RICE UNIVERSITY

Optimizing Web Virtual Reality

by

Rabimba Karanjai

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Vivek Sarkar

Vivek Sarkar, Chair
Research Professor, Department of
Computer Science

Ray Simar
Professor of Electrical and Computer
Engineering

Lin Zhong
Professor of Electrical and Computer
Engineering and Computer Science

Houston, Texas

December, 2017

ABSTRACT

Optimizing Web Virtual Reality

by

Rabimba Karanjai

Performance has always been a key factor in any virtual and augmented reality experience. Since Virtual Reality was conceived, performance has always been the factor that has often slowed down, or at times even halted the adoption of Virtual Reality related technologies. More recently, the hardware advancements have caught up with the development so that virtual reality experiences can be rendered satisfactorily. The performance gains, however, still depend a lot on both the hardware and the software platform that we use. With mobile phones becoming one of the primary devices to consume media, it is critical to pay attention to how these applications perform on portable devices. With help of the Web Graphics Library (WebGL), it is now possible to create Web Virtual Reality capable experiences that can directly be executed on supported web browsers. However, that raises new challenges like making these JavaScript-based web applications run with near-native performances for the user. Immersive reality applications, like those built for WebVR, assume that performance will always be satisfactory to avoid both screen latencies and physical side effects such as nausea. This thesis presents a collection of optimizations targeted specifically at WebGL and the library, Three.js – on top of which most Web Virtual Reality applications are built including the Mozilla aframe library – though the principles behind our optimizations can be applied to other frameworks as well. Our

approach identifies certain aspects and pain-points in the present framework including object loading, texture rendering and stereoscopic image production. We propose, implement and test our approach to optimize these aspects, and show that a visible performance gain is observed on both desktop and mobile web browsers. Further, we show that some of our approaches let the virtual and augmented reality applications utilize parallelism in a way that allows them to handle more complex scenarios than what is available with state-of-the-art solutions in production devices. Since Three.js and webgl are not only used by WebVR/AR/MR applications but also by a large number of games and graphic rendering applications, our improvements can impact all applications that utilize these frameworks. Experiments on both desktop and mobile browsers affirm our hypothesis. We also designed a new set of benchmarks and techniques to measure WebVR performance, since there is no pre-existing benchmark suite that could be used to capture VR performance on browsers.

Acknowledgement

First and foremost, I would like to thank my advisor Dr. Vivek Sarkar for his constant guidance and suggestion that made this work possible. Dr. Sarkar has been very patient, encouraging and supporting towards me at times when things did not seem to go well with my research. Thank you for all your support and time whenever I needed them.

Thank you, Dr. Ray Simar and Dr. Lin Zhong, for agreeing to be a part of my thesis committee. We are very fortunate to have professors like you in our department.

I would also like to extend my thanks and heartfelt gratitude towards the Mozilla Research Mixed Reality Team, especially my manager Dr. Lars Bergstrom, my mentor Fernando Serrano García, and Dr. Blair MacIntyre, who also heads the Augmented Environments Lab at Georgia Institute of Technology, for letting me pick their brains, guiding me and giving me an opportunity to work with the Mozilla Mixed Reality research team. The feedback and suggestions from members of the Mixed Reality team, including Kip, Trevor, Casey, Diego, and Emily, have been invaluable to my thesis work. I am thankful to Havi Hoffman, Jason Weathersby, Michael Ellis from the Mozilla Developer Outreach team and for their support throughout my journey with Mozilla and also Dietrich Ayala from the same team, who was one of the very first people who encouraged my code contribution towards Mozilla.

I would like to thank all the members of our "Habanero Extreme Scale Software Research" group for all their help and suggestions throughout the journey. Thank you Bumjin, Jaeho, Daniel, Rima, Sushovan, my wonderful office-mates, for your stimulating discussions and encouragements when I was at my lowest. I wish to thank all my friends at Rice, who have always been there for me when I needed them,

especially Arkabandhu, Rohan, Sourav, Hamim, Ankush, Arghya, Bitan, Sriparna, Sagnak, Sagnik for all the fun-filled meetups and Ankan, Avisha, Mitropam for bearing with me at times when I was at my most *unbearable* stage.

Finally, I would like to take the opportunity to thank my parents Malay and Manjari for their never-ending support and belief in me.

Contents

Illustrations

Chapter 1

Introduction

The concept of virtual reality is not a new one. Ivan Sutherland [5] introduced the idea of a head-mounted display which would work as a display capable of putting the user into a virtual space. The fundamental idea behind the three-dimensional display was to create the illusion that the user is observing a three-dimensional object. The image projected by the display had to synchronize with what the user would have seen in real life with his head movements taken into consideration. They built special-purpose digital matrix multiplier and clipping divider hardware to compute the appropriate perspective image dynamically because no available general-purpose computer was fast enough to provide a flicker-free dynamic picture. These experiments encouraged further exploration of Virtual Reality. Sutherland's paper [5] noted the hardware limitations on performance, especially for rendering stereoscopic images, which in turn ignited a lot of follow-on work. A technical report by Cohen et al. [6] proposed mathematical solutions to a number of problems which directly affect, and later became crucial for, building real-time scene rendering and graphics. In the mid-1970's, scientists at Bell Laboratories created a dynamically reconfigurable keyboard in which computer-generated labels are optically superimposed onto a two-dimensional array of pushbuttons that rapidly changed as the task required [7]. Later, this concept was extended at MIT to a limited three-dimensional virtual workspace in which a user could manipulate 3D graphical objects that are spatially correspondent with hand positions [8]. A key concept in each of these projects was the use of interactive

computer graphics to create a virtual workstation environment.

At a similar time-frame in the 1970's, researchers at MIT developed a prototype room-sized human-machine interface environment with wall-sized display and stereophonic sound surrounding the user [9]. The emphasis was on creating a highly visual and personalized interface environment where information access and manipulation is facilitated by discrete spatial organization and natural interaction [10]. Researchers at the University of Utah developed a binocular head-mounted display that superimposes computer-generated virtual objects into the real environment of the user [5]. Later at NASA Ames Research Center, Fisher et al. [11] envisioned the first virtual environment display system which closely resembles what we have today as a head-mounted display.

Almost all the prior work exhibited heavy investment towards special-purpose hardware for head-mounted display improvements. The advances in commodity hardware has finally caught up with the computational requirements of this problem domain. A mandatory requirement for a virtual reality scene to be properly immersive is to deliver a consistent throughout of 90 frames per second (fps) when viewing the scene [12] [13]. This creates some strict criteria for the devices that aim to run Virtual Reality content. The devices capable of running virtual reality range from capable and costly devices like HTC Vive [14] and Oculus Rift HMD [15] to mobile solutions like Daydream from Google and GearVR from Samsung. All of them have different platform and operating system support. DayDream and GearVR work exclusively on Android, and HTC Vive and Rift utilize Windows systems. Google Daydream also supports Web Virtual Reality in these headsets.

On another front, we have advances in the development of web augmented reality and mixed reality experiences being built. These are built sometimes using very

specialized hardware like Microsoft Hololens [16]. Chen et al. at Microsoft have also explored the case of using these Augmented Reality experiences to create connected experiences [17]. The nature of these experiences require a real-time rendering capability. With Apple ARKit [18] and Google ARCore [19] available in mobile devices, many of these Augmented and Mixed reality experiences have started to be used in similar frameworks such as Virtual Reality.

1.1 Motivation

Performance is and always has been the key to creating an immersive virtual reality experience. With Augmented and Mixed reality applications using similar technology stacks to create their respective experiences, it is more important than ever to create an optimized pipeline which can render graphics intensive applications easily even on mobile devices.

But this is not just about rendering it on the device. These applications are now being built on the web using JavaScript to program behavior and WebGL to render the animations. Having WebGL support within browsers has enabled web applications to access the graphical processing unit of the machines [20]. As hardware evolved, animations went from two-dimensional renderings into the realms of the third dimension. Although two-dimensional rendering could be done through the use of the Canvas application programming interface (API), it wasn't until the Web Graphics Library (WebGL) debuted that the graphics hardware was actually exposed to a lower level within the context of the browser. This API has now been implemented by every major browser vendor, making it possible to create a fully three-dimensional game without relying on a plug-in [21].

Even though WebGl is extensively used in animations, game development, and VR

scene creation, it is still fairly hard to learn and use. That prompted the creation of frameworks that abstracted away some portions of the API and provided developers with an easy way to access the functionalities [22]. One such framework is Three.js [23], which has received added help from many contributors, and its user base has grown to a large size. Three.js provides several different draw modes and can fall back to the 2D rendering context, if WebGL is not supported. Three.js is a well-designed library and fairly intuitive to use. Default settings reduce the amount of initial or "boilerplate" work needed. Default settings can be overridden as parameters passed in upon object construction or by calling the appropriate object methods afterwards. Since Three.js is easy to learn and use [24], it was a natural choice for the developers of Web Virtual Reality applications. However, Three.js was never designed with virtual reality's real-time requirements in mind. This creates performance drawbacks when creating virtual reality scenes.

Our approach in this thesis will be to identify the specific shortcomings of Three.js and to optimize them using our methods. Most of the shortcomings stem from the way head-mounted displays handle data, how the scenes render and understand the data, and how quickly these functionalities can be performed. For a scene to feel immersive at a level that does not induce any side-effects for the user like nausea, we will need a persistent and constant throughput of 90 frames per second on the display. A lower throughput will cause the user to experience nausea, and to start noticing anomalies in the created scene. The computation is also dependent on the browser executing JavaScript, which is sequential in nature. We will explore some parallelization techniques to speed up the application execution. In addition, an animated virtual scene might have animated models in it, which are generally texture files of considerable sizes. One challenge we will have to address is how to load

and render them in near real-time while the user is traversing a scene. It becomes a challenge because the GPU upload function for textures becomes a blocking operation for browsers. Addressing these challenges would lead to a much better experience and performance for WebVR applications, especially on mobile devices where processing power is at a premium but rendering expectations are still the same as for desktop systems.

1.2 The Scope of the Thesis

The contributions of this thesis will be demonstrated in the context of the WebGL 1.0 library, Three.js, and Aframe which is based on Three.js. Exploring other frameworks is beyond the scope of this thesis. The application benchmarks in this thesis will be evaluated on a Daydream capable mobile device, primarily a Pixel XL, and on an Oculus Rift headset. In addition, our evaluation will use some slight modification to Firefox DevTools and Chrome DevTools.

1.3 Previous Work

The majority of the previous work in the field has been to improve the performance and tracking capabilities of head-mounted displays. Bowman et al. [1] argued that immersive virtual reality works well, so long as the crucial components shown in Figure 1.1 are optimized for high performance so as to produce a compelling and immersive virtual reality experience. Parameters that impact performance include the size of the visual field (in degrees of visual angle) that can be viewed instantaneously, or Field Of View (FOV), field of regard (FOR) – the total size of the visual field (in degrees of visual angle) surrounding the user, display size, display resolution, stereoscopy – the display of different images to each eye to provide an additional depth

cue, head-based rendering – the display of images based on the physical position and orientation of the user's head (produced by head tracking), realism of lighting, frame rate, and refresh rate.

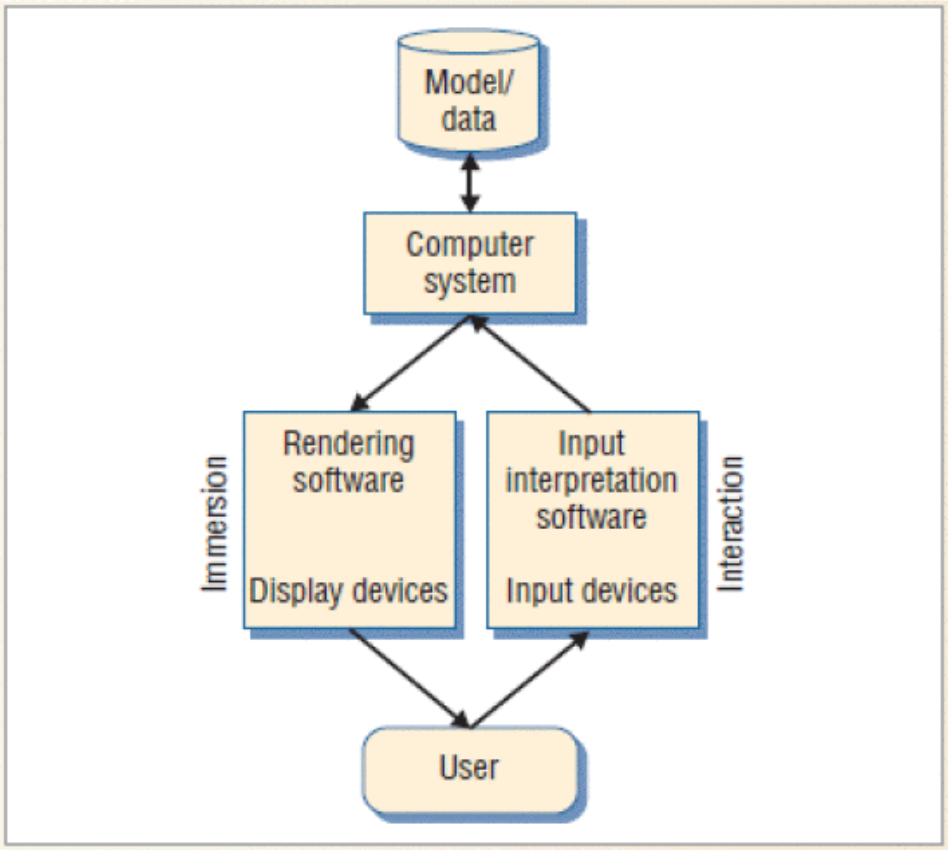


Figure 1.1 : The human-VE interaction loop, including parts of the loop that Bowman et al. [1] considered to be important components for immersion. Figure from [1]

Each of these problems have been worked on using different approaches. Schwartz et al. [21] proposed a WebGL framework through which they were able to generate highly accurate virtual persona with complex reflectance behavior. It also included visual hints about manufacturing techniques and signs of wear. They used a compression mechanism called Singular Value Decomposition to make it easier to stream. By

streaming the individual components obtained by the Singular Value Decomposition (SVD) based compression of the Bidirectional texture function (BTF) along with a wavelet-based image compression, they managed to present high-quality previews of the Bidirectional texture function (BTF) after a delay of only a couple of seconds. They also use a progressive approach to lazily load the remaining data. Watson et al. [25] studied how system responsiveness varies for Virtual Environments and the correlation between better responsiveness and having a better frame-rate. Lee et al.[26] found out in their Walking Interaction based Immersive Virtual Reality experience that a minimum of 75 FPS must be maintained to avoid motion sickness for VR scenes in their case. Different approaches have been taken to combat VR induced motion sickness if FPS cannot be improved. Fernandes et al. [27] utilized a way of limiting the field of view to decrease VR motion sickness, but that also reduces the sensory perception. Rendering stereoscopic images efficiently using WebGL is an open problem in the WebVR specifications [28]. Another aspect of the problem is loading large animated models and textures efficiently in WebGL in browser so as not to block the offloading of the textures to the GPU and ensure a responsive system for WebVR interactions. As David Hrachov noted in his thesis [29], loading a large texture in different WebGL frameworks produces very inconsistent results across the board. As a common theme, it is clear that increasing the size of the texture model leads to significant increases in the time needed to load and process the model. We see one potential solution by Kang et al. [30], where they try to lazily load a big texture using tile loading techniques. Another approach for dealing with how WebGL renders scenes is by trying to parallelize the calls using the Web Worker framework. Since web worker does not have access to the Document Object Model (DOM) directly, this method relies instead on utilizing message passing [31]. Though this is not the

optimal solution, it still gives us some performance boost. By using WebGL Worker [31] on BananaBread*, a javascript benchmark game created by Mozilla, we still get notable performance boost. Figure 1.2 shows the frame per second render output for increasing number of bots in a scene [32].

