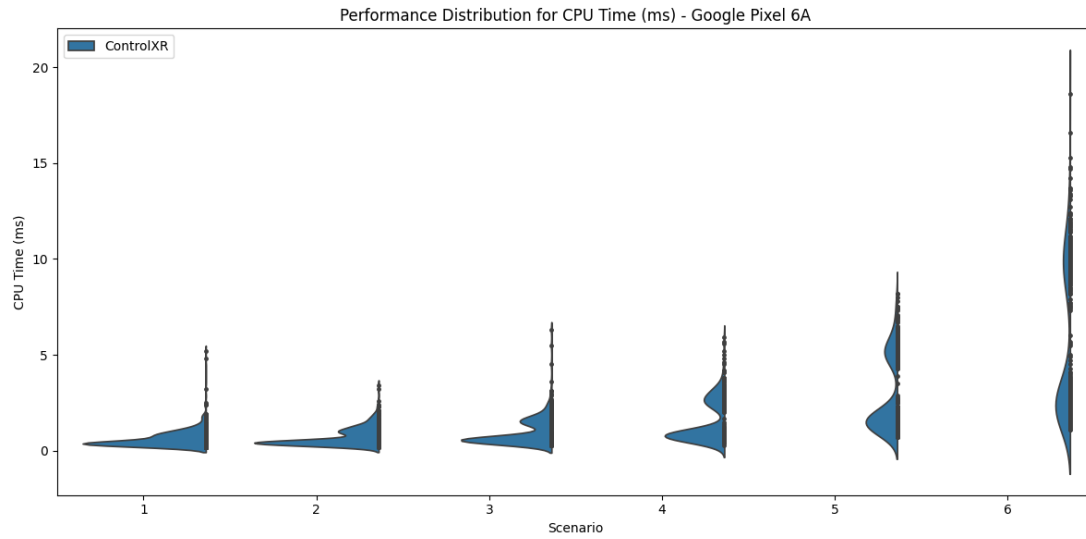


Figure A.39: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Augmented Reality (AR)

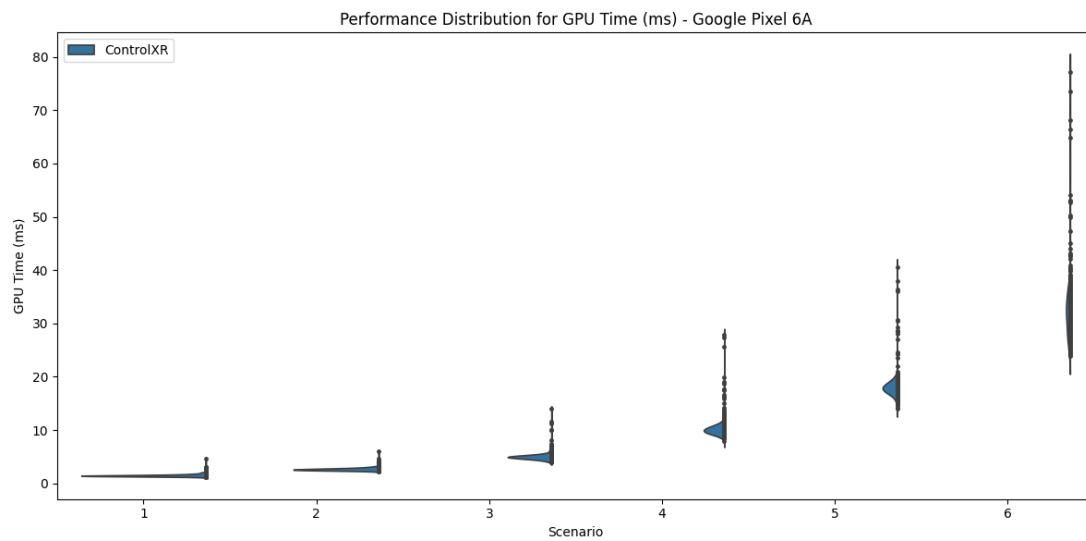


Figure A.40: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Google Pixel 6A, in the context of Augmented Reality (AR)

