Examensarbete

# WEBGL VS WEBGPU
Är WebGPU redo att ta WebGLs plats?

# WEBGL VS WEBGPU
Is WebGPU ready to take WebGLs place?

Examensarbete för kandidatexamen med
huvudområdet Informationsteknologi

Grundnivå 30 Högskolepoäng

Vårtermin 2024

Student: Kristoffer Danielsson

Handledare: Henrik Gustavsson

Examinator: Yacine Atif

# 1 Abstract

This thesis aims to study the performance differences between WebGL and WebGPU using the libraries Three.js and Babylon.js through an experimental approach. By developing an e-commerce website hosting both multiple and individual 3D models, external scripts were used to measure the loading and rendering time as well as the frames per second. The frames per second were measured by removing the frame limitation over a period of 20 minutes and the loading and rendering times were measured by repeatedly reloading the website and measuring how long it took, over the course of 300 measurements. The findings from this experiment shows that Babylon.js using WebGPU does in fact not present any superiority in terms of performance, where Three.js using WebGL consistently performed better than the other combinations of API and library. Given these foundational studies showing that WebGPU performs better than WebGL in other contexts, further research is needed to understand in which contexts each API excels and if so, what optimizations and specifications are necessary for WebGPU to shine.

Keywords: WebGL, WebGPU, Three.js, Babylon.js, E-commerce, 3D on the web.

# 2 Table of Contents


**3 INTRODUCTION** ..... **1**

**4 BACKGROUND** ..... **2**

4.1 OpenGL ..... 2

*4.1.1 WebGL* ..... *2*

*4.1.2 WebGPU* ..... *3*

*4.1.3 WebGL in E-commerce* ..... *4*

4.2 JavaScript ..... 5

*4.2.1 Libraries* ..... *5*

*4.2.2 Three.js* ..... *6*

*4.2.3 Babylon.js* ..... *8*

4.3 Application ..... 11

**5 PROBLEM** ..... **12**

5.1 Research Question ..... 13

5.2 Hypothesis ..... 13

**6 METHOD** ..... **14**

6.1 Alternative Methods ..... 14

6.2 Chosen Method ..... 15

6.3 Ethics & Validity ..... 16

*6.3.1 Technical* ..... *16*

*6.3.2 Environmental* ..... *16*

*6.3.3 Sustainability* ..... *16*

*6.3.4 Personal Integrity* ..... *17*

*6.3.5 GitHub* ..... *17*

**7 LITERATURE** ..... **18**

**8 IMPLEMENTATION** ..... **19**

8.1 Data ..... 20

8.2 Artifact ..... 21

*8.2.1 Three.js* ..... *21*

*8.2.2 Babylon.js* ..... *25*

8.3 Measurement Scripts ..... 28

*8.3.1 FPS Tracker* ..... *28*

*8.3.2 Load Time Tracker* ..... *30*

8.4 Progression ..... 32

*8.4.1 TransformNodes* ..... *32*

*8.4.2 Quaternion* ..... *33*

*8.4.3 Scaling Models* ..... *33*

*8.4.4 GM_Value* ..... *33*

**9 PILOT STUDY** ..... **34**

9.1 Discussion ..... 36

**10 ANALYSIS** ..... **37**

10.1 FPS Values ..... 37

*10.1.1 Catalogue Version* ..... *38*

*10.1.2 Individual Version* ..... *40*

10.2 Load & Render Time Values ..... 42

*10.2.1 Catalogue Version* ... *42*
*10.2.2 Individual Version* ... *44*

**11 DISCUSSION ... 47**

11.1 ETHICS ... 47
11.2 SUSTAINABILITY ... 48
11.3 CONCLUSION ... 48
11.4 FUTURE WORK ... 49

**12 REFERENCES ... 50**

# 3 Introduction

We humans live in a 3D world, although this hasn't necessarily always been the case in the computer world. Though with modern browser technologies and hardware-accelerated rendering of graphics 3D is now a fully functional option even for modern cellphones (Sing, 2019). One of the many reasons this came to be was the development of WebGL which is an API for 3D application development in the web browser that runs alongside what is known as HTML5 (Sing, 2019). This development that came to be only a little over ten years ago (Wikipedia, 2024) introduced the opportunity for 3D application development to be accessible to developers worldwide, only requiring a browser and a text editor (Sing, 2019). The development of WebGL was a collaborative effort between Mozilla, Google, Apple and Opera and was developed with the mentality of not needing external applications, to make it simple and manageable for even smaller developers (Spider Digital Group, 2024).

Because of the further development of the 3D application development potentials in the browser, more demands and necessities of developers have come to light that WebGL potentially couldn't satisfy. Because of these needs, further development has taken place and a potential successor has been developed known as WebGPU (Manor, 2021). WebGPU is based on another API known as Vulkan, which has initially shown to be superior in terms of performance relative to WebLG (Manor, 2021).

These developments aren't just good for the use-cases that one might assume, for example video games, virtual reality, data visualization, architectural modeling and so much more (Spider Digital Group, 2024), but also the world of e-commerce. The usage of 3D models in the e-commerce world brings about a lot of positive effects, where users are given the ability to interact with products in an immersive manner and get more information to make informed decisions in what products might be of interest (Reydar, n.d.). In an article by Reydar, 3D experiences double the users' level of engagement, increase the perceived willingness to pay a higher cost for a product and increase interest in shopping to name a few of the found benefits (Reydar, n.d.).

If we are to further develop the possibility of a 3D world in the browser, we must also cater to the importance of the world outside of the browser. The environment that we live in is a core part of our economy, society and even our very existence (WWF, n.d.). The development of 3D possibilities in the browser can have a positive effect on the environment, according to an article in Sana written by Camille Collins. She bases this on an international conference on environmental science and engineering study done in 2011, where the positives that were introduced were the possibility of lowering transportation emissions as there isn't a need for potential shoppers to travel to the stores, a lower amount of paper being wasted as information is transferred digitally and the potential to eliminate warehouses and create on-demand production as the digital transfer of information doesn't place the same need for a physical warehouse (Collins, 2021).

This thesis aims to show if the usage of WebGPU can increase the performance of an e-commerce website to further strengthen the potential benefits of 3D development in the e-commerce world.

# 4 Background

The Background chapter will contain all the necessary information to understand the future development and the details behind the goal of this thesis.

## 4.1 OpenGL

OpenGL is an API (application programming interface) that is used for rendering 2D and 3D vector-based graphics using hardware-accelerated rendering, which means the usage of computer hardware to facilitate the rendering process. It was released in 1992 and was based on immediate-mode graphics and a pipeline architecture (Angel, 2017). The first edition of OpenGL was based around vertices that were assembled into geometric objects specified by parameters within the beginning “glBegin” and the “glEnd”, that when executed passed through the pipeline for display.

```
glBegin(GL_LINE_LOOP);
glVertex2i(40, 40);
glVertex2i(40, 90);
glVertex2i(90, 40);
glEnd();
```

**Figure 1** – OpenGL Code Example.

**Figure 1** above is an example of how an early OpenGL drawing could look. The information was later not actually kept, so any changes and modifications required a re-execution of the code that defined the vertices (Angel, 2017). In OpenGL 2 the addition of support for programmable shaders and the OpenGL Shading Language (GLSL) was introduced, it still ran the same pipeline model (Angel, 2017). In OpenGL 3 the pipeline and functions that supported it were all deprecated and introduced the necessity for the developers to implement their own shaders and buffers for holding vertex properties such as position, color and texture coordinates (Angel, 2017). The primary advantages of OpenGL are that it is both language-independent, meaning it can be used across numerous programming languages and that it’s cross-platform, meaning it can be ran across numerous different platforms. It is currently not in active development any longer, with the last stable release being in 2017. It has historically become the standard for teaching computer graphics to various fields, especially computer science (Angel, 2017). The evolution of the OpenGL API gave way for the JavaScript implementation of OpenGL ES 2.0 (OpenGL for Embedded Systems 2.0), aptly named WebGL (Angel, 2017) and is a cross-platform API that works without plugins (Panchal et al, 2022).

### 4.1.1 WebGL

WebGL is considered as part of HTML5, which is the fifth major HTML edition. WebGL is a cross-platform and royalty-free JavaScript API integrated into the HTML “canvas” element (Resch, Wohlfahrt & Wosniok, 2013) for rendering low-level 3D graphics within web browsers that allows GPU accelerated usage of physics and image processing and can be combined with other HTML elements (Miao, Song & Zhu, 2017). It’s currently supported by a wide variety of web browsers and hardware products (Miao, Song & Zhu, 2017), (Angel, 2017) and its usage is only growing daily (Zhang et al, 2023). The benefits of WebGL are many, it’s capable of being ran in essentially all relevant browsers where these browsers provide code development environments including debuggers, its performance is clearly efficient and has been for some

time, it provides easy integration with other web APIs and the addition of interpreted code aids development (Angel, 2017). One of many reasons that WebGL gained such an established position in the graphics environment is that even as early as 2012 it showed a significantly increased speed when it comes to rendering because of the GPU acceleration, performing better than the then popular Adobe Flash (Hoetzlein, 2012) as well as having showed a performance that rivals compiled code (Angel, 2017). This means that WebGL can access the GPU without external plugins (Feng et al, 2011). Though Hoetzlein (2012) did find that WebGL was slower than native OpenGL because it used JavaScript for execution. Hoetzlein wasn't alone in finding the speed of WebGL to be problematic, Li et al (2020) found that third-party 3D JavaScript rendering engines based on WebGL with the combination of loading entire model files at once and running synchronous, meaning that the client needs to wait for all model data transmission to be complete before loading and rendering the model, there can be a considerable delay.