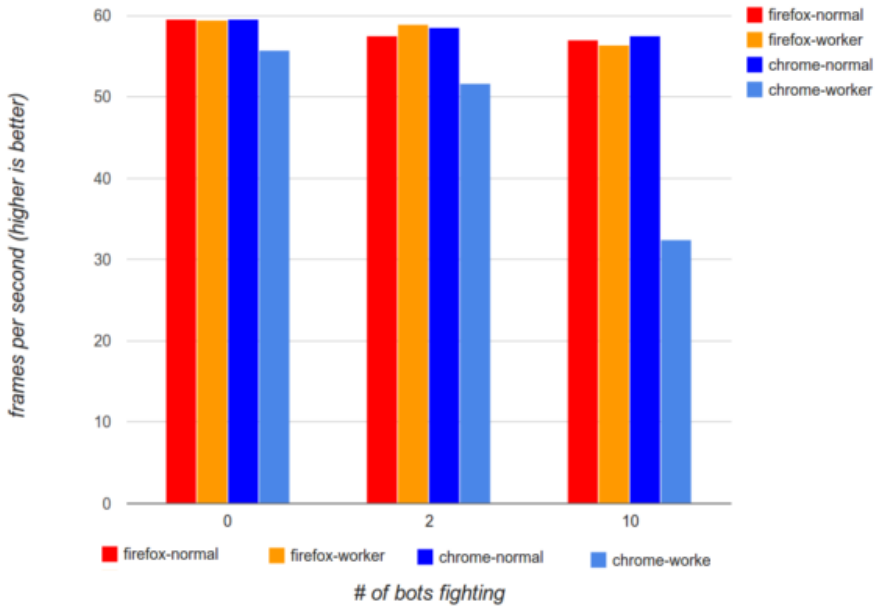


Figure 1.2 : *The chart shows frames per second (higher numbers are better) on the BananaBread demo using Firefox without webworker, Firefox with webworker , Google Chrome without webworker , Google Chrome with web worker*

But this still doesn't help in loading if the texture is big, as, in such a case, it will not block the loading but will also not show the user anything while inside the virtual scene, and will show a blank scene, in fact. Also, the proxying method is not suited for mobile browsers. Chrome's slowdown for worker arises due to the overhead of creating more proxies for the worker.

*BananaBread Benchmark: https://kripken.github.io/misc-js-benchmarks/banana/index.html

Moreover, in virtual reality applications often the aim is to provide real-time objects that can render at a high refresh rate.

1.4 Thesis Overview

The rest of the thesis follows the following format

- Chapter 2 begins with a little bit of background on Web Virtual Reality and Web Mixed Reality, the existing frameworks and their limitations
- Chapter 3 talks about specific bottlenecks and discusses one of them – loading textures and objects into VR scene. The limitations it poses today, the blocker issues and our approach on solving it exploiting parallelism granted by Web Workers is taken into consideration as well
- Chapter 4 illustrates other performance bottlenecks including the de-duplication of computation in stereoscopic image rendering for both eyes and the lack of incremental image loading in VR scene, ending with our approach to both of the problems
- Chapter 5 talks about our experimental setup and validates our approaches
- Chapter 6 touches upon the future improvements that can be done and related future works

Chapter 2

Background

2.1 Web Virtual Reality

Virtual Reality is a technology which allows users to manipulate, explore and immerse in computer generated interactive environments in real-time citesherman2002understanding. It has often been defined as an "interactive immersive experience generated by a computer" [33], and it facilitates a natural, intuitive way to interact with a computer. The concept of VR is based on the assumptions that we can fool our brains by mimicking the external stimuli we normally receive through our senses by utilizing electronic external stimuli, *"If a computer application can send the same external stimuli that the brain can interpret, then the simulated reality is potentially indistinguishable from reality"* [34].

When we navigate the world wide web, we mostly do it in a two-dimensional context. We click through pages and create bookmarks and tap as if we are handling physical books. Not all digital web content is meant to be experienced as flat two-dimensional pages, and lose impact or important informations due to the 3D-to-2D conversion. The Oculus Rift is a head-mounted device that enables a user to interact with 3D virtual environments in a natural way, and is for this reason suitable for experiencing virtual reality contents inside a suitable web browser.

At the heart of any Virtual Reality system, we have a computer-generated model that generally utilizes three-dimensional modeling techniques, similar to what a

CAD/CAM software does. All the geometry present in the model must be represented as polygons. This helps in the faster rendering time needed for real-time graphics generation. If any object is created using Conservative Solid Geometry, then it is first required to be converted into a boundary representation [35]. If the boundaries are curved surfaces then a polygon estimation can be derived using a tessellation algorithm * that replaces the surface by a mesh of polygons. To view and to do meaningful interactions with this geometry, the system needs information describing the appearances of the object like color, texture, characteristics, reflection, lighting environment, interaction, possible animation, sound and behavior functionality. This can be referred as the "virtual model". Once the virtual model is created, it can be used with any VR system. Different data formats and the absence of standards make the reuse of these assets a painful proposition. However a potential standardized solution is to use web-based VR.

The concept first started in 1994 by Mark Pesce and Tony Parisi as Virtual Reality Markup Language or VRML [36]. This took the concept of the Placeholder Virtual Reality Project [37] which allowed interaction with others, seen as avatars, and teleportation to worlds outside of the originating machines to the open web. VRML marked the beginning of WebVR, even though it did not take off due to the hardware limitations of creating Virtual Reality experiences. This all changed with the release of the Web Graphics Library (WebGL) [20]. WebGL evolved out of the Canvas 3D experiments started by Vladimir Vukićević at Mozilla. Vukićević first demonstrated a Canvas 3D prototype in 2006. By the end of 2007, both Mozilla [38] and Opera [39] had made their own separate implementations. WebGL opened the path to creating high-performance graphics application on the web. WebGL leverages the power of

*Polygon Estimation: https://gyires.inf.unideb.hu/KMITT/a52/ch10s03.html

OpenGL [40] to present accelerated 3D graphics on a webpage.

But as Leung et al. [41] argued that even though WebGL is a very powerful API which provides JavaScript bindings to OpenGL ES functions, it still is very low-level and requires very good understanding of 3D geometry and mathematics to create compelling applications. During 2014 -2015, over the course of two years, the Mozilla Virtual Reality team explored various approaches to presenting and navigating the web in virtual reality. Multiple functional prototypes of VR browsers were created with different technologies and design approaches, each enabling users to browse the web from inside a headset, moving from site-to-site or scene-to-scene seamlessly with new interfaces and input devices. The first of those prototypes, viz. "Hiro" [42] Figure 2.1 was created and demonstrated in the Game Developers Conference (GDG). Hiro solely used WebGL to render stereoscopic images. By June 2015, another prototype, dubbed "Horizon", was released which combined WebGL with Cascading Style Sheet (CSS) VR, and this rendered Document Object Model (DOM) elements in stereoscopic form using CSS 3D transforms. This was the stepping stone to creating a high-level WebVR framework.

2.1.1 Three.js

Three.js [23] was a library initially built to render DOM, SVG and Canvas. Canvas rendering allowed us to do 3D animations on the web by just using JavaScript so that the library could handle the rest [43]. Later, the WebGL renderer was added to the library. This vastly improved the creation of Web-based 3D content creation utilizing WebGL without using low-level API's and reduced the barrier of entry for Web Developers. Three.js laid out the foundation for creating A-Frame. Three.js is primarily based on WebGL 1.0.

Figure 2.1 : *This is how the first WebVR-enabled view looked like in the "Hiro" browser prototype. This is a stereoscopic image and, when seen through Head Mounted Displays or WebVR capable browsers in mobile devices, it will show a 3D immersive view. From [2]*

2.1.2 A-Frame

A-Frame[44] is an open-source web framework built on top of Three.js to rapidly prototype virtual reality applications. Developers can create 3D and WebVR scenes using HTML in A-Frame by using its entity component system. The benefit it provides over traditional WebGL or Three.js is gained by giving a way to control the objects and models from HTML. That makes creating virtual reality content and models much easier and faster, compared to creating directly in WebGL. Aframe is an open-source Domain Specific Language (DSL) for generating in-browser VR content. HTML tags are used to declaratively place built-in primitive artifacts in a virtual 3D space, and HTML attributes add different artifact properties, like color,

size, location, and animation. Under the hood, A-Frame is a Three.js framework that brings the entity-component-system (ECS) pattern to the DOM. A-Frame is built as an abstraction layer on top of Three.js and is extensible enough to do just about anything that Three.js can do [45].

Entity Component System (ECS) is a pattern commonly used in game development that favors composability over inheritance. Since A-Frame aims to bring highly interactive 3D experiences to the Web, it adopts existing patterns from the game industry. In ECS, every object in the scene is an entity, which is a general-purpose container that, by itself, does nothing. Components are reusable modules that are then plugged into an entity in order to attach appearance, behavior, and/or functionality.

The ecosystem also benefited a lot from 3D model creation utilities like sketchfab and Google Poly which act as a repository for 3D models. But efficiently loading massive models into real-time WebVR scenes is also part of the challenge that we face and need to solve.

2.2 Web Mixed Reality

The concept of Augmented Reality (AR) is almost as old as Virtual Reality. Since Ivan Sutherland showed his ultimate display [46], the idea has captivated researchers around the world to create hybrid immersive displays. In virtual reality, the user is completely immersed in a synthetic world. While immersed, the user cannot see the real world around him. S/he only interacts with the virtual objects created for her/him inside the scene. In contrast, Augmented Reality allows the user to see the real world with virtual objects superimposed on top of them. AR supplements the real world rather than replacing it. And hence it is more important to have a single

connected platform for creating AR experiences.

The idea of single AR environment has been proposed quite a few times over the last few decades [47], [48], [49], [50], [51] and is the motivation behind Augmented Reality web browsers, which are web browsers designed specifically to render custom Augmented Reality scenes while having the benefit of accessing and rendering web pages. Many of these browsers have achieved a good position as augmented reality viewing devices, like the Argon [52] AR browser from Macintyre et al. Argon utilizes an extension on KML (the markup language used in Google Maps and earth) called KARML to support AR functions on the browser. KARML allows the developers to encode the extra meta information to create AR applications on top of standard web technologies.

However, most of these AR browsers lack the benefit of having a unified platform. Every user needs to have that specific browser installed, and the developers need to use their platform or special techniques to create Augmented Reality applications. Which raises the question: why can we not enable WebAR on browsers as has been done for WebVR. To explore on that idea, first we need to define WebAR. Ron Azuma clarified in his survey paper "A survey of augmented reality" [3] what Augmented Reality is, as we see in Figure 2.2 and 2.3.

With the perspective of Figure 2.2 and 2.3 can define Virtual Reality as a subset of Augmented Reality, as shown in Figure 2.4.

And if we reiterate this concept further enough, we get Figure 2.5, which essentially is possible if we know what external informations are available to us and how well we can align real-world video feeds with virtual objects.

Essentially, we can use the same pipeline of creating virtual objects from WebVR and use camera feeds to get external video to recreate AR. However, Augmented

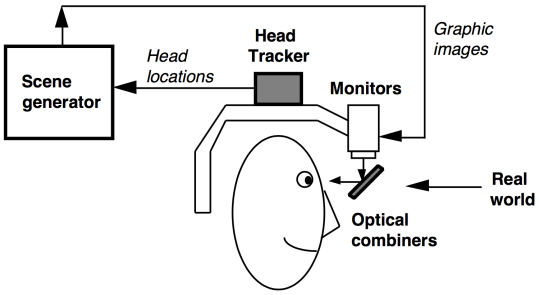


Figure 2.2 : Virtual Reality

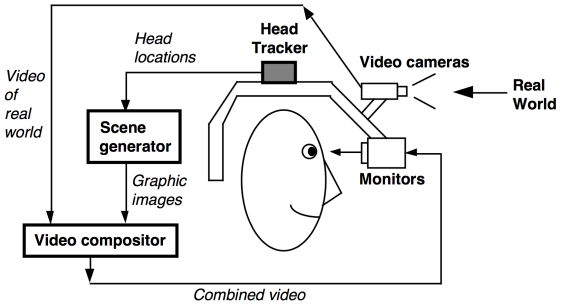


Figure 2.3 : Mixed Augmented Reality
Ron Azuma in "A survey of augmented reality" [3]

Reality is more than just overlaying images; positional tracking is also very important to create meaningful experiences. ARkit [18] and ARCore [19] provide us those capabilities in mobile devices today, making it possible to create proper Mixed Reality experiences in the web browser. The WebXR [53] API [†] proposed by Mozilla enables us to do that today in both iOS and Android applications. We can access the APIs and tracking capabilities right from the browser and build experiences today like Figure 2.6, Figure 2.6 and Figure 2.8.

2.3 *Case Study:* Augmented Reality Demonstration in Web

Here we will be talking about two Web Augmented reality applications created using the WebXR [53] specification capable of running both in Android and iOS inside a supported browser. They expose and demonstrate some of the capabilities of Web Augmented Reality and use cases where low latency, high performance web apps are necessary.