A.4.3 Samsung Galaxy A15

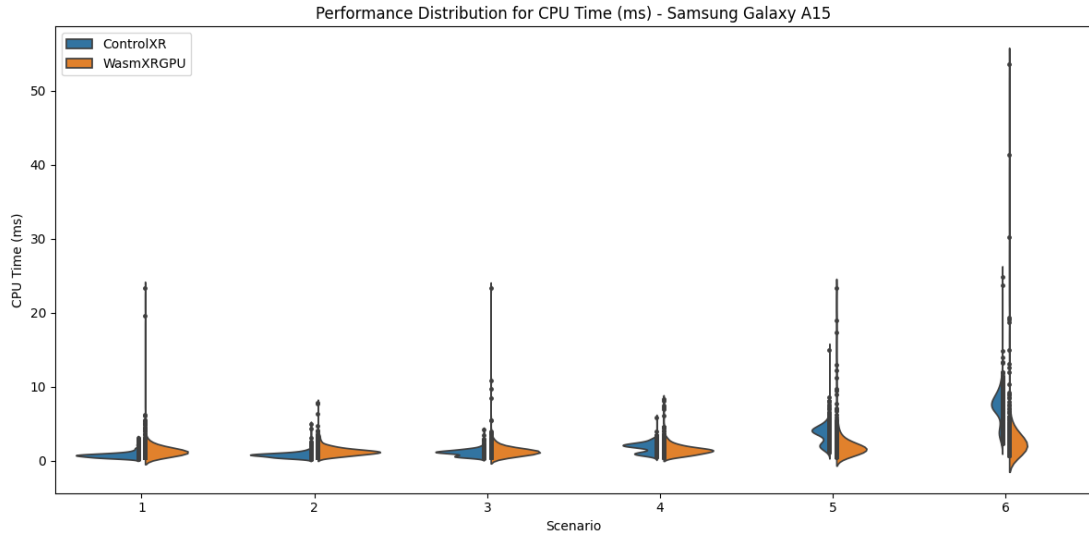


Figure A.41: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Augmented Reality (AR)

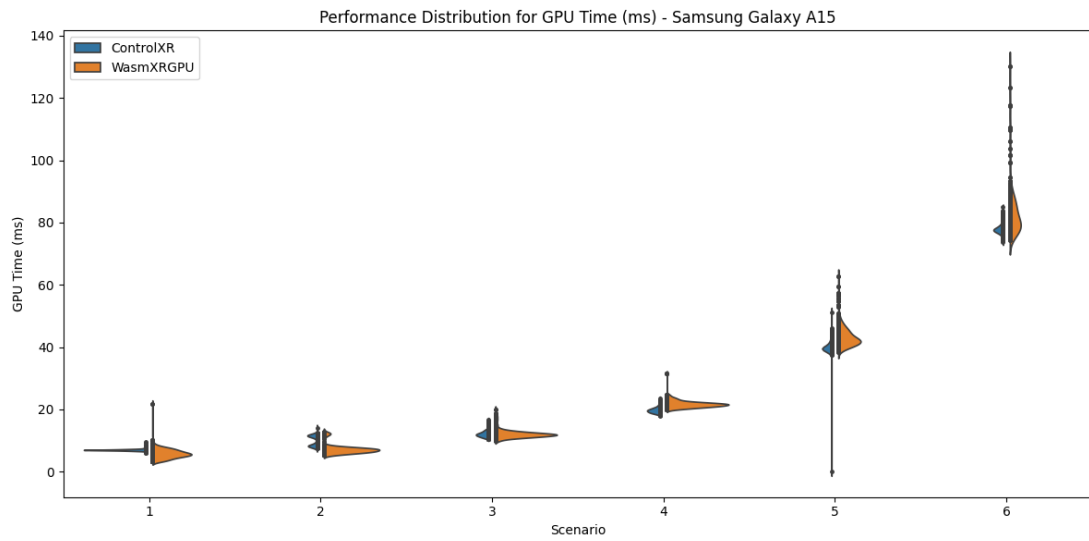


Figure A.42: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A15, in the context of Augmented Reality (AR)