There is a lack of research done on the topic of providing realistic 3D graphics on the web although WebGL does supports the addition of textures to a given object (Yu & Liu, 2016). This hasn't always been the case though, as in 2014 an article was published by Movania, Chiew & Lin (2014) mentions that WebGL at that time was based on OpenGL ES 2.0 which was a restricted subset of the desktop OpenGL, not all functionalities appear. Specifically, the functionality to add 3D textures which was implemented with a workaround by them by tiling each slice of a 3D texture into a flat 2D texture. Though there is a study done by Moloo et al. (2016) that analyzes the difference in time rate and frame rate between objects with and without textures in a WebGL-based environment with Three.js. Their finding shows that the more complex an object with textures is, the performance as well as loading time goes down. They further argue that while performances and optimizations may have improved their code, innate limitations remain and that users using a website running WebGL with textured 3D models should use a computer capable of good processing power.

Given that WebGL is based on OpenGL ES, which in turn is based on OpenGL (Angel, 2017) we can look at studies analyzing the performance of OpenGL ES compared to other APIs. One API that was developed just to handle the shortcomings of OpenGL ES was Vulkan, which was developed to address the weakness that OpenGL ES suffers from which is a high overhead. (Lujan et al., 2019). Further in the experiment by Lujan et al. (2019) they find that Vulkan outperforms OpenGL ES in both terms of absolute performance and performance with saving energy. Another experiment comparing OpenGL and Vulkan was done by Ferraz et al. (2021), the experiment was done with multiple different tests and showed that Vulkan achieves higher performance compared to OpenGL in two out of the three main tests. These two articles are of high relevance as another article was done by Usher & Pascucci (2020) showed that the performance of WebGPU is on par with Vulkan except for in smaller data sets where WebGPU was found to be slightly less efficient.

### 4.1.2 WebGPU

WebGPU is the practical name for a proposed and potential web standard and API for hardware accelerated graphics and computing. What differentiates WebGPU from WebGL is that WebGPU is not a direct port of any existing native API, while WebGL is based on OpenGL, WebGPU is based on Vulkan, Metal and Direct3D 12 and is meant to be a successor to WebGL. This is currently in development by the W3C (World Wide Web Consortium) with a broad array of engineers. The name comes from the ability to use the underlying system's GPU (Graphical

Processing Unit) to carry out the computations, drawing and rendering of complex images in the browser. While this hasn't been added as a web standard, it does find support in many browsers. Using WebGPU, the developer gains access to low-level GPU resources.

This can potentially solve the issue with the response time, given that WebGPU has shown to be beneficial in other studies, for example in a study done by Dyken et al. (2022) analyzed their own tool named GraphWaGu which is a web-based GPU-accelerated library for interacting with large-scale graphs that utilizes WebGPU to create the visualizations from input graphs onto mouse-interactive HTML Canvas elements. The results of their experiment were that GraphWaGu outperformed the other frameworks on both a high-end computer with a dedicated GPU and a low-end computer using an integrated GPU. Dyken et al. (2022) further lifts the advantages of WebGPU over WebGL for developing complex compute and rendering applications in the browser as well as WebGPU enabling rendering applications to construct a description of the rendering or compute pipeline state ahead of time, specifying the shaders, input and output data locations and data layouts to build a fixed description of the pipeline. In an article by Hidaka et al. (2017) they present an installation-free DNN (Deep Neural Network) execution framework called WebDNN to solve the issue of DNNs being computationally expensive and hardware-acceleration being required. WebDNN was implemented using WebGPU and was used to compare with a solution based on WebAssembly and WebGL and showed that the WebGPU-based solution outperformed the other solutions. Another article showing the efficiency of WebGPU was done by Poudel, Usher & Petruzza (2023) where they used WebGPU to stream, process and render point cloud data to effectively manage and render large amounts of point data, which is seen as an increasingly difficult task. The results of this experiment showed that using WebGPU for parallel data retrieval and rendering together with a multi-layer cache system yielded significant improvements, although the improvements of just WebGPU without the integrated cache system isn't known necessarily.

### 4.1.3 WebGL in E-commerce

Internet advertising in general has evolved drastically since the first piece of online advertisement in 1994 and using 3D to advertise products have been shown to increase the feeling of knowledge of the product as well as strengthen the impression of a brand over 3D counterparts since 2002 (Li, Daugherty & Biocca, 2002). Since then, e-commerce has grown to be a major part of the internet, accounting for trillions of dollars in sales (Hewawalpita & Perera, 2017). In a study done by Moritz (2010) three websites were used to analyze the emotional response from a participant, a website with static 2D models, a website with 2½D which is described as a VR-application realized as flash and a real-time 3D website was used. Moritz (2010) found that participants find real-time 3D web applications appealing because of their realism and interaction possibilities, though this perception diminishes when the product can't be manipulated in the form of configured or combined as the model only presented a computer-animated copy of reality. Moritz (2010) further lifts that too long loading periods and issues navigating can't be intuitively understood by all participants, especially those that are older. This lifts the idea of control and interactivity. The reason 3D can be beneficial in the area of e-commerce is that it can serve as a replacement for part of the experience in being able to physically manipulate and experience the product at hand, since we can't implement the ability to physically manipulate it, we can implement controls to represent interactivity which in this context is defined by Algharabat & Dennis (2010) as *"User ability to choose the content and form of the 3D virtual model, particularly: the ability to rotate and zoom in or out on the product, and to click on any part of the 3D virtual model and get instant information about*

*it; and the ability of the 3D virtual model to properly respond to the participant's orders"*. Further in the study done by Algharabat & Dennis (2010) they presented a website designed with Flash to illustrate an e-commerce website selling laptops using 3D visualization. Through the study they showed results that would indicate that implementing controls for the user to manipulate the 3D product strengthened their impression of the product being authentically real.

In a study done by Dethe & Joy (2023) an experimental survey was conducted on customers in e-commerce and showed that with product images 65.6% of participants did not understand the product and that some participants were interested in seeing the product In their own surroundings as well as a 360-degree view of the product. They proceeded explore the use of augmented reality in e-commerce and decided to design a website with 3D models where they used "model-viewer" which is hosted by Google on Github and uses Three.js for rendering the actual product at hand, which in turn is based on WebGL. Another study that showed a positive impact on customers by using 3D models to represent products was done by Geelhaar & Rausch (2015) who designed three websites for their experiment, an interactive 3D version based on WebGL, an interactive 3D version based on Flash and a static 2D version with images. The findings from this experiment were that most participants found that WebGL had positive emotional feedback and that the render quality of WebGL was almost photorealistic. The opposite of these findings was found with the Flash version, where users found themselves disappointed by the visual quality and hindered interaction performance.

## 4.2 JavaScript

JavaScript, commonly abbreviated as JS is the main language used for WebGL, hence why the following two libraries mentioned will end with ".js" to signify that they're written with JavaScript. JavaScript is a programming language that is considered at core as one of the most important languages for web-development in general, alongside HTML and CSS. It's therefore one of the most used programming languages for websites and continues to be included in an overwhelming majority of websites. According to Mehrara et al. (2011) approximately 95% of those browsing the internet using a web browser do so with JavaScript capabilities enabled. It was developed and released in the 1990s and has been used effectively in a wide array of different cases, from smaller scripts for creating menus to sophisticated applications that consists of thousands of lines of code, for example Gmail and Facebook which is enjoyed by millions of users (Mehrara et al, 2011). JavaScript works as a multi-paradigm language that supports event-driven, functional, object-oriented, and imperative programming styles with numerous additions like dynamic typing, prototype-based object-orientation and first-class functions. It runs with its own engine, the JavaScript engine where JavaScript code is parsed, optimized, and compiled at runtime by the engine in browser (Yan et al, 2021). It also handles its own garbage collection by automatically monitoring memory allocation and determine when a block of allocated memory is no longer in use and remove the data when necessary (Yan et al, 2021).

### 4.2.1 Libraries

JavaScript also supports the usage of external libraries, something that makes it possible for programmers to simplify the process and removes the necessity of writing all the code on their own (Ferrarezi et al, 2016) & (Bauer, Heinemann & Deissenboeck, 2012), which significantly reduces the effort for searching for possible computational errors (Borissova et al, 2021). This

means that developers don't need to write all the needed functionality for a program, from the low-level operating system to the high-level client interfaces, they can instead use features, functions and similar developed by others through not only libraries but also APIs for example (Palepu, Xu & Jones, 2013). There is a slight concern regarding the use of external libraries though, which is that they can significantly impact the maintainability of a software system (Bauer, Heinemann & Deissenboeck, 2012) as well as the decision-making issues that can appear when choosing a library or framework (Borissova et al, 2021). Three.js and Babylon.js are examples of external libraries.