Both of these two demo applications are running from a experimental We-

[†]WebXR Experimental API: https://github.com/mozilla/webxr-api

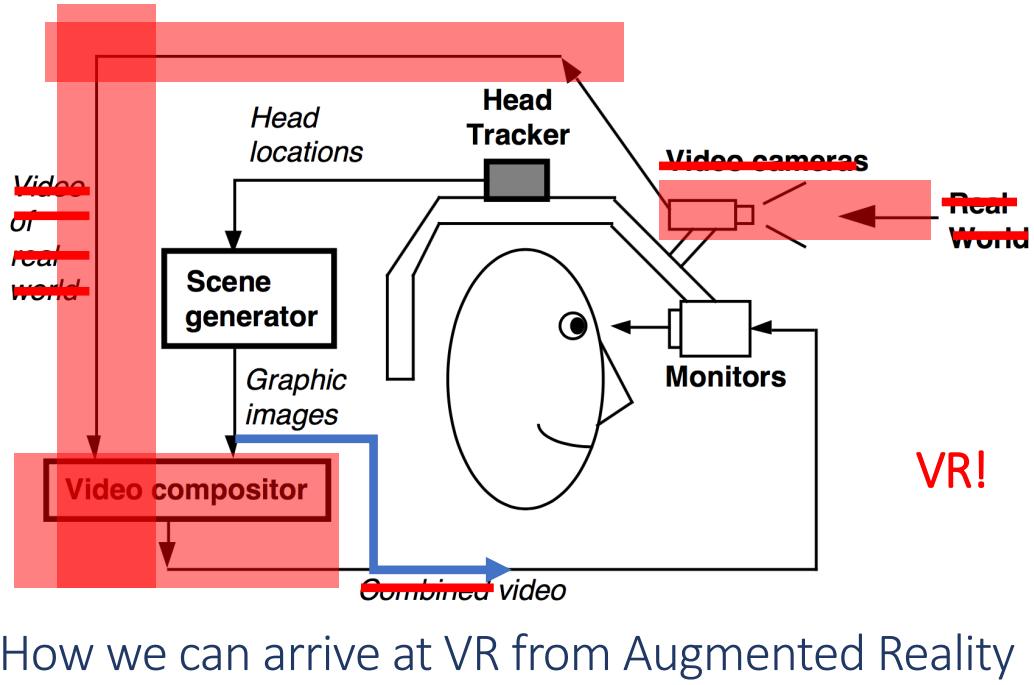


Figure 2.4 : Arriving at Virtual Reality from Augmented Reality defined by Azuma et al. [3]

bARonARCore browser [‡]. As can be seen from Figure 2.6, Figure 2.7 and Figure 2.8, object tracking is satisfactory despite challenging lighting conditions, making us believe that Web Mixed Reality is realizable in today's mobile phones. However since Augmented Reality is even more dependent upon real time rendering, reduction of latency, and rendering artifacts play a crucial role in how fluid they will be. Fortunately since WebXR api builds on top of some of the same technologies, some of the optimization techniques we discuss in this thesis benefits the WebXR specification too, as long as it utilizes Aframe [44], i. e.

[‡]WebARonARCore Repo: https://github.com/google-ar/WebARonARCore

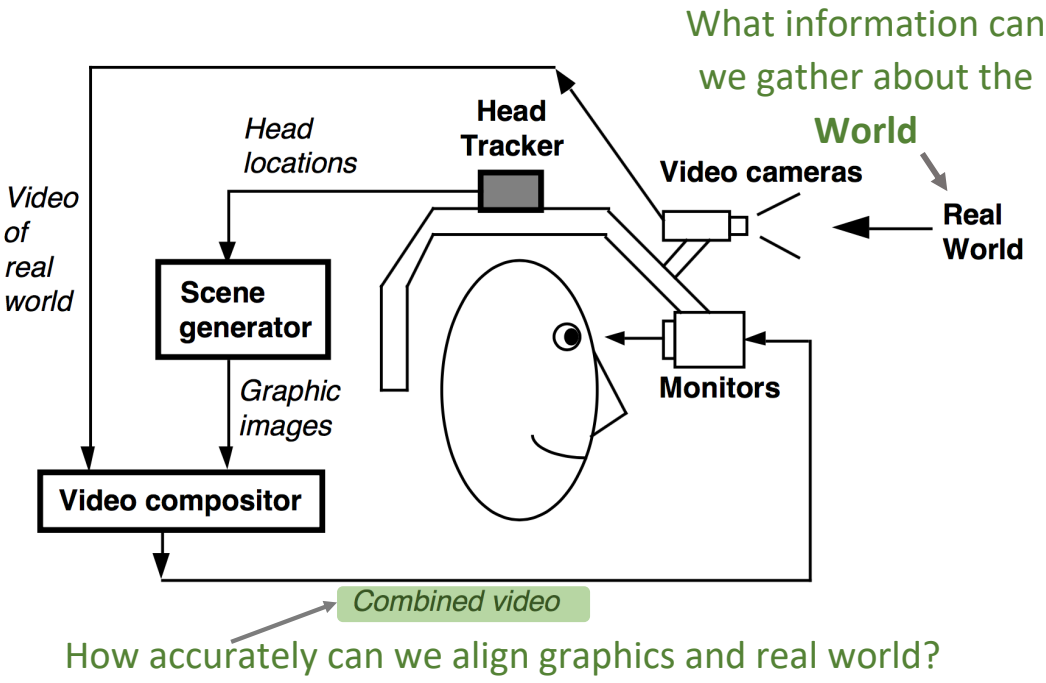


Figure 2.5 : Aligning Virtual Objects with Real World objects to create Mixed Reality from Figure 2.3

2.3.1 A-Painter in WebXR

A-Painter [54] is a web application which replicates a VR painting Application from Google called Tilt Brush [§]. Tilt Brush is a native application available to be used with HTC Vive and Oculus Rift through their respective stores in Windows only. A-Painter [¶] aims to recreate the same experience completely on the web with web technologies running just from a browser utilizing WebVR and Aframe at the same time being open source.

[§]Tilt Brush: https://www.tiltbrush.com/

[¶]A-Painter: https://aframe.io/a-painter/

This application is a port built on top of A-Painter which eliminates the need of room-scale VR devices like Vive to utilize the Visual Inertial Odometry available to us through ARkit [18] and ARCore [19] to create 6 Degree of Freedom for the user where they can create paintings in the real world with positional tracking. This is still an unreleased app by Mozilla Mixed Reality team, actively being worked on for performance improvements.

2.3.2 AR Furniture Suggestion App

This application was demonstrated as an entry to the Hackathon organized by the University Of Houston, called CodeRed. This is also a Web Application utilizing WebXR api's. This let the user define a room by drawing a perimeter and then place different furnitures throughout said predefined room. It can create those virtual furnitures, walls, and doors to give you a feel of how it will be after you are done furnishing a room in real life. The positional tracking ensures that the placed furnitures retain their position even when the user is moving around with his mobile. That allows the user to walk towards different furnitures to see how they look from different angles, to change their placements, textures etc. It uses Archilogic [‖] to query for different furniture models. An overview of how the application looks like while being used is available as a demo here [**] and here [††].

The application can be accessed from the webpage[*] and more details are available in the hackathon webpage[†].

[‖] Archilogic API: https://docs.archilogic.com/en/api/reference/requestmodel

[**] DecorateAR Hands-on: https://www.youtube.com/watch?v=nQstlsZOymg

[††] DecorateAR Demo: https://www.youtube.com/watch?v=OVo68GKXHlc

[*] DecorateAr: https://decoretear.org/

[†] CodeRed DecorateAr:https://cdred2017demork.surge.sh/

These demo applications were created for testing the AR aspect and bottlenecks of the browser. Figure 2.8 shows a Augmented Reality furniture suggestion app initially created for the University of Houston Hackathon submission [‡], and Figure 2.6 is an AR port of a A-Painter [§] , a WebVR painting application by Mozilla Mixed Reality team, but not yet released to the public due to the ongoing performance improvements of it being worked upon.

Both of these showcase the need for low latency, high quality rendering on the web fro virtual object creation and texture loading in real-time scenarios. We can see, for the augmented reality furniture demo, the time that the user has to wait for while the furnitures are being loaded and rendered. While a portion of that is definitely dependent upon network latency, another portion is also dependent upon loading the furniture models through ObjectLoader and rendering those.

[‡]DecorateAr: https://devpost.com/software/decoratear

[§]A-Painter: https://blog.mozvr.com/a-painter/

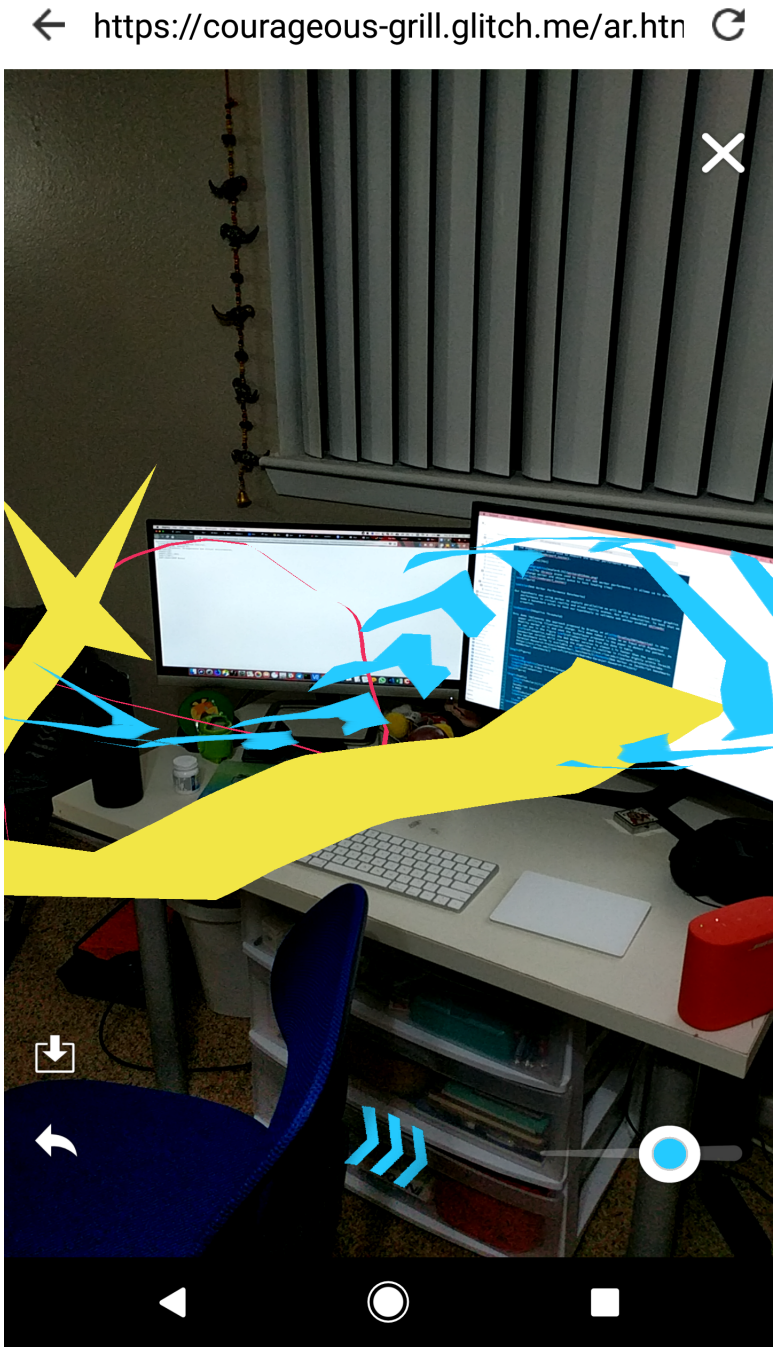


Figure 2.6 : **Web Augmented Reality Painting** *The yellow and the blue marks are actually painting brush strokes which retain their physical location even when you move around with the mobile*