A.4.4 Samsung Galaxy A52

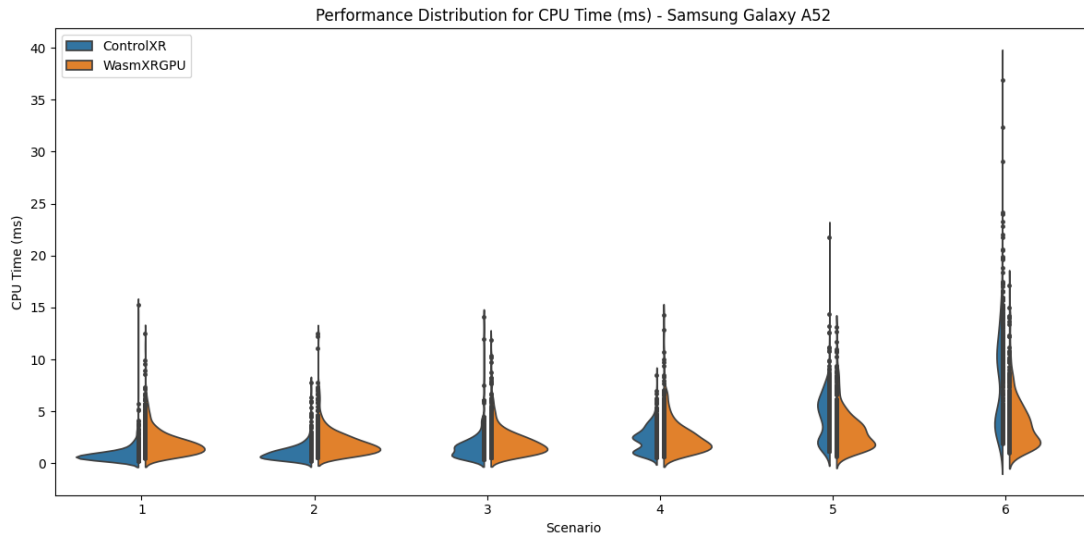


Figure A.43: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Augmented Reality (AR)

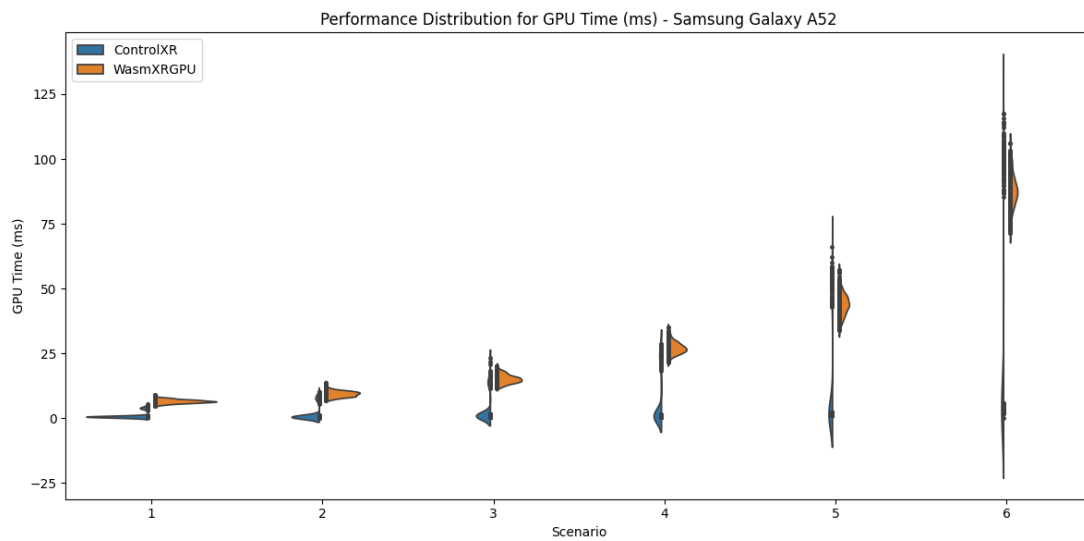


Figure A.44: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy A52, in the context of Augmented Reality (AR)