### 4.2.2 Three.js

Three.js is a JavaScript library and application programming interface API for rendering and displaying 3D models and graphics in a web browser using WebGL and is capable of using all the information in the entire scene to render the models with great efficiency using shaders, sorting and frustum calling (Angel, 2017). Outside of this, it removes the necessity of the developer to understand and use any of the low-level details required in pure WebGL programming (Angel, 2017) which is beneficial as native WebGL programming is quite complex (Zhang et al, 2023). Its aim is to be an easily understood and used library while maintaining the ability to work cross-browser as well as remaining light-weight. According to an article done by Zhang et. al (2023) they noted that the primary benefit of Three.js is its extensibility, ease of use, robust scalability, and a variety of rendering techniques. Some noticeable similarities between the libraries exist, as they're both JavaScript libraries and can use WebGL for rendering the 3D models. An example of the similarities can be found when creating a new scene and applying a camera to it, the below example being from Three.js.

```
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth /
window.innerHeight, 0.1, 1000 );
```

**Figure 2** – Creating a scene in Three.js.

Another example of the similarities in syntax can be found when creating a simple sphere, the example below is also from Three.js.

```
const sphere = new THREE.SphereGeometry( 15, 32, 16 );
```

**Figure 3** – Creating a sphere in Three.js.

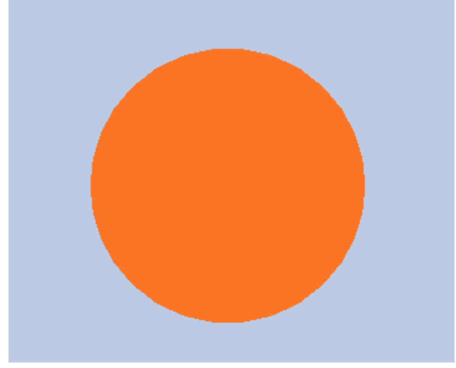

**Figure 4** – Sphere in Three.js.

**Figure 4** above is a demonstration of the result from executing the code in **Figure 2**, here with a color added in the same way as in **Figure 5**.

One noticeable difference in syntax and approach when it comes to adding light to your scene is that Three.js appears to come with more alternatives. Ambient which is a light that globally illuminates all objects equally, directional which is a light that gets emitted in a specific direction, hemisphere which is a light that is positioned above the scene, point which is a light that gets emitted in all directions, "rectarea" light which emits light in a rectangular plane and spotlight which is a light that gets emitted from a single point in one direction along a cone. Another noticeable difference is that Three.js supports the use of some external libraries and plugins to implement some functionality that isn't maintained by the core developers. For example, external physics via Omio.js or Enable3d, particle systems via three.quarks and three-nebula and GUI (Graphical User Interface) via dat.GUI or lil-gui or OrbitControls for controlling the camera with user input. One example of a difference in syntax is the approach to adding a texture to a 3D primitive, where in Three.js you create the box in a similar way to how to create it in Babylon.js, except you first create a geometry, then the material, then combine these.

```
var geometry = new THREE.BoxGeometry(1, 1, 1);
var material = new THREE.MeshBasicMaterial({color: '#FC7526' });
var cube = new THREE.Mesh(geometry, material);
cube.position.set(0, 0, 0);
scene.add(cube);
```

**Figure 5** – Creating a box with material in Three.js.

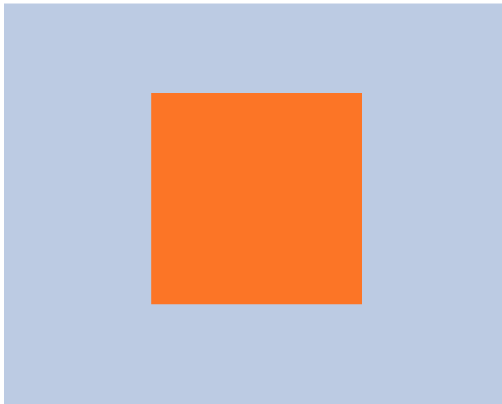

**Figure 6** – Box in Three.js.

**Figure 6** above demonstrates the results of the code from **Figure 5**, here we have a basic box presented over a background.

We can also add camera controls with "OrbitControls", to present the ability to move the cube, this is done by adding the code below.

```
const controls = new OrbitControls(camera, renderer.domElement);
function animate() {
  requestAnimationFrame(animate);
  controls.update();
  renderer.render(scene, camera);
}
animate();
```

**Figure 7** – Adding controls in Three.js.

The code in **Figure 7** adds the control as well as an animation loop to update the renderer and controls, with this we can move the cube and see the result below.

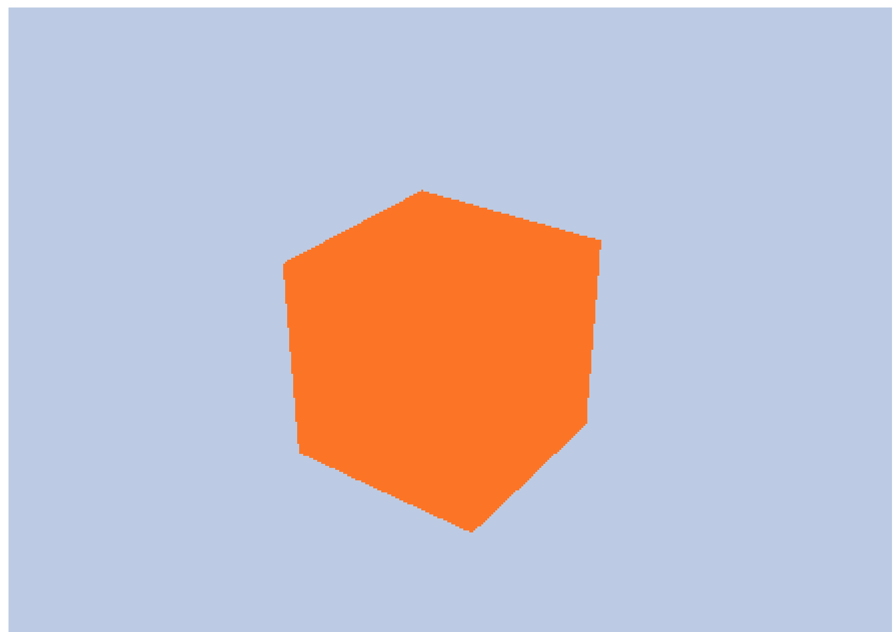

**Figure 8** – Manipulated box in Three.js.

### 4.2.3 Babylon.js

Babylon.js is another JavaScript library used to render and display 3D models and graphics in a web browser using WebGL. It has been used for everything from video games to crime data visualization and military training. Like three.js, it puts an emphasis on being powerful while remaining friendly and easy to use, but with a focus on video games according to Wöllman et al. (2020). A lot of technical functionality between both Babylon.js and Three.js remains the same, they both contain a scene graph, the ability to add textures, fog, shadows, lights, cameras, the ability to add a skybox and so on. They both also come with web-based editors where one can experiment and develop outside of their own application. One primary difference is that Babylon.js comes with its own game engine, with a collision engine, a physics engine based on Havok for the web and an audio engine based on Web Audio. So, while Babylon.js contains features not found in Three.js, Hirose, K. & Koba, M., notes that Babylon.js can take advantage of the physics simulations and makes it easy to develop 3D content and applications with WebGL. In the article by Shah et. al (2021) they mention the web-based editor tools as benefits to using the library, with both the Babylon sandbox for manipulating 3D models with a GUI and Babylon playground that contains a code editor and GUI to manipulate these models. This is also brought up in an article by Wöllmann et. al (2020) where the Babylon playground adds a level of ease of use by offering the ability to test functions in advance. They do denote something that can be held in contrast to the claim made by Li et al (2020) which is that files bigger than 10MB requires a long time to load and that Babylon.js has a limitation in vertices which is quickly exhausted with detailed models.

The example below is from Babylon.js, here we can see the similarities in how to initiate a new scene as well as a camera with the code from **Figure 2**.

```
const scene = new BABYLON.Scene(engine);
const camera = new BABYLON.ArcRotateCamera("camera", -Math.PI / 2, Math.PI /
2.5, 3, new BABYLON.Vector3(0, 0, 0), scene);
```

**Figure 9** – Creating a scene in Babylon.js

And the example here is from Babylon.js to see the similarity with the code found in **Figure 3**

```
const sphere = BABYLON.MeshBuilder.CreateSphere("sphere", {diameterX: 0.75, diam
eterY: 0.75, diameterZ: 0.75});
```

**Figure 10** – Creating a sphere in Babylon.js

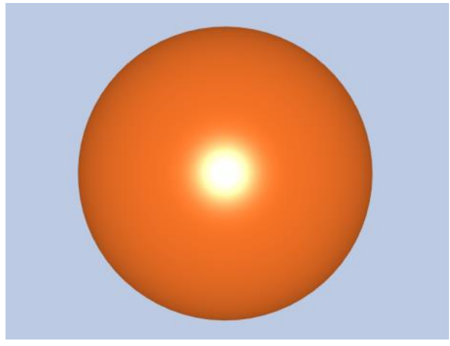

**Figure 11** – Sphere in Babylon.js.

**Figure 11** above demonstrates the results of executing the code found in **Figure 10,** here with material added in the same way as in **Figure 12.**

In contrast with **Figure 5**, in Babylon.js you can create and assign the material then add it as an attribute to the box instead of the approach from **Figure 5** where you combine these to create a box.

```
const box = BABYLON.MeshBuilder.CreateBox("box", {height: 1, width: 1, depth:
1});
var material = new BABYLON.StandardMaterial(scene);
material.diffuseColor = new BABYLON.Color3.FromHexString('#fc7526');
box.material = material;
```

**Figure 12** – Creating a box in Babylon.js.

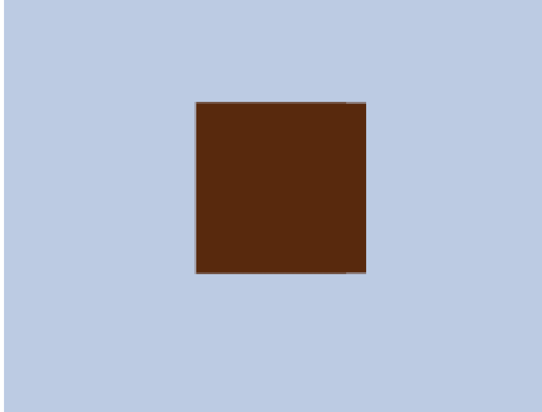

**Figure 13** – Box in Babylon.js.

**Figure 13** above demonstrates the results of the code from **Figure 12**, here we see a basic box, it does appear to be a different shade from its counterpart in **Figure 6** which is because Babylon.js introduces shadows with its "Hemispheric Light" while Three.js doesn't.