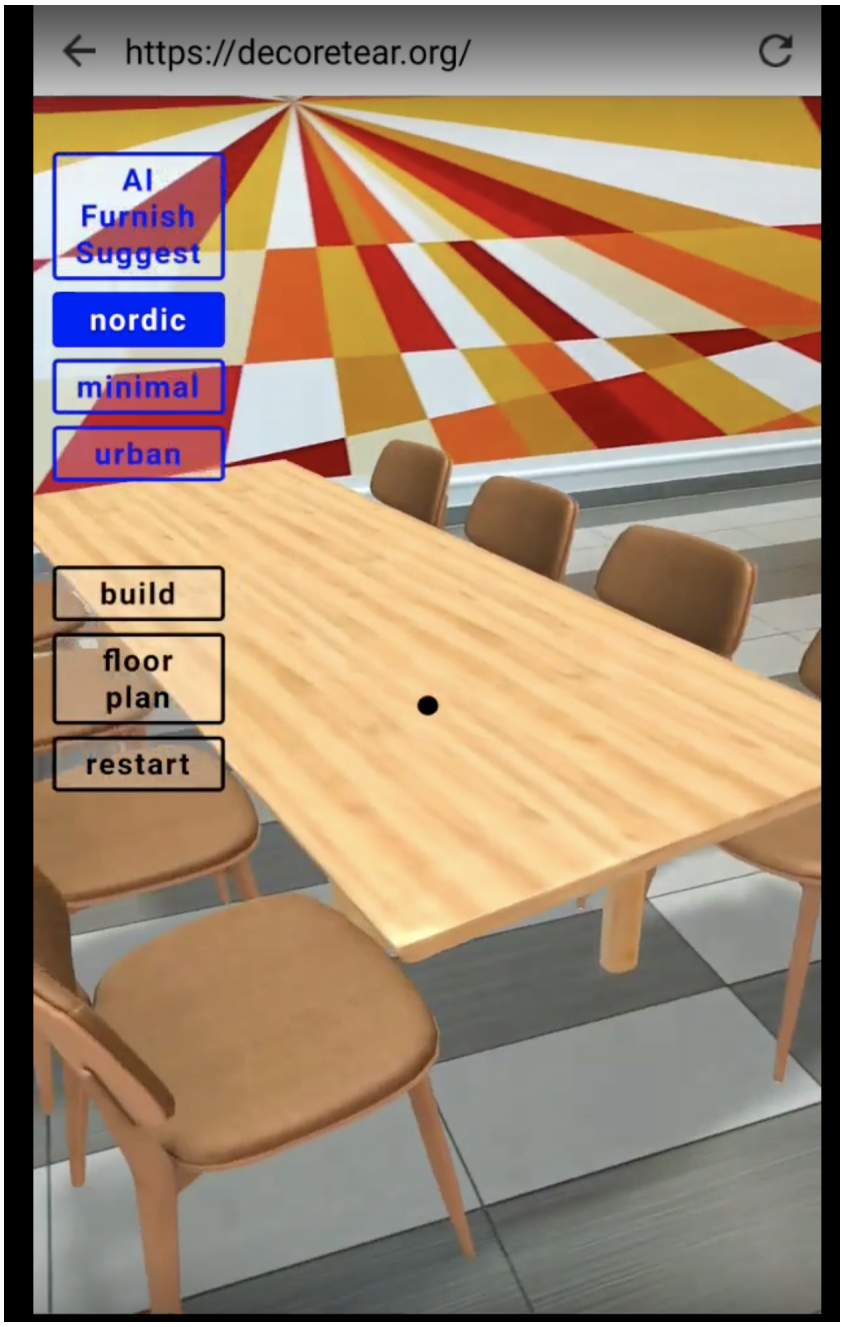
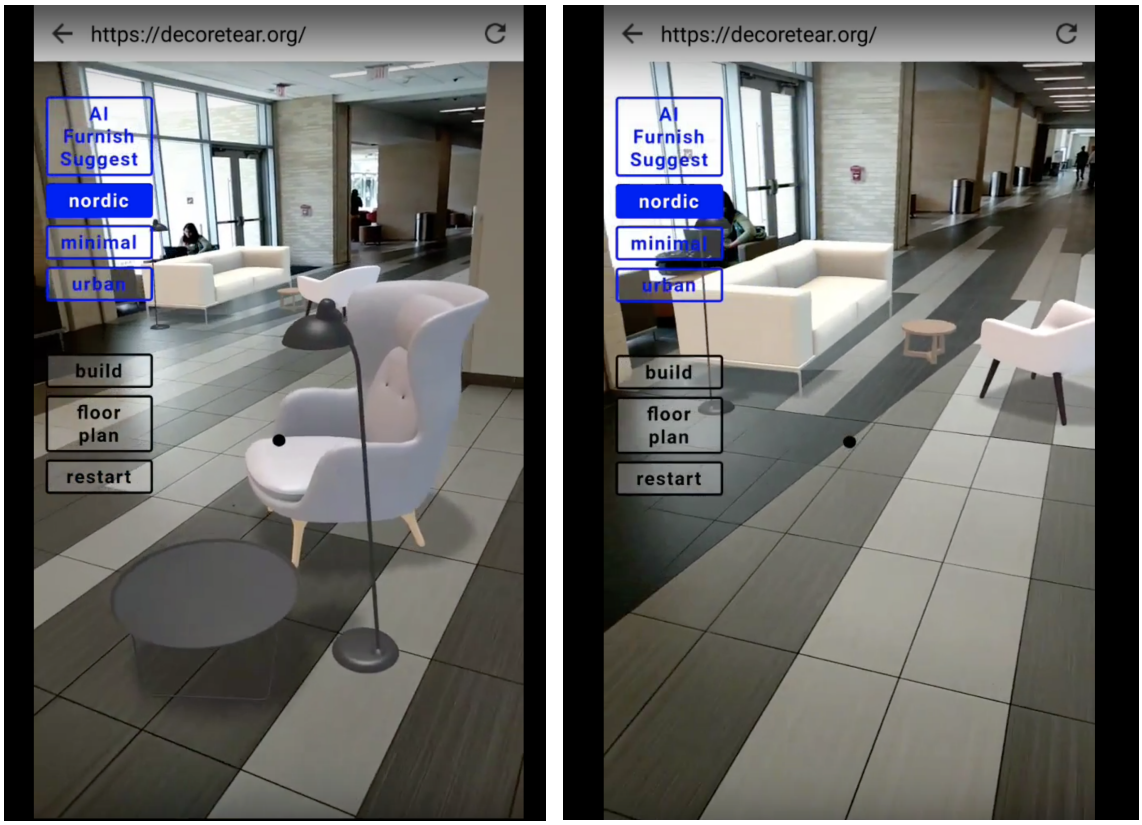


Figure 2.7 : **Augmented Reality Furnitures** *The chair and the table are virtual object. Though they retain heir position in real physical world. The background and the floor is real.*



(a) Augmented Furniture (b) Furnitures Retaining Position

Figure 2.8 : *The Augmented Reality demo with Furnitures was done in a daylit corridor at the University of Houston for a hackathon [4], and the Drawing application was done at a study room lit by fluorescent light while writing this thesis. The white chair, the black standing light, the black table, and the white sofa are all virtual objects anchored to a real-world floor in specific positions and ran from within a browser to consume the WebAR application*

Chapter 3

Optimizing the Object Loader

Threejs was created to provide an easy-to-use, lightweight 3D library. The library includes support for <canvas>, <svg>, CSS3D and WebGL renderer. Since Three.js was not primarily designed with WebVR in mind, there is a lot of room for optimizations specific to WebVR applications, as indicated in the previous chapters. Among the many frameworks available, Three.js at present is the most popular choice when developing Web Virtual Reality applications. Mozilla uses it extensively as a foundation for A-Frame. Hence any improvement in Three.js directly makes its way into Aframe and benefits the rendering path, also making it a very good candidate for our work.

One of the issues that is very critical apart from object creation in a virtual reality scene is loading existing asset and models in the VR view.

3.1 Creating an Efficient Object Loader

The WebVR render path or, in general any non-VR render-path for Three.js has two important elements in the path.

- One is polygon creation which we tackle in our de-duplication efforts in Chapter 4.
- The other is loading complex texture and object models.

Simple object models can range from geometric shapes to complex animated models which are normally parsed by the *ObJLoader* in Three.js. But this operation is blocking in nature and sequential. Hence it becomes a bottleneck when we have to load large object files to parse and to render. It has been a known problem for quite some time [55]. However, in web virtual reality applications, if a user wants to change a scene or to traverse to another scene, they essentially either change the web page or load new objects. In either case, traversing the scene becomes unsuitable if the new scene has a big enough object file that takes time to be parsed and loaded by the object loader. The resulting inactivity forces the browser to kill the non-responsive script, in this case our objectloader, and kick the user out of the VR scene. This behavior is not limited to VR only, but is critical for ensuring proper VR experience on the web. In WebVR, every time there is a delay for the headset in getting the frames from the computer, is in fact the number of times the user will get kicked out of the Vive/Oculus lobby. So if the assets are loaded at runtime when the experience is already presenting in VR, the user will have a sub-par experience by jumping in/out of their VR scene whenever a frame drop occurs. Link traversal in VR relies on the `onvrdisplayactivate` event. It is fired on the window object on page load if the precedent site was presenting content in the headset. To enter the VR mode for the first time, the user is expected to explicitly trigger the VR mode with an action like a mouse click or a keyboard shortcut to prevent sites from taking control of the headset inadvertently. Once VR is engaged, subsequent page transitions can present content in the headset without further user intervention. It's also important to notice that, with Link traversal, this problem is also important at the loading time, because when you enter a new website from a previously presenting WebVR page, the browser will wait for a small period of time before you send the first frame, and if you don't

do it fast enough, the browser will stop presenting and will return to 2D mode, requesting user interaction again. The other aspect of this is when a scene requests to load textures from within the scene. Dynamic loading of texture result in noticeable frame-rate drops across the scene. And the more textures the scene loads, the worse the effect becomes. This directly impacts the performance of Aframe as well, and is an open issue therein [56]. To understand what is going on, we look at a scene with object [57] and load a 13 mb model with 2048x2048 texture. When we look at the performance from Firefox DevTools, the primary blocker here is the texture upload to GPU Figure 3.1.

There are a couple of ways to tackle the problem. One of them is to try to offload the asset parsing out of the main thread to web workers so that it can parse the assets in parallel in a non-blocking way.

- An async-texture system can be added to the system which uses the *ImageBitmapLoader* api to decode images after the user enters Virtual Reality. This helps achieve lower frame drops during the texture upload. Google Chrome already supports *createImageBitmap* in async manner, but that support is not universal among all browser implementations.

- If *createImageBitmap* is used in a separate worker thread, the time wasted in blocking mode should be significantly reduced.

A benchmark using the Khronos Group sample gITF models [58] [59] shows the improvement Figure 3.2.

In both of the charts in Figure 3.2, Green signifies the loading duration (non-blocking), and Red denotes the duration for first render (blocking operation). Results were measured in Chrome.

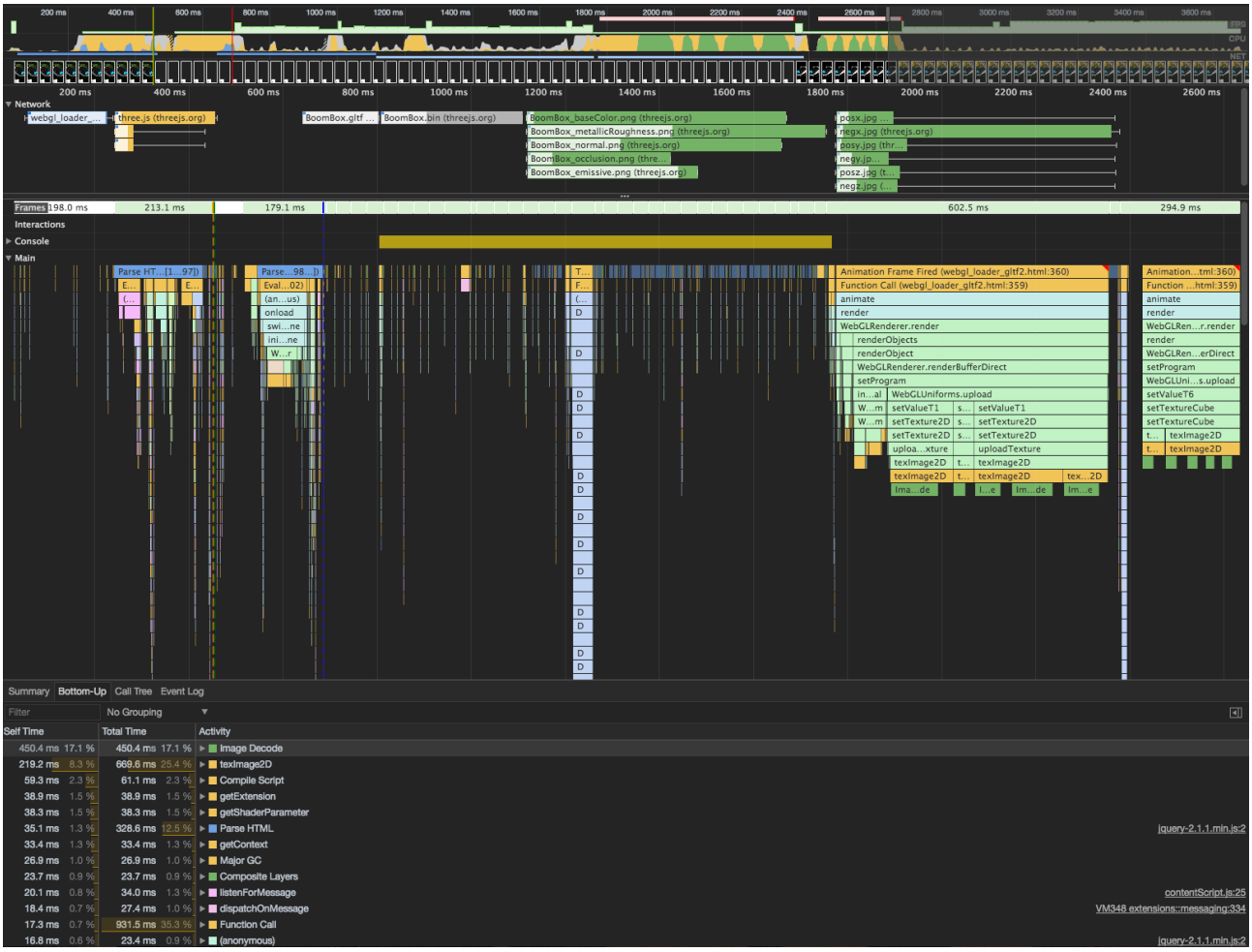


Figure 3.1 : *Here we load a gITF model of size 20mb and use Developer Tools to notice the loading performance impact. If you look at the bottom pane which has the activity name along with the time taken for them to load, you will notice that the blocking behavior here is for offloading the texture to gpu*

Here we see that our approach has introduced a tax on the loading duration in both the cases. This happens due to worker initialization for both of the experiments compared to the previous implementation where we did not need any worker. However, our approach reduces the blocking operation from the previous implemen-

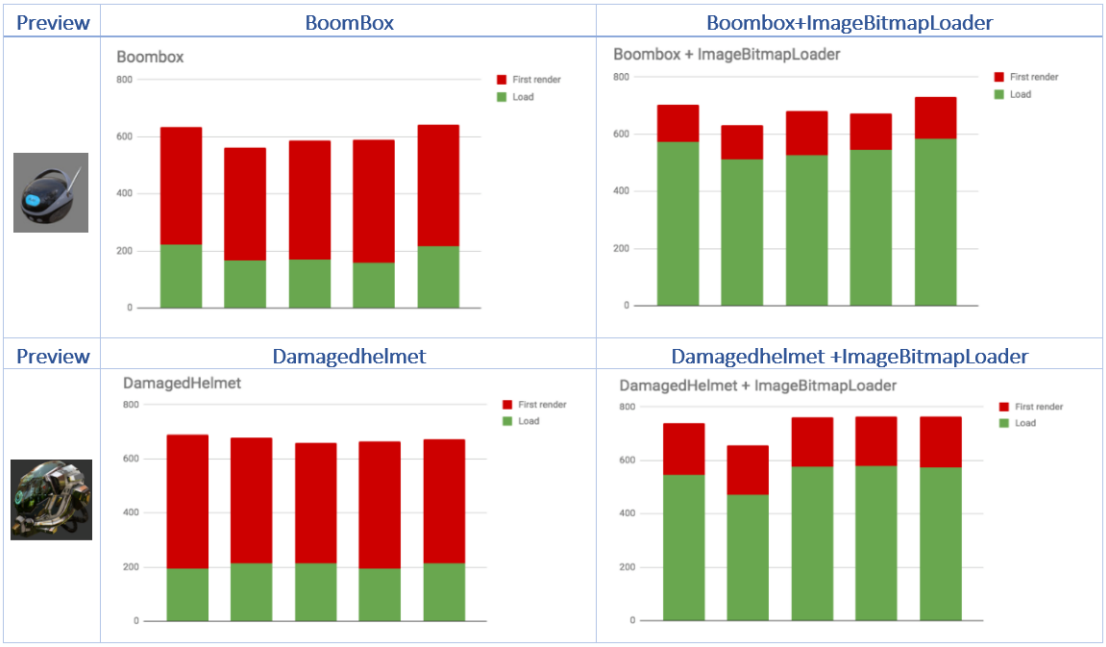


Figure 3.2 : Performance: Load and upload time (ms)

Here we load two standard gITF models from the Khronos sample models by using createImageBitmap in a separate worker to load the image

tation allowing us to load textures in a non-blocking way without making the user wait while loading the scene. The rendering time, even though increased, does not adversely affect the first render time; rather, it makes that faster, allowing us to use workers to keep on loading the remaining texture off the main thread even when the user is inside the scene.