A.4.5 Samsung Galaxy S23 Ultra

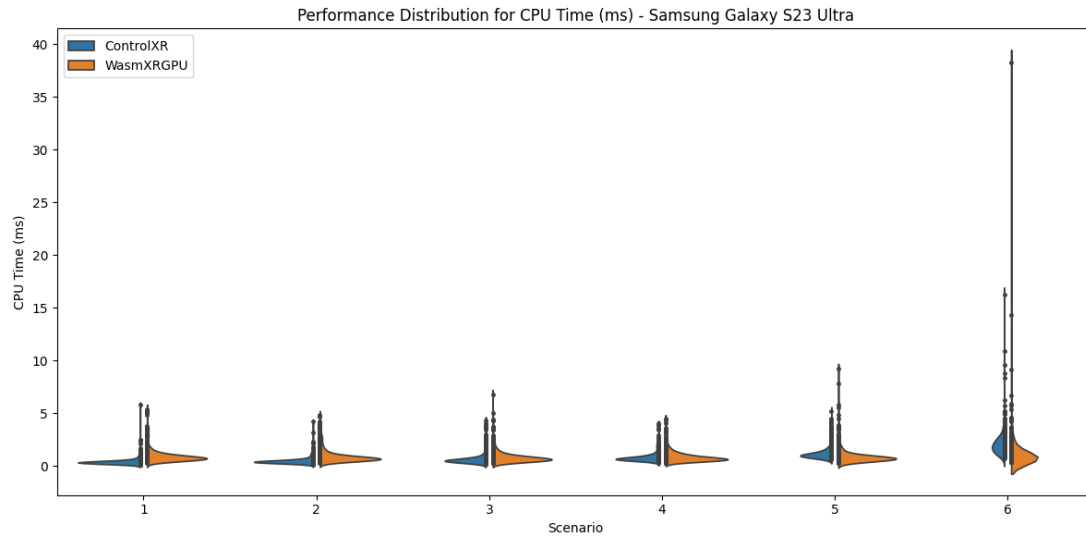


Figure A.45: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Augmented Reality (AR)

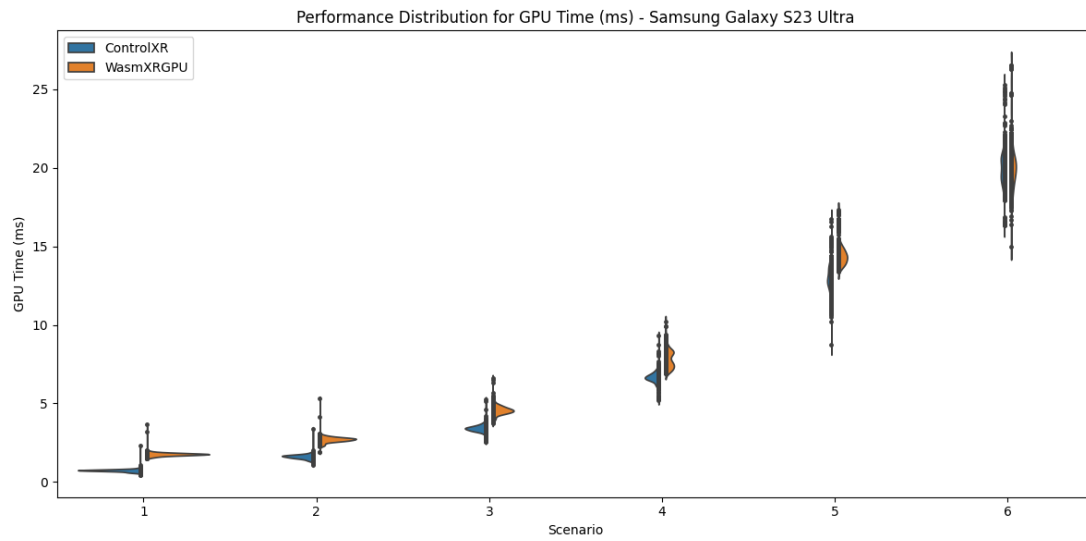


Figure A.46: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Samsung Galaxy S23 Ultra, in the context of Augmented Reality (AR)