We can implement something akin to that demonstrated in **Figure 8** by using a different camera that Babylon.js comes with, namely the "ArcRotateCamera" that works in a similar way together to the code implemented in **Figure 7** but without implementing the math on moving the box.

```
const camera = new BABYLON.ArcRotateCamera("arcCamera", 0, 0, 10,
BABYLON.Vector3.Zero(), scene);
camera.attachControl(canvas, true);
scene.activeCamera = camera;
camera.target = box;
```

**Figure 14** – Adding controls in Babylon.js.

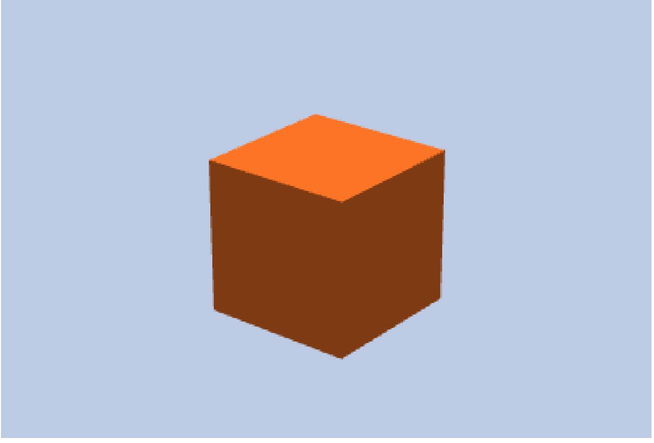

**Figure 15** – Manipulated box in Babylon.js.

**Figure 15** above demonstrates what happens when the code from **Figure 14** is executed and the mouse is used to move the camera, here we can note that the color difference mentioned in **Figure 13** is more obvious given the clear presence of shadows.

As mentioned in the chapter above, there are some differences when it comes to lights. Babylon.js only supports four different kinds of lights, point which is a light defined by a unique point in world space. A directional light that is defined by a direction and emitted from everywhere in the specified direction, a spotlight and finally a hemispheric light that works the same as their Three.js counterparts. There is also native support for mouse, keyboard, touch and gamepad input (Wöllman et al, 2017) which doesn't appear to be the case regarding Three.js where we used "OrbitControls" to handle the input as seen in **Figure 7**.

As mentioned previously, there is functionality that is only accessible via external libraries and plugins for Three.js, these covered physics, particle effects and GUI. Babylon.js comes with the ability to use Havoc for physics and solves the latter two on their own.

```
const particleSystem = new BABYLON.ParticleSystem("particles", 2000);
particleSystem.particleTexture = new BABYLON.Texture("textures/flare.png");
particleSystem.emitter = new BABYLON.Vector3(0, 0.5, 0);
particleSystem.start();
```

**Figure 16** – Creating a particle system in Babylon.js.

**Figure 16** contains an example of how to implement particle effects via what they call the "particle system". It can also implement GUI without external libraries and plugins, though it does support Dat.GUI just like three.js does, it also has other external options for this, for example CastorGUI. But the Babylon GUI comes equipped with the ability to generate an interactive user interface built based on their "DynamicTexture" class.

```
const UI = BABYLON.GUI.AdvancedDynamicTexture.CreateFullscreenUI("myUI");
const button = BABYLON.GUI.Button.CreateSimpleButton("button1", "Click!");
button.width = 0.1;
button.height= "50px";
button.color = "white";
button.background = "blue";
UI.addControl(button);
```

**Figure 17** – Creating a UI in Babylon.js.

**Figure 18** below demonstrates the result of the code from **Figure 17** being executed.

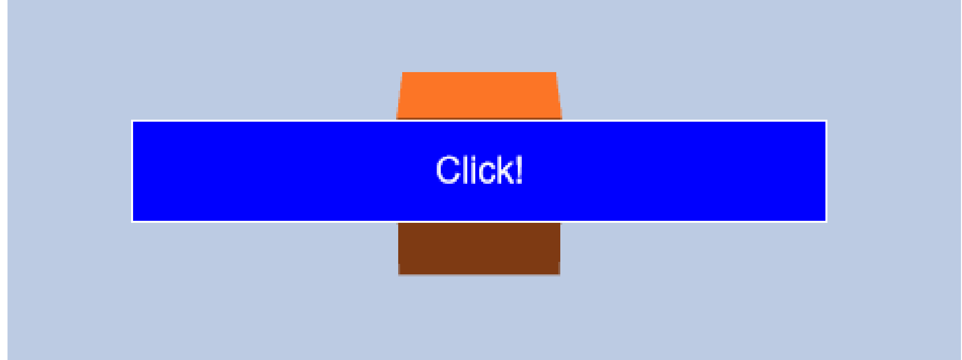


**Figure 18** – UI with button in Babylon.js.

## 4.3 Application

The application that is to be built to enable the experiment mentioned in "Problem" & Method" will be based on the examples provided by Dethe & Joy as well as Geelhaar & Rausch (2015), specifically the interactive 3D version based on WebGL. Both websites included a product overview with multiple products displayed next to each other as well as a detailed view on each product that was only found in the version by Gelhaar & Rausch (2015). This with the intent of having two aspects to test against, one with a detailed 3D model for a specific product with the possibility of interaction as well as a website with multiple 3D models, one for each product. This involves creating two semi-identical websites, one using Three.js and the other using Babylon.js, for rendering and presenting the 3D models. These will be implemented using core JavaScript functionality only, outside of that which is enabled through WebGL, WebGPU and the aforementioned libraries. Based on the findings by Algharabat & Dennis (2010) these rendered 3D models will include full control and functionality for the user to move around and manipulate the users' perspective of the model.

# 5 Problem

Previous studies have shown that visualizing products in 3D is both attention catching and evokes an emotional response in the user viewing the products, even showing a noticeable improvement in user satisfaction when viewing products rendered with WebGL in comparison to Flash-based applications (Geelhaar & Rausch, 2015) as well as an increase in the perceived product value when displayed with a 3D 360° view (Hewawalpita & Perera, 2017). But if we can to some extent conclude that the usage of 3D models in e-commerce has a positive effect on the user at hand, we need to analyze how to implement this properly.

To do this, we can look at the given tools for the implementation. This is important because of a study done by Zhou et al (2016) that shows that with an increase in response time in a mobile application a customer's subjective score of satisfaction goes down. This can be viewed in contrast with the study done by Li et al (2020) that lifts a primary issue with the existing WebGL-based rendering methods, which is that they cause excessive time delay due to one-time loading of model-data, they further explain that the engines they use for examples are three.js and babylon.js. Another study that lifts the issue of load-times was done by Moritz (2010) that brought up the issue of loading period for practical usability and understandability from older participants.

Previous research on the two libraries would show that there is a similarity in the practical application and usage of them, for example for virtual campus tours, where a conference paper done by Shah et al. (2021) would apply the babylon.js library to create a virtual campus tour, with the reasoning that it provides several important tools for the development, including the Babylon sandbox in which the writers could manipulate the parameters of the 3D model by using the included GUI and the Babylon playground which includes a code editor and GUI. This can be looked at in contrast to a conference paper done by Panchal et al (2022) that aimed to create a 3D model of a college campus using three.js, where the usage of three.js was to enable the 3D scene while models were done externally. Both conference papers apply one of the two aforementioned libraries with the goal of creating something akin to each other, showing a similarity in their practical application.

The problem with WebGL is that load times can potentially be too long. This can be a result of both underwhelming CPUs (Central Processing Unit) as well as network speeds. This article will compare the practical technical differences between these two popular libraries, three.js and babylon.js with the primary goal of measuring the performance of WebGL and WebGPU. The addition as a potential solution to the issue with WebGL comes from previous research showing the efficacy of WebGPU by Dyken et al. (2022), Hidaka et al. (2017) and Poudel, Usher & Petruzza (2023). The reason these two libraries were chosen was because of the previous lack of experience with working with similar libraries, the selection was partly made by analyzing the GitHub repositories of various WebGL libraries that both supported 3D and had recent commits, assuming that this means better support. It's also based on the arguments presented by Li et al. (2020) that used both libraries to lift underlying issues with the WebGL rendering method. Another core reason was Babylon.js having the capability of using WebGPU and Three.js showing potential ability to use WebGPU. They also present examples that would imply similar application usage in practice, where babylon.js presents examples on their website of companies and applications using the library, for example Nike, Microsoft, IBM and Mojang. These examples include smaller models which is what three.js also include in their examples.

## 5.1 Research Question

**RQ1**: What is the difference in performance regarding load-times and frames per second between WebGPU and WebGL, represented by Babylon.js and Three.js when used to render and present 3D models?