3.1.1 Our Approach

We have two possible ways to approach non-blocking the loading of OBJ files. Oculus has taken one approach in their implementation of Oculus ReactVR waveform OBJloader [60]. Their approach with the loader essentially makes it read lines using

small time-slots to prevent blocking the main thread by explicitly calling SetTimeout [61]. This approach, although ensuring that the main thread is not blocked, introduces some more overhead. The object loading becomes much slower as essentially the loader is parsing some lines in each time slot and then waits for some specific interval. The advantage is that once the parsing is complete, we have the objects ready to use without any additional overhead. But since this creates a forced overhead because of the hard-coded wait time, it does not solve our purpose very well. The premise of ReactVR taking that approach was not to have an optimal loader but to have a way to avoid sudden and unexpected frame outs when new components are loading. For texture loading, one approach is to optionally load the textures incrementally. That still brings up the question of how long the user will be able to wait while meaningfully interacting with the texture. For gltf object parsing, an average delay of 500ms was observed. For native OpenGL as well, the two blocking operations are compiling shader and uploading image data. Our approach consisted of loading the texture and gltf into web worker and parsing them inside worker. That allows us to utilize the parallelism for parsing without blocking the main thread. While we parse the texture on the worker, objects are created back on the main thread and include an incremental loading on the renderer.

Listing 3.1 Object Loader Worker initialization

```
WorkerRunnerRefImpl.prototype.run = function ( payload ) {
  if ( payload.cmd === 'run' ) {
    console.log( 'WorkerRunner: Starting Run...' );
    var callbacks = {
      callbackBuilder: function ( payload ) {
        self.postMessage( payload );
      },
      callbackProgress: function ( message ) {
        console.log( 'WorkerRunner: progress: ' + message );
      }
    };
    // Parser is expected to be named as such
    var parser = new Parser();
    this.applyProperties( parser, payload.params );
    this.applyProperties( parser, payload.materials );
    this.applyProperties( parser, callbacks );    parser.parse(
        payload.buffers.input );
    console.log( 'WorkerRunner: Run complete!' );
    callbacks.callbackBuilder( {
      cmd: 'complete',
      msg: 'WorkerRunner completed run.'
    } );
  } else {
    console.error( 'WorkerRunner: Received unknown command: ' +
        payload.cmd );
  }
};
```

In Listing 3.1 the web workers are able to configure any parser inside the worker via parameters received by a message. *WorkerSupport* provides a reference worker runner [github citation] that will be replaced by personal code if needed. The worker will create and run the parser in the *run* method of the *WorkerRunnerRefImpl*. Parser is available under worker scope. Message from *OBJLoader2.parseAsync* works as shown in Listing 3.2

Listing 3.2 Passing Message from worker

```
this.workerSupport.run(
  {
    cmd: 'run',
    params: {
      debug: this.debug,
      materialPerSmoothingGroup: this.materialPerSmoothingGroup
    },
    materials: {
      materialNames: this.materialNames
    },
    buffers: {
      input: content
    }
  },
  [ content.buffer ]
);
```

The message object is loader-dependent, but the configuration of the parser in the worker is generic. This provided the foundation of our work to implement worker support for generic parsers and object loaders. The goal was to

- Separate the asynchronous mesh provision of the worker from OBJLoader. This will allow us to exploit task parallelism in JavaScript in the context of object loading.
- Provide a way to load large textures in an incremental and efficient on-demand basis to achieve consistent frame-rates within the application
- Provide a way to handle both object and texture files loading without blocking the gpu loading pipeline

Object Loader with Worker

The new Object Loader consists of three logical blocks. Two of them are private and one public.

- *OBJLoader2*(public): This is the only class the developer will interact with. It is used for setting up any scene and for loading data from any given file. It will also forward the data to the parser.
- *Parser*(private): It is used by *OBJLoader2* and the web worker enabled *WWOBJLoader2* to parse and transform the data into raw representation.
- *MeshCreator* (private): It builds the meshes that can be incorporated into the scene.

3.1.2 Reason for Separation

The design decision to separate out the three blocks comes from the fact that the loader should be easily accessible. The loader should be easily usable from within a web worker, but each web worker has its own scope. Which ensures that any imported code will have to be reloaded. The aim is to have to enclose the parser with two different wrappers

1. Standard direct usage
2. Embedded within a web worker

As *Parser* is not dependent on any other code piece of the Three.js [23] library, the wrapper code will have to satisfy either of the two functions

- It will directly handle integration like *OBJLoader2* with *MeshCreator* or

- With *WWOBJLoader2* where *WWOBJLoader2* serves as a control interface to the web worker code. This is dynamically created when it initializes.

Here, *WWOBJLoader2* functionally is equivalent to *OBJLoader2* and *MeshCreator*, but the parsing and mesh preparation is done by the web worker thus giving us performance improvement.

3.1.3 Directing The Synchronization

We introduce *WWOBJLoader2Director* to make the use of *WWOBJLoader2* more easier. It can create configurable amount of loaders by getting parameters and using object reflection [62].

3.1.4 Parser Design Choices

- *ArrayBuffer* is used as input for the *OBJLoader2* parser method. If it fails to parse due to legacy code, the legacy parser takes over but is typically around twenty percent slower.
- Parser now supports polygons with more than four vertices.
- All involved classes can now be re-used.
- Support for multi-material is now added. It gets invoked in an on-demand basis.
- Flat smoothing is now supported and enabled by "s 0" or "s off". Multi-Material is created when one object equals both smoothing groups not equal and equal to zero.

3.2 Implementation Overview

The following choice have been made for implementation.

- OBJLoader2 and WWOBJLoader2 are now merged. Asynchronous executions which are supported by worker are now enabled using *parseAsync*, *load* with *useAsync* flag. For batch processing, *run* is used.
- The library now includes a package THREE.LoaderSupport located in `LoaderSupport.js` which has all the common functionalities bundled together
- The Parser can now run independently or inside a worker thread (THREE.LoaderSupport.WorkerSupport handles the building and execution)
- A common mesh builder function now uses the raw results.

3.2.1 OBJLoader2

These are some interesting point of interests:

- `OBJLoader2.parseAsync` now only accepts `arraybuffer` as input. Worker handles the passed buffer
- Face models with plygons consisting of more than four vertices are now support supported
- *setUseIndices* is now used to enable Indexed Rendering.
- Every time OBJLoader2 is used it must now be re-instantiated, but the developer has option to cache the worker by using WorkerSupport, or LoaderDirector is available

- 'v' and 'f' occurrences are now used for new mesh detection. 'o' and 'g' are meta information, and are no longer needed for mesh detection.

3.3 Web Worker Support

- *LoaderSupport* is now used to serialize existing code into strings. It offers utility functions which is used to build web worker.

- Loader which utilizes the web worker must provide an entry point to the function which will have the parser code.

- *WorkerSupport* now is used as a wrapper code for the web worker. It is used for instantiating and for communication within the workers.

  A configuration object is used additionally which configures the parser to act similarly as a synchronous application would have.

3.4 Solution

This final approach 5 solves our initial goals of worker creation with a new version of OBJLoader2 that fuses OBJLoader2 and WWOBJLoader2 into one, plus the proof-of-concept MeshSpray example which is not a loader, but it uses all actors and provides a worker-executable parser function.

The different parts of the solution consists of:

- *Validator*: validator is a set of tools to check for null/undefined values and default value assignment

- *Commons (optional)*: It bundles common functions and parameters and acts as a base class for loaders

- *WorkerSupport + WorkerRunnerRefImpl*: WorkerSuppport handles communication between workers and also acts as a helper function to create workers from existing Parser. WorkerRunnerRefImpl creates the communication with the front-end and configures and executes the parser inside the worker.

- *Builder*: Builder builds one or many THREE.Mesh from one raw set of Arraybuffers, materialGroup descriptions and further parameters.

- *WorkerDirector*: Uses *WorkerSupport* in automating the loaders using queued run/PrepData instruction with configurable amount of workers. Primary function includes parser creation by using reflection.

- *Callbacks (onProgress, onMeshAlter, onLoad) + LoadedMeshUserOverride (helpful)*: Primarily used for automation with PrepData.

3.4.1 OBJLoader2:

We have a number of design advantages over our initial solution. Those can be summed up as follows:

- *parse* allows synchronous loading of *arraybuffer* or string data.

- *parseAsync* allows asynchronous loading of *arraybuffer* or string data.

- Both synchronous and asynchronous loading of an OBJ file is supported now using *load*

- Automated loading of MTL and OBJ files according to instructions is now supported using *run*. It now includes sync/async, mesh-streaming, and callbacks.

- MTL loading using MTLLoader and provision of materials to OBJLoader2 is now realized using *loadMtl*
- Generic WorkerSupport can now configure and run an internal parser which is now fully serialized.
- OBJLoader2 now is automatically re-instantiated every time it is used. A parameter in *run* allows support for external WorkerSupport and cached parser worker. A rebuilding is no longer necessary.
- *setUseIndices* now loads data into indexed BufferedGeometry. This is now fully supported by OBJLoader2. Loading speed has decreased by one third than before, but up to three quarters less vertices are created.

Chapter 4

Web Virtual Reality: Other Optimizations

Three.js initially was built as a 3D library. The improvements that we have discussed till now deals with the overhead of loading large object files or 3D models into a three-dimensional scene. It is immensely useful, and it improved the performance of the library, but it still is generic enough to be considered as an overall improvement of the 3D library. However, since we are now looking at three.js solely from the viewpoint of a WebVR developer. There are a number of VR specific enhancement and optimizations possible to improve our VR scenes. We have already tackled the issue of how we can load heavy texture files in parallel without weighing down the object loader pipeline. Below, we have two more issues in the rendering path, and what we can do to optimize the render path of Three.Js affecting the rendering time in webvr scenes.

4.1 Multiple Viewpoints in Camera

For Web Virtual Reality to render images in both eyes, we now use "VREffect [63]" for stereo rendering. What VREfffect essentially does is to create a VR scene for each eye. When a VR capable headset or head mounted display is connected to a computer, the Three.js rendering pipeline gets a value for vrDisplay. For a VR capable display it will be 1, and for everything else it will get a 0 value. It gets a camera value passed as a parameter, and then it gets duplicated and the scene is rendered

twice for each eye. The method of stereo rendering supported in WebGL is currently achieved by rendering to the two eye buffers sequentially. This typically incurs double the application and driver overhead, despite the fact that the command streams and render states are almost identical.

One easy brute-force solution to this, as has been proposed by Oculus, is to use multiple processor cores to render both eyes simultaneously [64]. However that is not always possible when we use portable devices like mobile phones and tablets with often only one low-powered GPU. This has been discussed in WebGL specifications, but has not yet been implemented as part of the WebGL_multiview [65] extension. The idea was to expose the stereo rendering methods in OpenGL so that WebVR can use that same techniques.

Also this relates to how frustum culling works right now in WebVR scenes. If we can do efficient frustum culling, then that removes a lot of overhead from our render-path for both eyes. Efficiently recognizing polygons that are visible from a dynamic viewpoint is a well-studied problem in computer graphics. Normally, visibility determination is performed using the z-buffer algorithm*. Since this algorithm must examine every triangle in the scene, z-buffering can be very expensive for graphics processing. One way to avoid needless processing of invisible portions of the scene is to use an occlusion culling algorithm to discard invisible polygons early in the rendering pipeline. That can be very useful in urban and architectural models as we see from the work of Coorg et al. [66]. However, that technique doesn't prove to be of much use in a VR scene where we have depth-of-field information. Techniques such as exploiting frame-to-frame coherency to compare previous distance and rotation has also been successfully used to optimize culling [67], but for a very limited

*Z-Buffer Algorithm: https://www.cs.helsinki.fi/group/goa/render/piilopinnat/z_buffer.html

circumstances only.

4.1.1 Our approach

We try to tackle the problem of frustum culling and duplicate stereo rendering keeping in mind our very limited scenario of VR. The proposed method utilizes the camera to supply the renderer a generic "Culling Volume" rather than the renderer computing the frustum from the camera each frame, and doing intersection test one of those frames. This approach for a normal camera view would yield no performance improvement. But to tackle the duplication we introduce a *ArrayCamera* which receives a list of camera view points. *ArryCamera* basically will extend *Camera*, and will add an array of camera view points. *StereoCamera* extends *ArrayCamera* to populate it with the info *WebVR* gives us form HMD. The primary renderer, which for us is *WebGLrendered.renderer()*, checks *isArrayCamera*. It will still project using the main *Camera*, but will render the scene using the cameras in the array, thus avoiding duplication. Now if we use Culling Volume for just the *Camera*, it will return us a *Frustum* object, which will have no effect on the performance. However, for an *ArrayCamera*, it would return an object that tests against a Frustum for each camera in the array and will return true if the object is in any of them.

4.1.2 Modified Algorithm

1. We add `getCullingVolume` to `Camera`

2. Pass the value of `projScreenMatrix` to `getCullingVolume`

3. `ArrayCamera` is built using `ArrayFrustum` as its default `cullingVolume`

4. `ArrayFrustum` is an array with the frustums of each subcamera and it overrides the frustum methods to test against each frustum in the array

4.1.3 Multiview in Servo

Another approach was to implement multiview renderer pipeline into the browser. For Three.Js it was not possible for us to do anything so drastic, keeping in mind the browser and WebVR 1.0 W3C specification [68] standards. However, when implementing the same methods in Servo [69], we had full freedom. The multiview architecture in Servo essentially follows the below pipeline 4.1.