A.4.6 Meta Quest 2

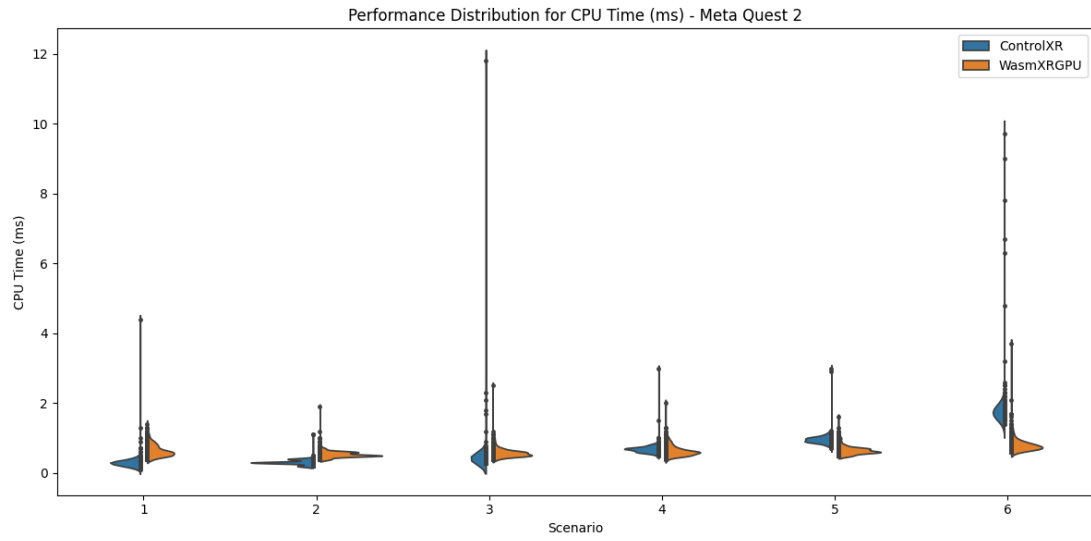


Figure A.47: Distribution of CPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Augmented Reality (AR)

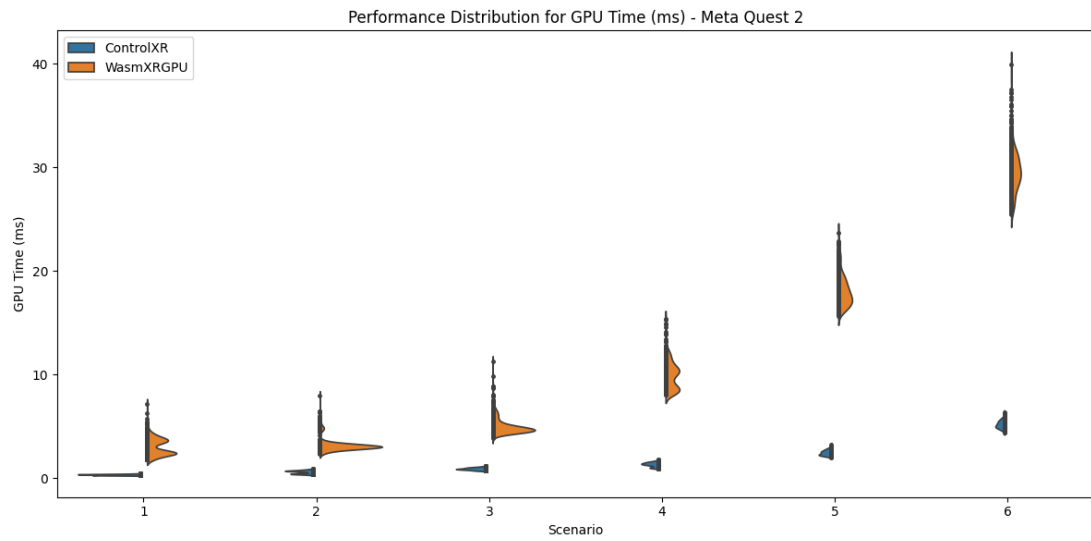


Figure A.48: Distribution of GPU time between ControlXR and WasmXRGPU with respect to the scenarios defined in Table 5.3 for Meta Quest 2, in the context of Augmented Reality (AR)