## 5.2 Hypothesis

The hypothesis for this study is that Babylon.js using WebGPU will outperform both Babylon.js using WebGL and Three.js using WebGL.

The null hypothesis is therefore that Babylon.js using WebGPU won't outperform either Babylon.js using WebGL nor Three.js using WebGL.

# 6 Method

There are two different research paradigms to approach empirical studies, exploratory research which is concerned with studying an artifact in a normal setting and deriving findings based on observing the artifact in question as well as explanatory research which is concerned with analyzing a relationship between two points or to compare these with the aim of identifying a cause-effect relationship (Wohlin et al, 2012). The primary approach of this paper will be explanatory, as the paper looks to discover and analyze relationships between the usage of WebGL or WebGPU through two different libraries and see the effect of these relationships. The reason of the approach is that Wohlin et al. (2012) argues that exploratory research, also known as quantitative research, is appropriate when testing the effect of some manipulation or activity and the thesis aims to directly manipulate variables in the form of library and API this is the most appropriate. There are also different techniques and methods for approaching research, these being a survey, a case study and an experiment. A survey is used to collect information from or about people to describe, compare or explain their knowledge, attitudes and behavior (Wohlin et al, 2012). A case study in software engineering is explained by Wohlin et al (2012) as an empirical enquiry that draws on multiple sources of evidence to investigate one instance. This method works by collecting data to research a potential project. An experiment is explained by Wohlin et al (2012) as an empirical enquiry that manipulates one factor or variable in a studied setting. This is done by controlling and keeping other variables constant and seeing the effects of only changing one of them. The latter, an experiment, being the most applicable as the thesis aims to analyze the difference after changing some variables around. The reason it's the most applicable is that a survey will be impossible to analyze and fetch information regarding the performance of WebGPU or WebGL with.

## 6.1 Alternative Methods

I want to present and argue for other methods and why they're not equally applicable in this instance. Qualitative research is a form of research where fewer amount of data points are typically collected but usually of higher quality, for example in terms of interviews, where they can be argued to be higher quality than digital surveys. Both surveys and case studies can be both quantitative research and qualitative research while experiments are strictly quantitative. The inherent reason as to why qualitative research isn't applicable is that to measure statistical differences in the performance between Three.js and Babylon.js using WebGPU and WebGL, multiple data-points will be required to analyze and find mathematical differences. While qualitative answers in the form of user experience, perspective and interpretation of the performance is interesting, it doesn't fall within the aim of this survey.

A survey as mentioned earlier is a method where information from or about people is collected, usually through interviews or questionnaires. This isn't necessarily applicable for the given context as researching pure performance is hard if based on user feedback. While users can contribute to findings by presenting their perspective on the performance, there are multiple issues that can be faced using this approach. A user might not be able to tell the difference between differences in frames per second nor loading time, these variables can present differences that aren't necessarily noticed by a user, but still statistically significant. If the goal was to research the users' interpreted performance of either WebGPU or WebGL, this would be more relevant, although that isn't the goal for this thesis.

A case study is interesting but not applicable as it revolves around investigating an artifact in its actual context, this means that the context must be real usage. In this article, the aim is to develop a website that simulates an e-commerce store, this doesn't reflect real world usage as it's inherently a simulation of an actual artifact, not the artifact itself. A case study is also a form of observational study, which is a type of study where something is observed, for example the usage of an e-commerce store, without any attempts to affect the usage or outcome of the usage. As there will be an attempt to change the outcome by changing the API and library that is used to render and present the 3D models while running this on a local machine instead of in actual realistic usage, a case study can be deemed inefficient to measure the difference in performance.

## 6.2 Chosen Method

The primary methodology for this paper will be experimental, manipulating the library and related APIs used to present the same model in the same environment. To perform this analysis, a website will be developed with the intent of being presented as an online store, where a sample product will be presented in the form of a 3D model and the libraries facilitating this presentation will change. To further investigate the difference, the browser Google Chrome will be used to analyze the relationship between the libraries and API to find quantitative differences. The primary methodology of analyzing the data will be done through the lens of a quantitative study, meaning that the data used for the analysis will be in the form of basal values to statistically compare. The primary values analyzed will be load-time and FPS (frames per second). While analyzing CPU usage, GPU usage, and memory usage can also be relevant, it can be difficult to measure and interpret without external libraries and functionality. Analyzing load-time and FPS is perfectly viable within the tools presented to us through Google Chrome as well as with baseline JavaScript with the aforementioned APIs and libraries. To analyze load-time over multiple environments, throttling will be used to simulate lower network capabilities. This approach was applied by Li et al. (2020) in their study on WebGL optimization strategies. Google Chrome is also one of the few browsers that currently have full support for WebGPU. Another study that applied throttling of network speed to a fixed variable was done by Sheng et al. (2017) where they fixed the loading times to 10Mbps and 5Mbps to analyze the FPS as well as load-time.

This creates some inherent ethical issues as this essentially keeps analyzation to Google Chrome only, which isn't necessarily representative of the entirety of all browsers that exist. The usage of this will remain to stay in line with Li et al. (2020) as well as Sheng et al. (2017), while throttling network speeds is possible through admin panels for various routers, this presents an external level of difficulty and potential issues when it comes to the generated data to be analyzed later. It also remains the case that Google Chrome is one of few browsers that innately support WebGPU, while others support it, it must be manually enabled. This results in Google Chrome being able to support both WebGPU natively without modification as well as being able to handle throttling of network speeds with a custom profile instead of pre-defined settings. This gives full control of the network speeds to present findings that are in relation to other findings, more accurate, but less accurate than real world usage. This means that a case study would present other findings that could be more align with realistic usage but remains inapplicable for the given study. While this remains a dilemma, my approach aims to find the most statistically significant and accurate data while all ran over the exact same equal and standardized point instead of the most accurate data while ran in a real usage setting.

## 6.3 Ethics & Validity

### 6.3.1 Technical

To avoid potential issues regarding the validity of the data, the collection of data values will be performed on a computer running a freshly installed version of Windows, with only the browsers in question having been installed and without any potential underlying processes being ran in the background and all these data values will be collected on the same machine under the same premises. The specifications of the computer this experiment will take place on are as follows:

| Motherboard: | TUF GAMING Z590-PLUS WIFI |
|---|---|
| Graphics Processing Unit: | NVIDIA GeForce RTX 3070 |
| Central Processing Unit: | Intel Core i7-10700 @ 2.90Ghz, 8 Core |
| RAM-Memory: | 32 GB DDR4 @3600Mhz |

For the specifications of the libraries, the following versions were used for the pilot study as well as the final experiment.

| Three.js: | 0.162.0 |
|---|---|
| Babylon.js: | 6.48.1 |

### 6.3.2 Environmental

A secondary environmental purpose of this study will be to ease potential use of physical stores in favor of digitalized stores with 3D models to accurately show products at hand, this would lessen environmental impact of these stores as less area is needed to show off the products at hand. There is also a need to analyze which library and API is superior for the task at hand, to establish that it works across multiple environments in regard to network speeds and computer hardware possibilities, to avoid having to spend time developing strategies to handle users potential worse or better environments.

### 6.3.3 Sustainability

Another reason as to why these libraries are used is because of the popularity of these libraries, presenting more information to be found on these libraries in the form of documentation, forums, books and research articles. With more information readily available, it becomes easier to develop applications with these libraries in hand. That means that for future developers, it’s easier to develop and maintain applications that have already been created, lessening the aspect of development time to uphold the system in question.

### 6.3.4 Personal Integrity

As there are no products there is no copyright-issues regarding the study, nor will there be any other individuals included and therefore no personal information saved anywhere during the experiment and writing of the study.

### 6.3.5 GitHub

All code presented in this study as well as the code for the website including the usage of these aforementioned libraries will be published on GitHub for external validation of readers and for potential future research.

# 7 Literature

For the implementation of the software, there are two primary sources of information to learn and understand these libraries and how to work with 3D models on the web. These two sources are also recommended to anybody interested in the field and to further do experiments using these libraries, or simply recreate the experiment and the implementation presented within this study. The first source is “Learn Three.js: Program 3D Animations and Visualizations for the Web with HTML5 and WebGL” by Jos Dirksen. For this thesis, the basis was in the third edition of this book, although there is a fourth edition that one can imagine is equally as interesting with more modern functionality. This book goes through the absolute basics and further develops on this understanding with reoccurring code-examples as well as images to illustrate what the code is supposed to present for the user to visualize. It does a good job of explaining what each line of code does but does present a slight issue of being overly specific some parts. While the actual content that is presented is great it does spend a considerable amount of time describing functionality within the code that is based on the code alone easily understood by somebody with a foundational understanding of JavaScript. The second source was for understanding Babylon.js, a book named “Babylon.js Essentials: Understand, Train, and Be Ready to Develop 3D Web Applications/video Games Using the Babylon.js Framework, Even for Beginners” by Julien Moreau-Mathis. This book serves as a healthy resource for learning and understanding Babylon.js although unlike the previously mentioned book this one expects a deeper understanding of JavaScript and programming alike, it also doesn’t go into the same depth of alternatives the developer has at his disposal. For example, the chapter regarding cameras and lights doesn’t explain the different alternatives but simply shows an example implementation of what can be perceived as the most common option. Though it doesn’t face the same issue as the previously mentioned book, here the code examples are allowed to speak for themselves and there isn’t an overly intricate explanation of every line of code.