4.1.4 Implementation Code

For Servo, we directly open up the WebVR framebuffers that we get from the headset using the approach depicted in the WebGL Multiview[†], utilizing the Opaque multiview framebuffers. These are WebGLFramebuffer objects that act as if they have multi-view attachments, but their attachments are not exposed as textures or renderbuffers and cannot be changed. Opaque multiview framebuffers may have any combination of color, depth and stencil attachments allowing us to use WebGL 1.0 with multiview. With the only requirement for the browser being capability to support GLSL 300 [70] version [‡].

The rendering pipeline in Servo provides much lower latency because it can render straight to the headset and doesn't require the texture copy of a frame. There still was

[†]WebGL Multiview Draft: https://www.khronos.org/registry/webgl/extensions/proposals/WEBGL_multiview/

[‡]GLSL V3: https://www.khronos.org/registry/OpenGL/specs/es/3.0/GLSL_ES_Specification_3.00.pdf

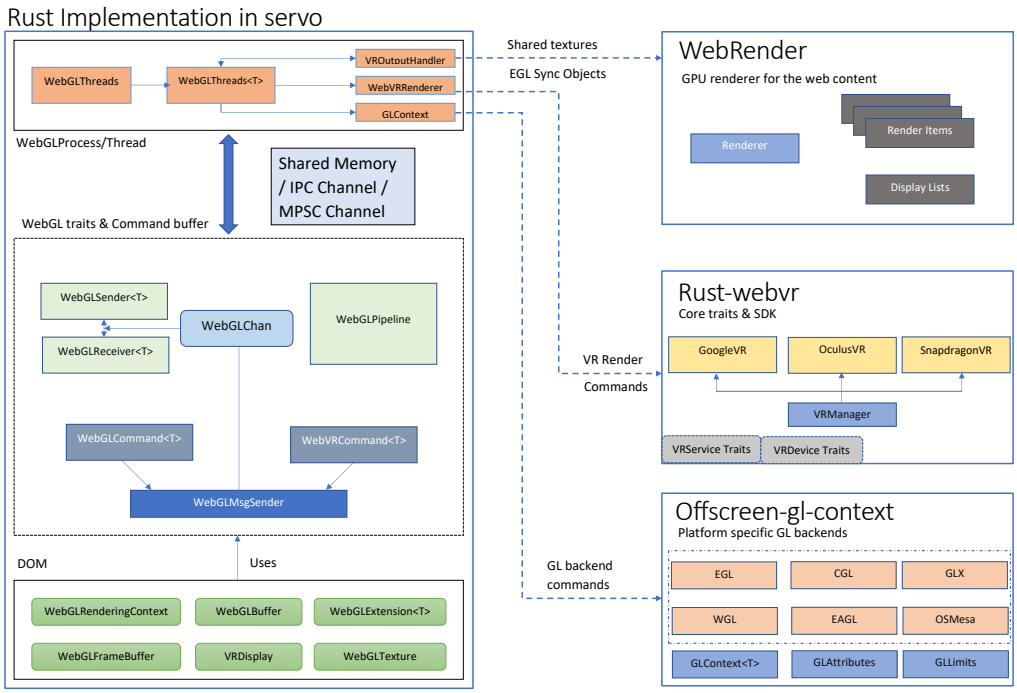


Figure 4.1 : Multiview Implementation in Servo: *The implementation is divided into three separate blocks apart from the browser part. WebRender handles the texture rendering part; offscreen-gl-context handles graphics library related computation off main thread; rust-webvr creates the VR stereoscopic image from WebVRRenderer inputs*

not a good way to expose this for WebVR in JavaScript with the present specification.

- **WebVR 1.1**[§]: Does not properly support multiview; rather supports side-by-side rendered canvas objects

[§]WebVR 1.1 Specs: https://w3c.github.io/webvr/spec/1.1/

- **WebVR 2.0**[¶]: Provides multiview support, but is still under heavy modification along with a "do not implement" status

To support multiview we implemented `WebGLFramebuffer` from WebVR 2.0 to WebVR 1.1 using an ad hoc extension method `vrDisplay.getViews()`.

[¶]WenbVR 2.0 Specs: https://w3c.github.io/webvr/spec/1.1/

Listing 4.1 The entry point to WebGLFramebuffer

```
function onAnimationFrame (t) {
    vrDisplay.requestAnimationFrame(onAnimationFrame);
    vrDisplay.getFrameData(frameData);
    if (vrDisplay.isPresenting) {
    var views = vrDisplay.getViews ? vrDisplay.getViews() : [];
    for (var i = 0; i < views.length; ++i) {
        var view = views[i];
        var multiview = view.getAttributes().multiview;
        var viewport = view.getViewport();
        gl.bindFramebuffer(gl.FRAMEBUFFER, view.framebuffer);
        gl.viewport(viewport.x, viewport.y, viewport.width,
            viewport.height);
        gl.scissor(viewport.x, viewport.y, viewport.width,
            viewport.height);
        gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
        if (multiview) {
            var projections = [frameData.leftProjectionMatrix,
                frameData.rightProjectionMatrix];
            getStandingViewMatrix(viewMat,
                frameData.leftViewMatrix);
            getStandingViewMatrix(viewMat2,
                frameData.rightViewMatrix);
            var viewMats = [viewMat, viewMat2];
            renderSceneView(projections, viewMats, frameData.pose,
                /*multiview*/ true);
            break;
        }
        else {
            // Direct render to VR framebuffer
            getStandingViewMatrix(viewMat, i == 0 ?
                frameData.leftViewMatrix :
                frameData.rightViewMatrix);
            renderSceneView(i == 0 ?
                frameData.leftProjectionMatrix :
                frameData.rightProjectionMatrix, viewMat,
                frameData.pose);
        }
    }
}
```

The advantages of this approach is that when WebGL 2.0 will be implemented in

future, these improvements will still be reusable, and these are forward compatible as long as the specification does not change.

4.1.5 Performance Gains

We used published samples from webvr.info[‖] to measure multiview impact in our WebVR implementation. From our measurements, up to forty percent improvements in CPU-bound webvr scenes have been noticed 4.2.

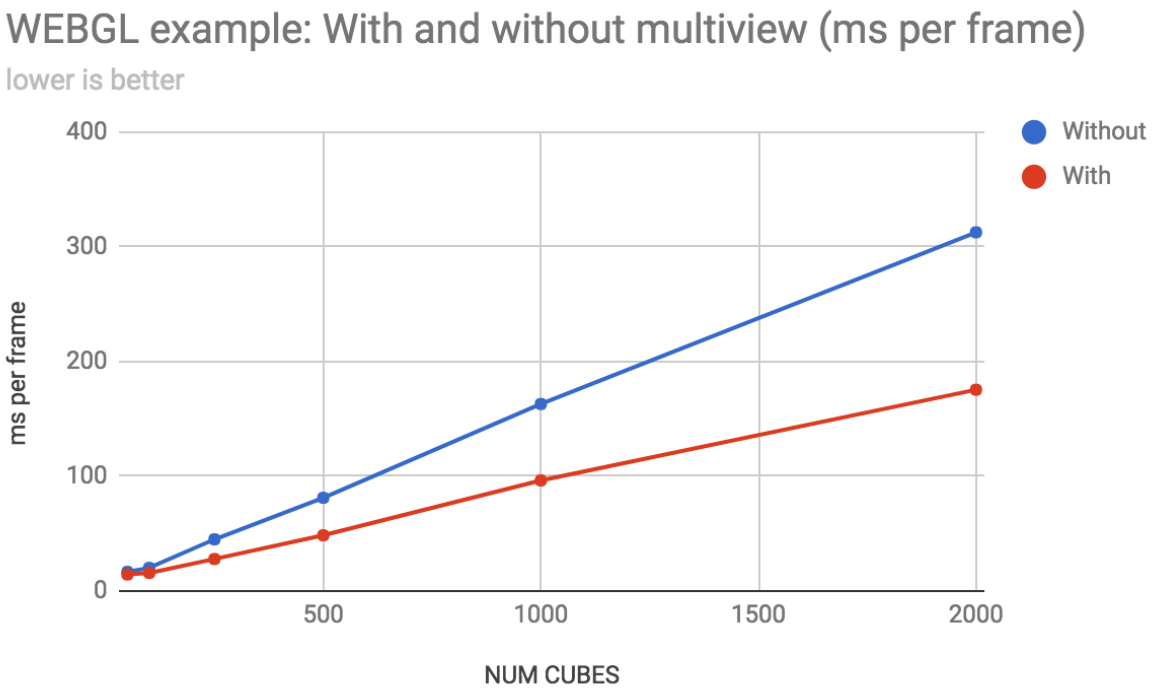


Figure 4.2 : **Multiview Implementation in Servo**: *We used a scene to create an increasing number of rotating cubes, and measured the rendering time of the scene with the increase of objects (cubes) in the scene. This intentionally uses duplicated draw-calls to simulate extreme test condition*

[‖]Samples from here: https://webvr.info/samples/

4.2 Handling Incremental texture Loading

To handle incremental texture loading inside a virtual reality scene, we need to use the `createImageBitmap` [71] API. The API implementation in Firefox does not accept an *options* argument. Our approach is to fetch and create ImageBitmap in a worker by default, which enables us to incrementally load multiple portions of the texture in parallel out of the main thread.

This can be used with any loader with a `THREE.Loader.Handlers` web hook

Listing 4.2 Web Hook to THREE.Loader.Handlers

```
1  class CustomTextureLoader {
2  constructor () {
3    this.loader = new THREE.ImageBitmapLoader();
4  }

6  load ( url, onLoad, onProgress, onError ) {
7    this.loader.load( url, ( imageBitmap ) => {
8      onLoad( new THREE.CanvasTexture( imageBitmap ) );
9    }, onProgress, onError );
10 }
11 }
12 THREE.Loader.Handlers.add( / .(png|jpg|jpeg)$/, new
     CustomTextureLoader() );
```

Listing 4.2 gives us some very interesting results. The loading time for the textures have increased to almost 2-3 times, but texture upload proves to be much faster. The overall time to first render is only slightly slower (1.1x to 1.15x). But parsing glTF does not typically block the main thread. The only significant blocking happens during texture upload. With the ImageBitmap, we spend 60-70 percent less time uploading textures, and drop fewer frames as a result.

4.2.1 Our Approach

Updatable Texture

This is an extended `THREE.Texture` method which provides support for incremental partial updates utilizing `texSubImage2D`. We create and expose a method `UpdatableTexture` that lets users update only a part of a texture: it's useful for incremental loads of large images for tiled resources.

However, this approach relies on a modified version of Three.Js which is good for benchmarking and for experimental support, but not good enough if we actually want to pursue a stable production use.

Our approach here consists of the following steps

- Image decoding should be off the thread and in a non-blocking manner so that new textures don't block the thread
- Utilize Web Worker to distribute image decoding
- Have a poll of web workers where we can delegate the loaded data
- A-Frame [44] would enable this version once it detects there is an asset loading while it is in VR mode.

4.2.2 Implementation

The implementation uses createImageBitmap** to decode image off the main thread. But the API implementation for Chrome only works for binary data blobs††. How-

**https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/createImageBitmap

††https://www.chromestatus.com/features/5637156160667648

ever, when the source is ImageElement or SVGElement, we utilize the workerpool to delegate the work.

For that, our approach is to create and utilize *WorkerDirector*. ImageBitmapLoader now uses WorkerSupport. WorkerDirector creates parallel workers to handle the image on loading and depending on the image size chunks it and creates new parallel workers. The workers are created once and then re-used. This showed us that common, instruction-based and repeatable worker usage can be applied to this class of problems.

Chapter 5

Experiments and Validation

5.1 Experiment Setup

The majority of the experiments were conducted in a desktop machine with WebVR-enabled browsers and some of them were replicated in handheld mobile devices to measure the performance impacts. The following hardware and software have been used to capture the benchmark numbers reported in this thesis.

Hardware	Specification
Processor	2.9 GHz Intel Core i7 Processor
Memory	16GB
Graphics	Intel HD 530 + Radeon Pro 460 4GB

Table 5.1 : Hardware Setup for the benchmark

Software	Specification
Operating System	OSX High Sierra & Windows 10 in Bootcamp
Browser	Firefox Nightly 59.0a1 and Chromium Experimental Build (expiry date:
Head Mounted Display	Oculus Rift

Table 5.2 : Software Setup for the benchmark

Worker Count	Time Taken to Render (ms)
1	13.8
4	9.17
8	8.54
12	10.42
16	10.82

Table 5.3 : Object Loader with 128 Objects in Parallel

For the mobile device, we used a Google Pixel XL running android 8.0 Oreo along with Firefox Nightly and Google Daydream.

5.2 Object Loader Performance

The experimental scene loads a threejs scene in a VR-enabled browser with web worker support. Since without worker the scene essentially will block the loading we test the performance impact of parallelizing the scene using worker with a different worker count. We test it with variable worker counts having a fixed queueLength and measure the loading time. To rule out any network related latency, the tests were run from local machine and multiple times to compensate for network-based latencies, if any. The models used were standard obj files. We can see the result in 5.1 that the most efficient rendering is possible with around 8 web workers loading the scene. The graph is drawn from the data collected in 5.3. The experiment was performed in Firefox Nightly.

The experimental scene used to measure the worker performance is shown in Figure 5.2.

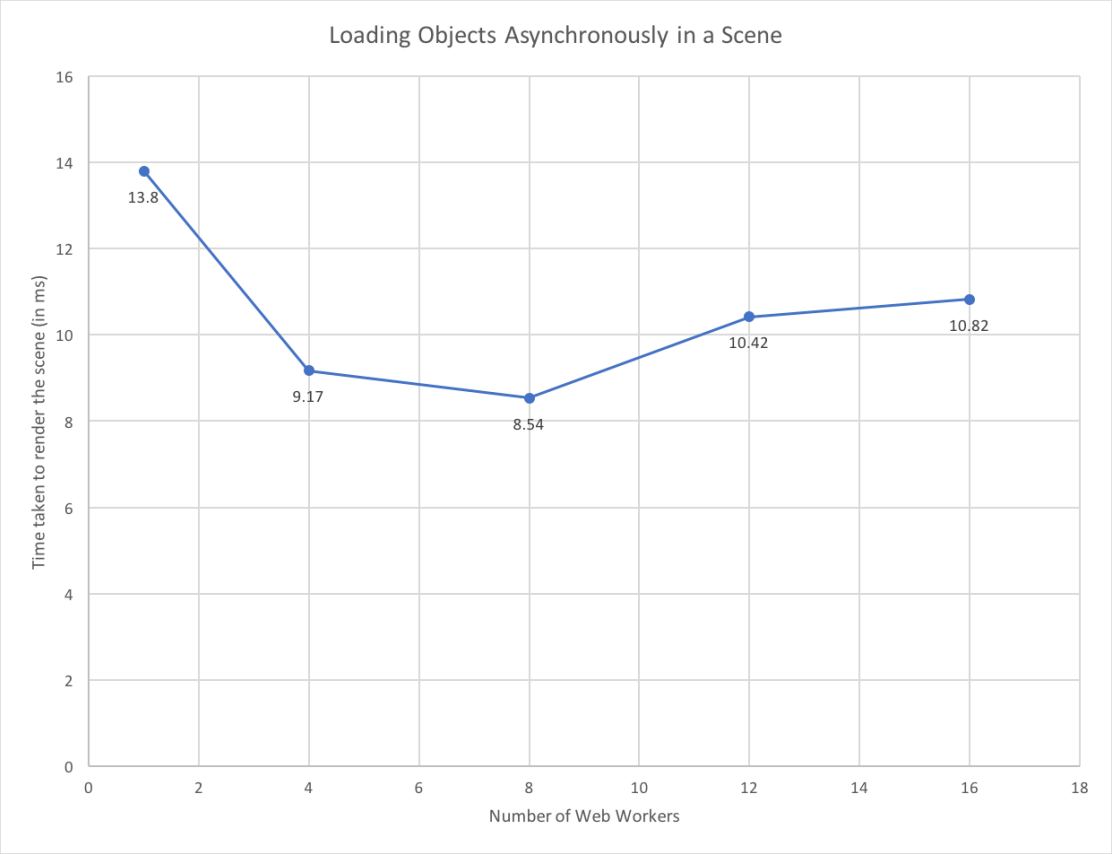


Figure 5.1 : We load 128 object from a queue with varying numbers of Web Workers enabled. The graph shows a lowest rendering time of 8.54 ms when 8 web workers are running

5.3 Web Worker Performance Benchmarks

Our hypothesis was that using worker to exploit parallelism we will be able to achieve better graphics performance in virtual reality and mixed reality scenes, and not only in computations. To test that we made a benchmark suite to test our hypothesis to validate web worker enabled objloader creation.

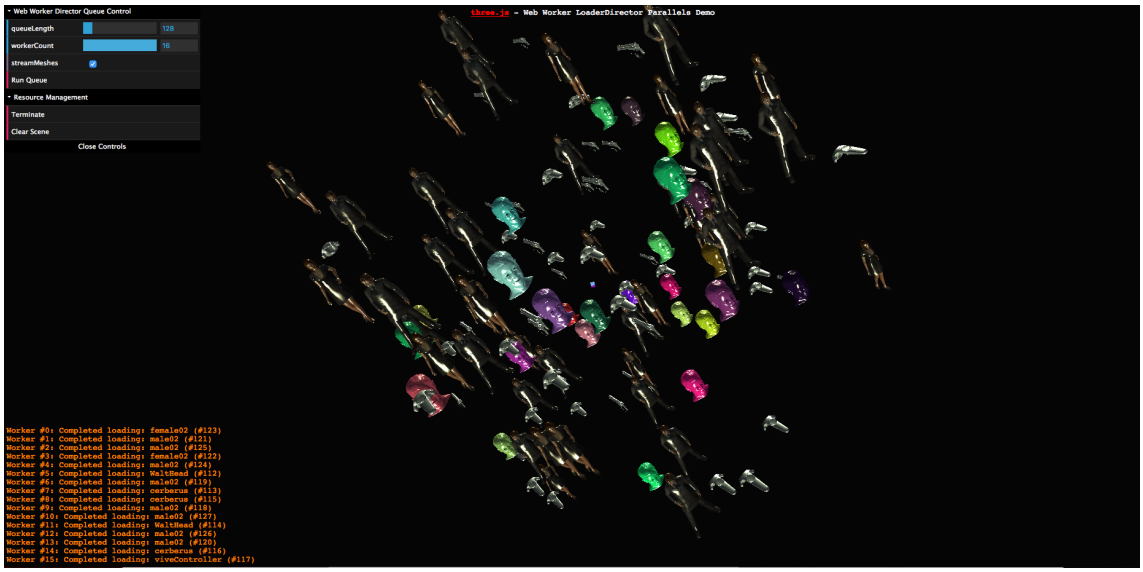


Figure 5.2 : The threejs scene used to test out web worker performance. It allows us to dynamically change worker and object queue to test loading time

5.3.1 Adaptive Threshold

This demo implements the approach proposed by Bradley et al.[72] in their paper utilizing illumination change. This demo takes a video as an input with an Augmented Reality marker on it and uses the techniques proposed in the paper to detect the pattern. As we can see in the figure 5.3 and from the data in table 5.4, the Web Worker version is almost 1.7x faster than the vanilla javascript version while running the algorithm in real-time in Firefox Nightly within ThreeJS. The benchmark suite is available at `https://svn.rice.edu/r/parsoft/projects/Rabimba-Karanjai-Research-Project/WebWorker_benchmark/`*

*Web Worker vs JavaScript benchmark: https://svn.rice.edu/r/parsoft/projects/Rabimba-Karanjai-Research-Project/WebWorker_benchmark/

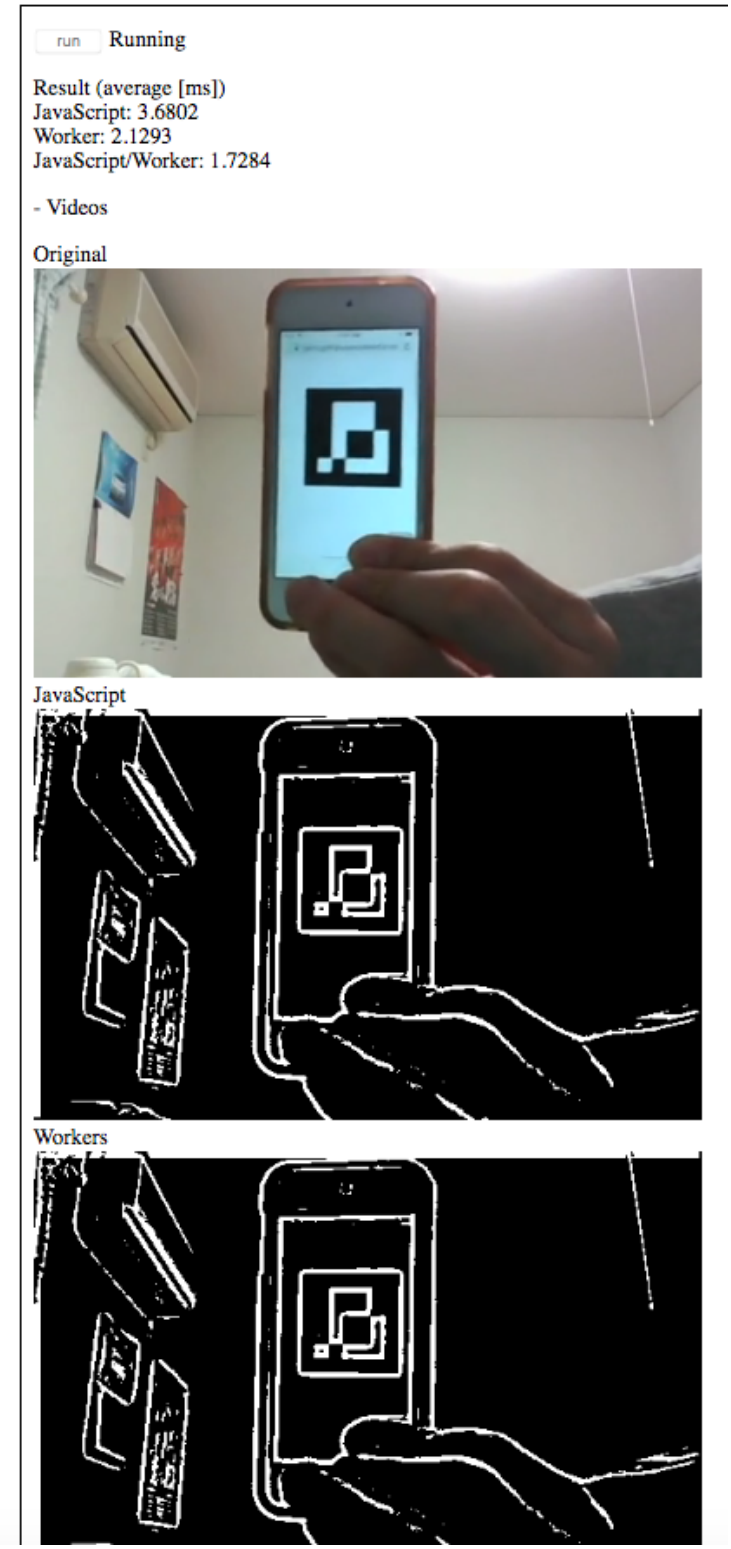


Figure 5.3 : Using Video Threshold detection in threejs using webworker and without using webworker

JavaScript Implementation runtime (ms)	WebWorker implementation runtime (ms)
3.6802	2.1293

Table 5.4 : Runtime comparison for each iteration of pattern detection for Adaptive Thresholding on the video for vanilla javascript and with webworker

Chapter 6

Conclusion and Future Work

The optimization of Web Virtual Reality and Web Mixed reality applications are an important and time-critical problem. In this thesis, I explore three of the existing challenges in this area and propose three solutions to address them. The use of parallelization with the help of Web Workers enables us to parse and load objects in a non-blocking manner. Using tiling methods and offloading to worker threads helped us to incrementally load large objects and textures into the webview without blocking the browser, thereby avoiding frame skips. The use of shared arraybuffer to de-duplicate stereoscopic rendering for VR view lets us make rendering pipelines faster. And since all of the three solutions are implemented in the Three.js framework, any library utilizing Three.js automatically benefits from these including Aframe [44], which is used to create Web Virtual Reality scenes and WebXR [53] with Aframe, which is used to create Web Mixed Reality applications.

There are many opportunities for future research and exploration into optimization and improvement of the framework. The problems initially were explored with WebVR in mind. Especially dealing with multi-user scenarios for social VR, which can include virtual classrooms, meeting rooms, training rooms etc. All of these require realtime low latency, high quality renders as well as synchronization. Till now, most of the environments deal with local objects; with multi-user scenarios we will have shared objects and minimal new latency requirements related to ensuring that all the users are in sync, thereby opening up new avenues for performance improvements.

There are also many different directions that be pursued for improving the performance. Regarding the existing frameworks, there still are a lot of avenues that can be explored further. In this thesis, we mostly looked into using web workers. But another avenue to off load complex computations is Web Assembly *. WebAssembly is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages such as C/C++ with a compilation target so that they can run on the web †. It is supported by Chrome, Edge, Firefox, and WebKit, making it universal enough to be run in most machines. In our initial explorations, it was noticed that the call back times from JavaScript eventually produced an overhead to even off the performance improvement benefits it gave. As we can see, If we implement the same Adaptive Thresholding from 5, a comparison with WebAssembly implementation and JavaScript gives us worse performance for WebAssembly in Figure 6.1. The benchmark code is available at https://svn.rice.edu/r/parsoft/projects/Rabimba-Karanjai-Research-Project/WASM_benchmark/ ‡

However, if we look at mathematical operations like integer multiplications, web assembly gives us almost 4.5x speedup on the operations as we see from Figure 6.2.

It would be quite interesting to have a way or tool to automatically analyze a code snippet and advise the developer as to what kind of optimization is most beneficial for her/his WebVR app. Eventually, we want to have WebVR and WebMR applications that perform consistently on par with native applications on any platform including

*WebAssembly: http://webassembly.org/

†WebAssembly Specs: https://developer.mozilla.org/en-US/docs/WebAssembly

‡WASM Benchmark: https://svn.rice.edu/r/parsoft/projects/Rabimba-Karanjai-Research-Project/WASM_benchmark/

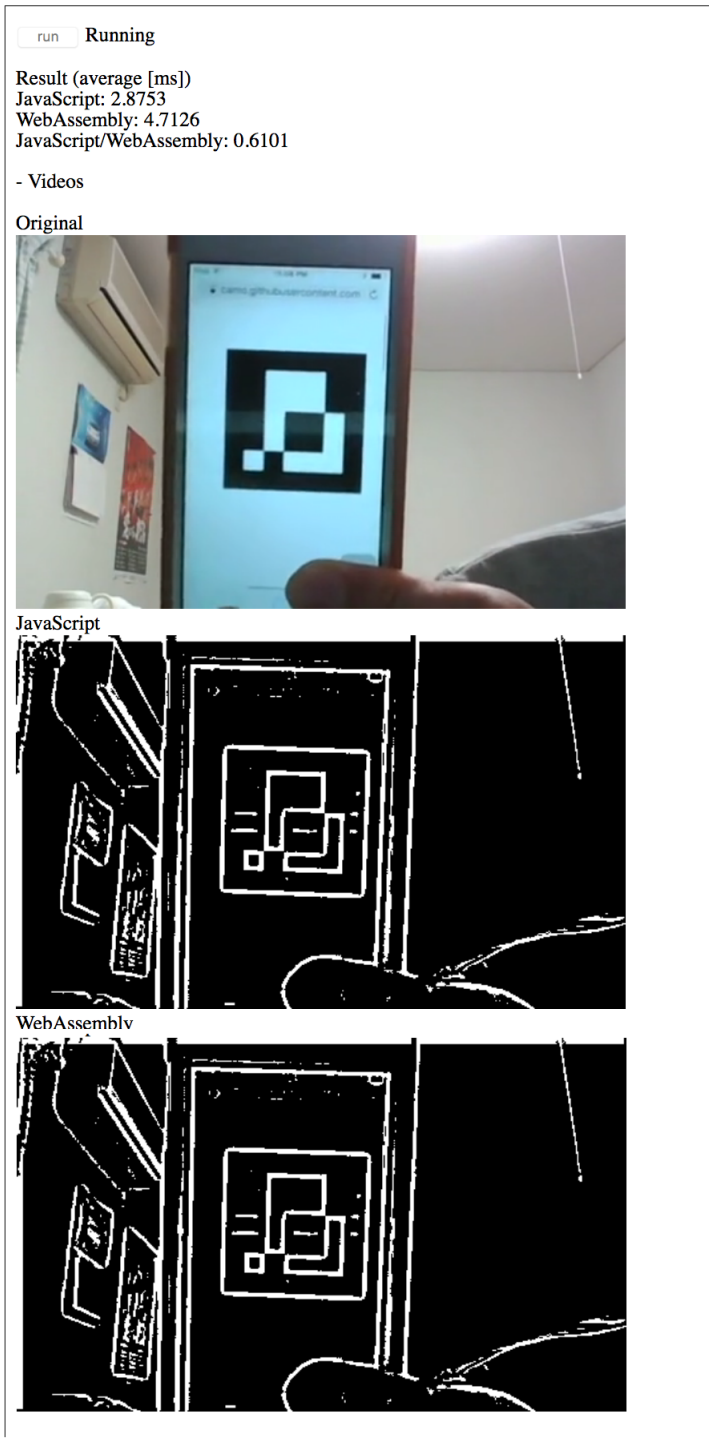


Figure 6.1 : Using Video Threshold detection in threejs using WebAssembly and with JavaScript

run Done

Result (average [ms])
JavaScript: 1070.4110
WebAssembly: 238.3550
JavaScript/WebAssembly: 4.4908

+ Test code

- JavaScript code

```
function jsMultiplyInt(a, b, n) {
  var c = 1.0;
  for (var i = 0; i < n; i++) {
    c = c * a * b;
  }
  return c;
}
```

- WebAssembly C code

```
int multiplyInt(int a, int b, int n) {
  int c = 1.0;
  for (int i = 0; i < n; i++) {
    c = c * a * b;
  }
  return c;
}
```

+ WebAssembly Compile shell script

+ WebAssembly Instantiation code

Figure 6.2 : Using Web Assembly for integer multiplication compared to using javascript

the mobile.