Outside of these two primary books that is recommended for the reader of this study to read to further understand these libraries, a lot of information was retrieved through using search engines for posts on various forums, for example the currently commonly used Stack Overflow for coding-related questions. The queries to search for relevant information was always started by using the library at hand, so Babylon.js or Three.js, then the actual topic at hand. Only if relevant information wasn’t found was there further specification of the issue in the search term. For example, to handle the code within **Figure 33** and subsequently discussed within “Progression” under the title “TransformNodes” the used search terms were “Babylon.js debugging”, “Babylon.js” debug tool”, “Babylon.js nodes”, “Babylon.js TransformNodes” and so on until presented with the documentation covering specifically TransformNodes.

One of the primary tools for the implementation was using the actual documentation, the documentation for both libraries is vast and very easily understood as they often contain examples with bode the code and the visual aspects. The documentation was what served me the most purpose during the implementation as a lot of time was spent traversing it to find solutions to problems and struggles during the implementation phase.

# 8 Implementation

Both implementations in Three.js and Babylon.js are based on the exact same HTML file, excluding the <title> tag which changes name depending on the version.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="style.css">
  <title>Thesis Three.js</title>
  <style>
  </style>
</head>
<body>
  <div id="sidebar">
    <img id="logo" src="logo.png">
    <div id="menu">
      <div class="menu-buttons">
        <button onclick="window.location.href='index.html'">Products</button>
        <button
onclick="window.location.href='information.html'">Information</button>
      </div>
    </div>
  </header>
  </div>
  <div id="content">
    <div class="canvas-container">
    </div>
  </div>
  <script type="module" src="script.js"></script>
</body>
</html>
```

**Figure 19** – Code from index.html.

The HTML-file shown in **Figure 19** includes the necessary tags such as “<html>”, “<head>” & “<body>”, while trying to keep it as minimalistic as possible as to not affect the final measurements. To maintain the similarities between this experiment and an actual e-commerce website, an icon is included along with a simplistic menu to navigate between a “Products” page and an “Information” page. The only interesting aspect of this is the “<div>” with the class “canvas-container” which will contain the generated canvases to hold each model, these will be created and initialized within the JavaScript code.

```
  <div id="content">
    <div class="product-canvas-container">
    </div>
  </div>
  <script type="module" src="productscript.js"></script>
```

**Figure 20** – Individual differences in product.html.

For the individual products, an almost exact replica of the code shown in **Figure 19** was created, with slightly altered class names and a different JavaScript file. These changes are present in **Figure 20**.

## 8.1 Data

To keep track of all the models as well as creators and referencing for these in compliance with legal requirements a file was created called "modelinformation" that holds all relevant information. The usage of the JSON format instead of a database is based on responses to a StackOverflow thread, one response by Barmar stating that if there is no need for complex queries and the entire data set will be loaded, JSON is most likely simpler and more efficient and one response by DougM stating that using a raw file format like JSON is preferrable if none of the benefits of using a database provides is necessary and the entire data set will be loaded (Barmar, 2013; DougM, 2013).

```
[
  {
    "id": 1,
    "name": "Doodles",
    "path": "models/doodles.glb",
    "references": [
      {
        "text": "Doodles",
        "href": "https://sketchfab.com/3d-models/rtfktchallenge-doodles-rtfkt-
rtfkt-bb7da6cc59894a95b8c302937711e174"
      },
      {
        "text": "Eduardo Kuhn",
        "href": "https://sketchfab.com/eduardokuhn"
      }
```

**Figure 21** – Example of model in modelinformation.json[1].

As seen in **Figure 21**, the JSON is structured as an array containing multiple objects, each object representing a model. It consists of basal information regarding the name and path as well as all referencing the models. Additional referencing information is added in JavaScript.

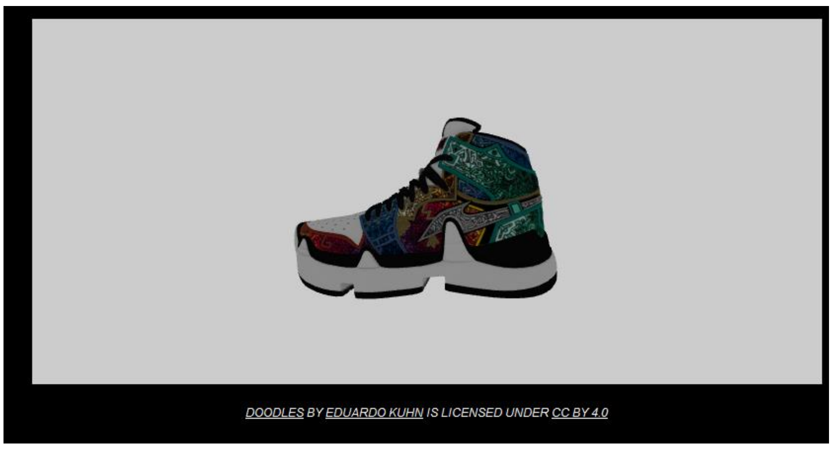


**Figure 22** – Model in product.html.

**Figure 22** is an example of a model presented with the corresponding reference with source, creator and licensing information.

[1] Implemented in #78c71e7 (Babylon.js) & #83009b6 (Three.js), finished in #775842b (Babylon.js) & #b24e91e (Three.js).

## 8.2 Artifact

The artifact section will contain the details regarding the implementation of both the Three.js version as well as the Babylon.js version of the final website.

### 8.2.1 Three.js

The Three.js implementation will contain two different scripts, one for index.html shown within **Figure 19** and one for product.html mentioned in **Figure 20**. The latter of the two has very few differences and only these will be mentioned.

```
function generateCanvases() {
  fetch('modelinformation.json')
    .then(response => response.json())
    .then(modelData => {
      modelData.forEach((model, i) => {
        createCanvases(model, i);
      })
    })
}
```

**Figure 23** – Function generateCanvases()[2] to handle modelinformation.json.

The Three.js implementation is initiated through a "window.onload" function that calls the function "generateCanvases()", within this function the JSON-file that was presented within **Figure 21** is retrieved through a network request and once the file is fetched successfully the response is iterated over for each model within the file to call the "createCanvases()".

```
function createCanvases(model, index) {
  const canvasId = `canvas${index + 1}`;
  const canvasWrapper = document.createElement('div');
  canvasWrapper.classList.add('canvas-wrapper');
  const canvas = document.createElement('canvas');
  canvas.classList.add('product-canvas');
  canvas.id = canvasId;
  canvas.width = 800;
  canvas.height = 370;
  canvasWrapper.appendChild(canvas);
  const text = document.createElement('div');
  text.classList.add('product-canvas-text');
  text.textContent = model.name;
  canvasWrapper.appendChild(text);
  canvasContainer.appendChild(canvasWrapper);
  canvas.addEventListener('click', function () {
    window.location.href = `product.html?name=${model.name}`;
  });
  initializeThree(canvasId, model.path);
```

**Figure 24** – Function createCanvases()[3] to handle creation of canvases.

[2] First implementation in #b36d0be, final implementation in #9f7ca9c
[3] Final implementation in #9f7ca9c

The code within **Figure 24** represents "createCanvases()" which takes in relevant parameters and creates a "<canvas>" element for each of these models. This canvas is initialized as well as the variable "canvasId" which will function as the ID of each canvas. Outside of these elements an "EventListener" is added to listen for clicks on each of these individual models, with the hypertext reference leading to "product.html", shown in **Figure 22**.

```
function initializeThree(canvasId, modelPath) {
  const canvas = document.getElementById(canvasId);
  const scene = new THREE.Scene();
  scene.background = new THREE.Color(0xCCCCCC);
  const camera = new THREE.PerspectiveCamera(75, canvas.clientWidth /
canvas.clientHeight, 0.1, 1000);
  camera.position.set(0, 0, 35);
  const renderer = new THREE.WebGLRenderer({ canvas: canvas, antialias: true });
  renderer.setSize(canvas.clientWidth, canvas.clientHeight);
  renderers[canvasId] = renderer;
  const light = new THREE.HemisphereLight(0xffffff, 0xffffff, 1);
  light.position.set(0, 10, 20);
  scene.add(light);
  dispatchPossibleFlags[canvasId] = true;
  initializeModel(modelPath, scene, camera, canvasId);
}
```

**Figure 25** – Function initializeThree()[4] to set up Three.js.

In the code within **Figure 25** we fetch the correct element and initialize a scene that will hold the model. A camera is implemented of the type "PerspectiveCamera" which is designed to mimic the way the human eye works, as in being the human perspective (Three.js, 2024). This camera is initialized in a way to present the entire scene. A renderer is initialized using the WebGL API that takes in parameters for what scene to render, then it's saved to an array of renderers that will become relevant later. Finally, the "dispatchPossibleFlags" flag corresponding to the given canvasId is initiailized to true, for the FPS Tracking script.

```
function initializeModel(modelPath, scene, camera, canvasId) {
  const loader = new GLTFLoader();
  loader.load(modelPath, function (gltf) {
    loadedModels++;
    const model = gltf.scene;
    scaleModel(model);
    adjustCamera(model, camera);
    scene.add(model);
    models[canvasId] = model;
    animate(model, renderers[canvasId], camera, scene, canvasId);

    if (loadedModels == 12) {
      window.dispatchEvent(new CustomEvent('allModelsLoaded'));
    }
  });
}
```

**Figure 26** – Function initializeModel()[5] to load and render models.

[4] First implementation in #b36d0be, refactored function name, final implementation in #2605e48
[5] First implementation in #cb3fdd4, refactored to individual function, final implementation in #9f7ca9c

The code within **Figure 26** is the function handles the actual fetching and loading of the individual models from the "modelinformation.json" file shown within **Figure 21**. This is done by initializing a loader of the type "GLTFLoader". The model is then initialized to variable "model" by calling for the entire "gltf.scene", this is to have a working variable that represents all aspects of the scene that was loaded. These models are saved to an array in a similar fashion to the array "renderers" within **Figure 25**. Afterwards the "loadedModels" variable increase in value for each model and when it reaches 12, which is the amount of models to be loaded, the "CustomEvent" named "allModelsIsLoaded" is dispatched which is used for the script Load Time Tracker.

```
function scaleModel(model) {
  let targetHeight = 0.2
  const boundingBox = new THREE.Box3().setFromObject(model);
  const size = boundingBox.getSize(new THREE.Vector3());
  const scaleFactor = targetHeight / size.y;
  model.scale.set(scaleFactor, scaleFactor, scaleFactor);
}
```

**Figure 27** – Function scaleModel()[6] to scale models.

The code within **Figure 27** handles scaling the models, this function works by calculating the bounding box of the model, which is an invisible box around the model that envelops the model in question. This is because after the downloading of the models, the models appeared to be of different sizes, although maintaining the same ratio. This way all the models will scale down or up depending on their initial size, so that all models will hold the same size.

```
function adjustCamera(model, camera) {
  const boundingBox = new THREE.Box3().setFromObject(model);
  const center = boundingBox.getCenter(new THREE.Vector3());
  const size = boundingBox.getSize(new THREE.Vector3());
  const maxDim = Math.max(size.x, size.y, size.z);
  const distance = maxDim * 1;
  camera.position.set(center.x, center.y, center.z + + distance);
  camera.lookAt(center);
}
```

**Figure 28** – Function adjustCamera()[7] to position camera correctly.

The code within **Figure 28** displays the "adjustCamera". This is implemented in a similar fashion to the code within **Figure 27**. Except the center is calculated of the bounding box based on the model to aim the camera towards as well as the size of the bounding box is calculated to apply a fitting distance of the camera from the model.

[6] First implementation in #83009b6, refactored name, final implementation in #9f7ca9c
[7] First implementation in #83009b6, refactored name, final implementation in #9f7ca9c

```
function animate(model, renderer, camera, scene, canvasId) {
  requestAnimationFrame(() => animate(model, renderer, camera, scene,
canvasId));
  if (loadedModels == 12) {
    model.rotation.y += 0.008;
  }
  renderer.render(scene, camera);
  if (window.fpsTrackerActive && loadedModels == 12 &&
dispatchPossibleFlags[canvasId]) {
    const fpsEvent = new CustomEvent('logFPS', { detail: { value: getFPS() } });
    window.dispatchEvent(fpsEvent);
    dispatchPossibleFlags[canvasId] = false;
    setTimeout(() => {
      dispatchPossibleFlags[canvasId] = true;
    }, 1000);
  }
}
```

**Figure 29** – Function animate()[8] to handle model rotation and external scripts.

The code within **Figure 29** displays the "animate" function, this function has the core role of handling the animations. This is achieved by implementing an "requestAnimationFrame" that will repeatedly call "animate()", essentially creating a loop. Within this function, the "model.rotation.y" value is increased to create the rotation, this is only called after all models have been loaded, using the same variable as mentioned under **Figure 26**. Here one of the externals scripts come into play with the if-statement that checks if the FPS Tracker is active, if all models are loaded and if the individual logging of FPS is possible, the latter is to make sure that all models aren't logging constantly to avoid overfilling the measurements and checking more regularly than the FPS has a possibility to alter. Within this if-statement, the "CustomEvent" logFPS is dispatched with the value of the "getFPS()" function.

```
function generateCanvas(modelName) {
  fetch('modelinformation.json')
    .then(response => response.json())
    .then(modelData => {
      const rightModel = modelData.find(row => row.name === modelName);
```

**Figure 30** – Function generateCanvas()[9] for individual models.

**Figure 30** displays one of the differences in the index.html version using script.js to show the catalogue of options and the individual choice that you reach by clicking on the models as mentioned under **Figure 24**. Here there is not a need for all the models, instead searching for the correct one based on the URL parameter.

[8] First implementation in #be4323b, final implementation in #7718bc8
[9] First implementation in #b24e91e, final implementation in #b24e91e

```
function initializeModel(modelPath) {
  const loader = new GLTFLoader();
  loader.load(modelPath, function (gltf) {
    model = gltf.scene;
    scaleModel(model);
    adjustCamera(model, camera);
    scene.add(model);
    renderer.render(scene, camera);
  });
}
```

**Figure 31** – Function initializeModel()[10] for individual models.

Because of the usage of individual models, there is not a necessity to save these to an array. The same holds true for renderers as shown in **Figure 25**.

### 8.2.2 Babylon.js

The chapter regarding the implementation of the Babylon.js version will follow the same structure as the Three.js chapter, except the similar parts will not be mentioned. A lot of the code is similar to maintain a fair and equal comparison.

```
function initializeBabylon(canvasId, modelPath) {
  const canvas = document.getElementById(canvasId);
  const engine = new BABYLON.Engine(canvas, true);
  const scene = new BABYLON.Scene(engine);
  const camera = new BABYLON.UniversalCamera("UniversalCamera", new
BABYLON.Vector3(0, 5, -10), scene);
  scene.clearColor = new BABYLON.Color3.FromHexString('#cccccc');
  const light = new BABYLON.HemisphericLight("light", new BABYLON.Vector3(0, 1,
0), scene);
  dispatchPossibleFlags[canvasId] = true;
  initializeModel(canvasId, modelPath, scene, camera, engine)
}
```

**Figure 32** – Function initializeBabylon()[11] to set up Babylon.js.

The “initializeBabylon()” function which is the Babylon.js version of the code shown within **Figure 25**. Here there are some noticeable differences, primarily that the initialization of scene and camera takes in more parameters but less total lines of code, a similar principle was discussed previously under **Figure 12**. Here an engine, scene, camera and lights are declared with the goal of finding similar approaches to the approach in Three.js, where “UniversalCamera” is described as “is the one to choose for first person shooter type games”, which is similar to “PerspectiveCamera” as in being the users’ point of view (Babylon.js, n.d. a).

[10] Final implementation in #b24e91e
[11] First implementation in #e196f13, refactored name, final implementation in #775842b

```
function initializeModel(canvasId, modelPath, scene, camera, engine) {
  BABYLON.SceneLoader.ImportMesh("", modelPath, "", scene, function (meshes) {
    let model = scene.transformNodes.find(node => node.name ==
"Sketchfab_model");
    if (model) {
      loadedModels++;
      models[canvasId] = [model];
      scaleModel(model, 1);
      adjustCamera(model, camera);
      model.rotationQuaternion = null;
      model.rotation.x = -Math.PI / 2;
      model.rotation.y = -Math.PI / 2;
      animate(scene, canvasId, engine);
      if (loadedModels == 12) {
        window.dispatchEvent(new CustomEvent('allModelsLoaded'));
      }
    }
  })
}
```

**Figure 33** – Function initializeModel()[12] in Babylon.js to load and render models.

One noticeable difference shown in **Figure 33** is how the loader works, while in the Three.js version shown within **Figure 26** allowed the declaration of loaded models onto a variable, Babylon.js works differently. Here a search for the specific nodes within the model must be done to find the correct "parental" node to save as the variable. There is also a necessity to make sure that "rotationQuaternion" is set to null. Both cases will be discussed under the header "Progression". After loading in these models, they were also angled differently, hence the need to modify the "model.rotation" of both x and y.

```
function scaleModel(model, targetSize) {
  let boundingBox = model.getHierarchyBoundingVectors(true);
  let size = boundingBox.max.subtract(boundingBox.min);
  let maxDimension = Math.max(size.x, size.y, size.z);
  let scaleFactor = targetSize / maxDimension;
  model.scaling = new BABYLON.Vector3(scaleFactor, scaleFactor, scaleFactor);
}
```

**Figure 34** – Function scaleModel()[13] in Babylon.js to scale models.

The "scaleModel()" function within **Figure 34** is the equivalent of the function shown in **Figure 27.** Here there is a slight difference in calculation, where the size is calculated by taking the largest vector and subtracting the smallest vector.

```
function adjustCamera(model, camera) {
  let boundingBox = model.getHierarchyBoundingVectors(true);
  let center = boundingBox.min.add(boundingBox.max).scale(0.5);
  camera.position = new BABYLON.Vector3(center.x + 1.6, center.y, center.z);
  camera.setTarget(center);
}
```

**Figure 35** – Function adjustCamera()[14] in Babylon.js to position camera correctly.

[12] First implementation in #78c71e7, final implementation in #f354416
[13] First implementation in #78c71e7, final implementation in #5796dad
[14] First implementation in #78c71e7, final implementation in #5796dad

The “adjustCamera()” function within **Figure 35** is the equivalent of the function shown in **Figure 28**. This required fewer lines of code as the camera positioning didn’t present a similar issue, the necessity is just to set the camera to the center of the model.