Bibliography

[1] D. A. Bowman and R. P. McMahan, "Virtual reality: How much immersion is enough?" *Computer*, vol. 40, no. 7, pp. 36–43, July 2007.

[2] "Mozvr "hiro" leap-enabled," https://www.youtube.com/watch?v=KlZnKW2qVZ8, (Accessed on 12/10/2017).

[3] R. T. Azuma, "A survey of augmented reality," *Presence: Teleoperators and virtual environments*, vol. 6, no. 4, pp. 355–385, 1997.

[4] "Decoratear | devpost," https://devpost.com/software/decoratear, (Accessed on 12/10/2017).

[5] I. E. Sutherland, "A head-mounted three dimensional display," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I.* ACM, 1968, pp. 757–764.

[6] D. Cohen, "Incremental methods for computer graphics," HARVARD UNIV CAMBRIDGE MA DIV OF ENGINEERING AND APPLIED PHYSICS, Tech. Rep., 1969.

[7] K. C. Knowlton, "Computer displays optically superimposed on input devices," *The Bell System Technical Journal*, vol. 56, no. 3, pp. 367–383, March 1977.

[8] C. Schmandt, "Spatial input/display correspondence in a stereoscopic computer graphic work station," *SIGGRAPH Comput. Graph.*, vol. 17, no. 3, pp. 253–261,

Jul. 1983. [Online]. Available: http://doi.acm.org/10.1145/964967.801156

[9] N. Negroponte, "Media room," *Proc of the Society for Information Display*, vol. 22, p. 2, 1981.

[10] C. F. Herot, "Spatial management of data," *ACM Trans. Database Syst.*, vol. 5, no. 4, pp. 493–513, Dec. 1980. [Online]. Available: http://doi.acm.org/10.1145/320610.320648

[11] S. S. Fisher, M. McGreevy, J. Humphries, and W. Robinett, "Virtual environment display system," in *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, ser. I3D '86. New York, NY, USA: ACM, 1987, pp. 77–87. [Online]. Available: http://doi.acm.org/10.1145/319120.319127

[12] M. K. Elden, "Implementation and initial assessment of vr for scientific visualisation: Extending unreal engine 4 to visualise scientific data on the htc vive," Master's thesis, 2017.

[13] "The technical challenges of virtual reality," https://iq.intel.com/the-technical-challenges-of-virtual-reality/, (Accessed on 12/10/2017).

[14] C. Zellweger, B. Barberis, C. Kim, C. Conlee, and B. Robertson, "Head mounted display," Jul. 12 2016, uS Patent D761,258. [Online]. Available: https://www.google.com/patents/USD761258

[15] P. R. Desai, P. N. Desai, K. D. Ajmera, and K. Mehta, "A review paper on oculus rift-a virtual reality headset," *CoRR*, vol. abs/1408.1173, 2014. [Online]. Available: http://arxiv.org/abs/1408.1173

[16] Microsoft, "Microsoft hololens: a new way to see your world," 2015. [Online]. Available: https://www.microsoft.com/microsoft-hololens/en-us/hardware

[17] H. Chen, A. S. Lee, M. Swift, and J. C. Tang, "3d collaboration method over hololens™and skype™end points," in *Proceedings of the 3rd International Workshop on Immersive Media Experiences*, ser. ImmersiveME '15. New York, NY, USA: ACM, 2015, pp. 27–30. [Online]. Available: http://doi.acm.org/10.1145/2814347.2814350

[18] Apple, "Arkit," 2017. [Online]. Available: https://developer.apple.com/documentation/arkit

[19] Google, "Arcore: Augmented reality at android scale," 2017. [Online]. Available: https://www.blog.google/products/google-vr/arcore-augmented-reality-android-scale/

[20] C. Leung and A. Salga, "Enabling webgl," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 1369–1370. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772933

[21] C. McAnlis, P. Lubbers, B. Jones, D. Tebbs, A. Manzur, S. Bennett, F. d'Erfurth, B. Garcia, S. Lin, I. Popelyshev, J. Gauci, J. Howard, I. Ballantyne, J. Freeman, T. Kihira, T. Smith, D. Olmstead, J. McCutchan, C. Austin, and A. Pagella, *Optimizing WebGL Usage*. Berkeley, CA: Apress, 2014, pp. 147–162. [Online]. Available: https://doi.org/10.1007/978-1-4302-6698-3_9

[22] B. Danchilla, *Three.js Framework*. Berkeley, CA: Apress, 2012, pp. 173–203. [Online]. Available: https://doi.org/10.1007/978-1-4302-3997-0_7

[23] R. Cabello *et al.*, "Three. js," *URL: https://github. com/mrdoob/three. js*, 2010.

[24] Y. Liu, H. Liu, Y. Zhao, and R. Song, "Teaching of advanced computer graphics with three. js," in *Proceedings of International Conference on Education and New Developments*, 2016, pp. 13–17.

[25] B. Watson, N. Walker, W. Ribarsky, and V. Spaulding, "Effects of variation in system responsiveness on user performance in virtual environments," *Human Factors*, vol. 40, no. 3, pp. 403–414, 1998.

[26] J. Lee, M. Kim, and J. Kim, "A study on immersion and vr sickness in walking interaction for immersive virtual reality applications," *Symmetry*, vol. 9, no. 5, p. 78, 2017.

[27] A. S. Fernandes and S. K. Feiner, "Combating vr sickness through subtle dynamic field-of-view modification," in *3D User Interfaces (3DUI), 2016 IEEE Symposium on*. IEEE, 2016, pp. 201–210.

[28] W. W. S. Group, "Fast stereo rendering for vr," 2016. [Online]. Available: https://github.com/w3c/webvr/issues/101

[29] D. HRACHOVÝ, "Faster webgl graphics," Ph.D. dissertation, Masarykova univerzita, Fakulta informatiky, 2012.

[30] S. Kang and J. Lee, "Developing a tile-based rendering method to improve rendering speed of 3d geospatial data with html5 and webgl," *Journal of Sensors*, vol. 2017, 2017.

[31] "kripken/webgl-worker," https://github.com/kripken/webgl-worker, (Accessed on 12/10/2017).

[32] "Webgl in web workers, today - and faster than expected! - mozilla research," https://research.mozilla.org/2014/07/22/webgl-in-web-workers-today-and-faster-than-expected/, (Accessed on 12/10/2017).

[33] K. Pimentel and K. Teixeira, "Virtual reality through the new looking glass," 1993.

[34] A. Akins, "Virtual reality and the physically disabled: Speculations of the future," in *Proceedings of Virtual Reality and Persons with Disabilities Conference, Northridge, CA*, 1992.

[35] M. Pilz and H. A. Kamel, "Creation and boundary evaluation of csg-models," *Engineering with computers*, vol. 5, no. 2, pp. 105–118, 1989.

[36] G. Bell, A. Parisi, and M. Pesce, "The virtual reality modeling language: version 1.0 specification," *URL: http://vrml. wired. com/vrml. tech/vrml10-3. html*, 1995.

[37] B. Laurel, R. Strickland, and R. Tow, "Placeholder: Landscape and narrative in virtual environments," *ACM SIGGRAPH Computer Graphics*, vol. 28, no. 2, pp. 118–126, 1994.

[38] "Canvas 3d gl." [Online]. Available: https://web.archive.org/web/20110717224855/http://blog.vlad1.com/2007/11/26/canvas-3d-gl-power-web-style/

[39] "Opera webgl." [Online]. Available: https://web.archive.org/web/20071117170113/http://my.opera.com/timjoh/blog/2007/11/13/taking-the-canvas-to-another-dimension

[40] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[41] C. Leung and A. Salga, "Enabling webgl," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 1369–1370.

[42] "Hero." [Online]. Available: http://www.joshcarpenter.ca/vr-browsing-explorations/

[43] "Three.js." [Online]. Available: https://github.com/mrdoob/three.js/issues/1960

[44] "A-frame." [Online]. Available: https://github.com/aframevr/aframe

[45] A. Gill, "Aframe: A domain specific language for virtual reality," in *Proceedings of the 2nd International Workshop on Real World Domain Specific Languages*. ACM, 2017, p. 4.

[46] I. E. Sutherland, "The ultimate display," *Multimedia: From Wagner to virtual reality*, 1965.

[47] R. Kooper and B. MacIntyre, "Browsing the real-world wide web: Maintaining awareness of virtual information in an ar information space," *International Journal of Human-Computer Interaction*, vol. 16, no. 3, pp. 425–446, 2003.

[48] W. Piekarski, "3d modeling with the tinmith mobile outdoor augmented reality system," *IEEE Computer Graphics and Applications*, vol. 26, no. 1, pp. 14–17, 2006.

[49] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalavári, L. M. Encarnaçao, M. Gervautz, and W. Purgathofer, "The studierstube augmented reality project,"

Presence: Teleoperators and Virtual Environments, vol. 11, no. 1, pp. 33–54, 2002.

[50] D. Schmalstieg, T. Langlotz, and M. Billinghurst, "Augmented reality 2.0," in *Virtual realities*. Springer, 2011, pp. 13–37.

[51] J. C. Spohrer, "Information in places," *IBM Systems Journal*, vol. 38, no. 4, pp. 602–628, 1999.

[52] B. MacIntyre, A. Hill, H. Rouzati, M. Gandy, and B. Davidson, "The argon ar web browser and standards-based ar application environment," in *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*. IEEE, 2011, pp. 65–74.

[53] "mozilla/webxr-api: A proposal for webxr, based on the webvr extension," https://github.com/mozilla/webxr-api, (Accessed on 12/07/2017).

[54] "A-painter: Paint in vr in your browser," https://blog.mozvr.com/a-painter/, (Accessed on 12/10/2017).

[55] "Incremental loading and non-blocking parsing and rendering · issue," https://github.com/mrdoob/three.js/issues/4397, (Accessed on 11/28/2017).

[56] "Api to automatically force preload all material uploads to gpu · issue," https://github.com/aframevr/aframe/issues/3135, (Accessed on 11/28/2017).

[57] "three.js webgl - gltf 2.0," https://threejs.org/examples/webgl_loader_gltf.html, (Accessed on 11/28/2017).

[58] "gltf-sample-models/2.0/boombox at master · khronosgroup/gltf-sample-models," https://github.com/KhronosGroup/glTF-Sample-Models/tree/

master/2.0/BoomBox, (Accessed on 11/28/2017).

[59] "gltf-sample-models/2.0/damagedhelmet at master · khronosgroup/gltf-sample-models," https://github.com/KhronosGroup/glTF-Sample-Models/tree/master/2.0/DamagedHelmet, (Accessed on 11/28/2017).

[60] "react-vr: Wavefrontobj," https://github.com/facebook/react-vr/tree/master/ReactVR/js/Loaders/WavefrontOBJ, (Accessed on 11/28/2017).

[61] "react-vr:objparser.js," https://github.com/facebook/react-vr/blob/master/ReactVR/js/Loaders/WavefrontOBJ/OBJParser.js#L281-L298, (Accessed on 11/28/2017).

[62] "Reflect - javascript | mdn," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect, (Accessed on 12/03/2017).

[63] "Vreffect," https://github.com/mrdoob/three.js/blob/dev/examples/js/effects/VREffect.js, (Accessed on 11/28/2017).

[64] Advanced vr rendering performance gdc. [Online]. Available: https://alex.vlachos.com/graphics/Alex_Vlachos_Advanced_VR_Rendering_Performance_GDC2016.pdf

[65] K. Group, "Webgl multiview extension proposed specification," https://www.khronos.org/registry/webgl/extensions/proposals/WEBGL_multiview/, (Accessed on 11/28/2017).

[66] S. Coorg and S. Teller, "Real-time occlusion culling for models with large occluders," in *Proceedings of the 1997 symposium on Interactive 3D graphics*. ACM, 1997, pp. 83–ff.

[67] U. Assarsson and T. Moller, "Optimized view frustum culling algorithms for bounding boxes," *Journal of graphics tools*, vol. 5, no. 1, pp. 9–22, 2000.

[68] "Webvr," https://w3c.github.io/webvr/spec/1.1/, (Accessed on 11/28/2017).

[69] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Engineering the servo web browser engine using rust," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 81–89.

[70] J. Simpson, "The opengl es® shading language language version: 3.10 document revision: 3," 2014.

[71] "self.createimagebitmap() - web apis | mdn," https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/createImageBitmap, (Accessed on 12/04/2017).

[72] D. Bradley and G. Roth, "Adaptive thresholding using the integral image," *Journal of Graphics Tools*, vol. 12, no. 2, pp. 13–21, 2007.