```
function animate(scene, canvasId, engine) {
  engine.runRenderLoop(function () {
    if (models[canvasId] && loadedModels == 12) {
      models[canvasId].forEach(rootMesh => {
        rootMesh.rotation.y += 0.008;
      });
    }
    scene.render();
    if (window.fpsTrackerActive && loadedModels == 12 &&
dispatchPossibleFlags[canvasId]) {
      const fpsEvent = new CustomEvent('logFPS', { detail: { value: getFPS() }
});
      window.dispatchEvent(fpsEvent);
      dispatchPossibleFlags[canvasId] = false;
      setTimeout(() => {
        dispatchPossibleFlags[canvasId] = true;
      }, 1000);
    }
  });
}
```

**Figure 36** – Function animate()[15] in Babylon.js to handle model rotation and external scripts.

The “animate()” function in Babylon.js shown within **Figure 36** is remarkable simple, except the necessity to call for an “engine.runRenderLoop” instead of the approach with “requestAnimationFrame” as done in Three.js, shown within **Figure 29**.

```
async function initializeBabylon(canvasId, modelPath) {
  const canvas = document.getElementById(canvasId);
  const engine = new BABYLON.WebGPUEngine(canvas);
  await engine.initAsync();
```

**Figure 37** – Function initializeBabylon()[16] with WebGPU to set up Babylon.js with WebGPU.

The only difference in the version using WebGPU is that the function to initialize Babylon.js is running asynchronously and there is an “await” for the engine as is necessary according to documentation on the matter (Babylon.js, n.d. b)

[15] First implementation in #e196f13, final implementation in #775842b
[16] Implemented in #841233f

## 8.3 Measurement Scripts

The following scripts are the external scripts used via TamperMonkey to measure variables necessary to conclude an answer to the research questions.

### 8.3.1 FPS Tracker

The FPS Tracker is used to calculate the number of frames per second (FPS) for each model over a specific span of time.

```
function Loop() {
    window.requestAnimationFrame(() => {
        const now = performance.now();
        while (times.length > 0 && times[0] <= now - 1000) {
            times.shift();
        }
        times.push(now);
        fps = times.length;
        Loop();
    });
}
```

**Figure 38**– Function Loop()[17] to handle looping of FPS Tracker.

The function in **Figure 38** handles the actual calculation of FPS, this equation is based on the post on StackOverflow by "Daniel Imms" that wrote an article about it, it's explained as using "window.requestAnimationFrame" to fire the "Loop()" function on every animation frame, the function then adds the current timestamp to the queue "times" and removes timestamps that occurred more than a second ago from the front of the queue. The FPS is then calculated by counting the size of the queue (Imms, 2017).

```
window.getFPS = function() {
    return fps || 0;
}

function addFPSData(name) {
    FPSData.push({ name: name, value: getFPS() });
}

window.addEventListener('logFPS', function(e) {
    addFPSData(e.detail.name);
});
```

**Figure 39** – Fetching and adding FPS[18].

The code within **Figure 39** contains the following, "window.getFPS" is used to make a function global, it simply returns the FPS for usage in further functions. This implementation to make a function global comes from the StackOverflow account "Etheryte" that explained that TamperMonkey scripts are run in a separate scope, so to make a function global you can use this approach (Etheryte, 2014). "addFPSData" is used to push the current return from "getFPS()" to an array named "FPSData" and "window.addEventListener" is used to listen for

[17] First implementation in #43a8cad, final implementation in #be2a293
[18] First implementation in #43a8cad, final implementation in #ddbfd56

a custom event that is dispatched within the primary script in Three.js and Babylon.js as shown in **Figure 29** and **Figure 36** respectively.

```
setTimeout(() => {
    window.fpsTrackerActive = false;
    const resultsJSON = JSON.stringify(FPSData, null, 2);
    const blob = new Blob([resultsJSON], { type: 'application/json' });
    const downloadLink = document.createElement('a');
    downloadLink.href = URL.createObjectURL(blob);
    downloadLink.download = 'results.json';
    downloadLink.textContent = 'Download Results';
    document.body.appendChild(downloadLink);
    downloadLink.click();
    document.body.removeChild(downloadLink);
}, 10000);
```

**Figure 40** – Timeout()[19] to download data and end script.

The code within **Figure 40** is used to download the actual measured FPS in the JSON format. It translates the saved measurements in FPSData into JSON and then downloads this file, this is implemented using "JSON.stringify" to translate the contents and "Blob" to create the file to download. This file is then appended to a created element that is initiated by calling "click" on it, which initializes the download.

```
104,
104,
104,
104,
104,
119,
144,
144,
144,
144,
```

**Figure 41** – Example Data.

The downloaded file by the name "results.json" will look like **Figure 41** above, showing the value of the FPS.

[19] First implementation in #7d1cde6, final implementation in #2ee8048

### 8.3.2 Load Time Tracker

The Load Time Tracker is used to calculate how long it takes for the Three.js and Babylon.js implementations to load all the models. It's implemented to be ran using TamperMonkey or GreaseMonkey and saves the loading time in milliseconds to then be downloaded in the JSON format after a specific amount of reloads and measurements have been completed. To initialize the script its based on an immediately invoked function expression, which is a function that starts itself, within it necessary variables gets declared and the "measure()" function ran which is pictured below.

```
function measure() {
    let startTime = performance.now();
    window.addEventListener("allModelsLoaded", () => {
        let endTime = performance.now();
        let loadTime = endTime - startTime;
        save(loadTime);
    });
}
```

**Figure 42** - Function Measure()[20] to measure load and render time.

The code within **Figure 42** presents the "measure()" function which does the actual calculation of the total load time. This is implemented by initializing a "startTime" variable which checks the current time as the script loads in and awaits a custom event named "allModelsLoaded", this custom event is fired from the actual Three.js or Babylon.js implementations after all models have successfully been loaded as shown in **Figure 26** and **Figure 33** respectively. The time after this event is fired is initialized as "endTime" and the variable "loadTime" is initialized with the value of "endTime" minus the "startTime", to get a result showing the total time it took before the event "allModelsLoaded" is fired. The value is then sent as a parameter to the "save()" function.

```
function save(loadTime) {
    let loadTimes = GM_getValue('loadTimes', []);
    loadTimes.push(loadTime);
    GM_setValue('loadTimes', loadTimes);
    if (loadTimes.length >= runs) {
        download();
    } else {
        reload();
    }
}
```

**Figure 43** – Function Save()[21] to save measurements.

The "save()" function presented within **Figure 43** is used to save the actual measured data to a "GM_Value" system, which is TamperMonkey and GreaseMonkeys version of Local Storage. The "loadTimes" array is declared and initialized as the value from the fetched "GM_Value" containing the loading times from previous reloads of the website. The array is then set as the value of the "GM_Value". After the declaration and saving of the value, a check is done to see how long this given array is, this is to make sure an exact number of measurements are collected, the check is implemented to control if the number of measurements taken is less than or equal to the amount of exact measurements to be taken. If the amount of

[20] First implementation in #57672d9, final implementation in #168aee3
[21] First implementation in #57672d9, refactored name, final implementation in #bedcac7

measurements taken correlate to the goal amount then the function "download()" is called, otherwise the function "reload()" is called.

```
function reload() {
    setTimeout(() => {
        window.location.href = url;
    }, 5000);
}
```

**Figure 44** – Function Reload()[22] to reload website.

The "reload()" function simply opens the website again by calling "window.location.href" to a previously saved variable containing the URL to the website. After the timeout presented in the **Figure 44** was implemented, this decrease disappeared.

```
function download() {
    let results = GM_getValue('loadTimes', []);
    const resultsJSON = JSON.stringify(results, null, 2);
    const blob = new Blob([resultsJSON], { type: 'application/json' });
    const downloadLink = document.createElement('a');
    downloadLink.href = URL.createObjectURL(blob);
    downloadLink.download = 'results.json';
    downloadLink.textContent = 'Download Results';
    document.body.appendChild(downloadLink);
    downloadLink.click();
    document.body.removeChild(downloadLink);
    GM_setValue('loadTimes', []);
}
```

**Figure 45** – Function Download()[23] to download stored measurements.

The "download()" function presented within **Figure 45** is implemented in a similar fashion to the one within the FPS Tracker as shown in **Figure 40**, with the addition of fetching the values from "GM" and removing the saved information afterwards.

```
[
  4157,
  4249,
  4205,
  4287,
  4295
]
```

**Figure 46** – Example Data.

The then downloaded file named "results.json" contains all the measurements, the contents is exemplified in **Figure 46** above.

[22] Final implementation in #57672d9
[23] First implementation in #57672d9, final implementation in #0faa223

## 8.4 Progression

This chapter will cover some of the issues and conflicts noted during the implementation phase, to ease similar implementations for future attempts in other experiments.

### 8.4.1 TransformNodes

One problematic issue that was found was during the implementation of the function "initializeModel()" in Babylon.js, seen in **Figure 33** was how to reach the elements loaded in externally to the scene.

```
let model = scene.transformNodes.find(node => node.name == "Sketchfab model");
```

**Figure 47** – TransformNodes[24] example.

As seen in **Figure 47**, this is the example solution. To approach finding this solution, the built-in debugging tool from Babylon.js was used as well as documentation on how nodes are interpreted in Babylon.js.

```
scene.debugLayer.show();
```

**Figure 48** – Debugging tool.

The debugging tool as previewed in **Figure 48** works innately on each scene, opening an external menu in the website that can be used to traverse the node-tree and get a deeper understanding of what is being rendered within the scene.

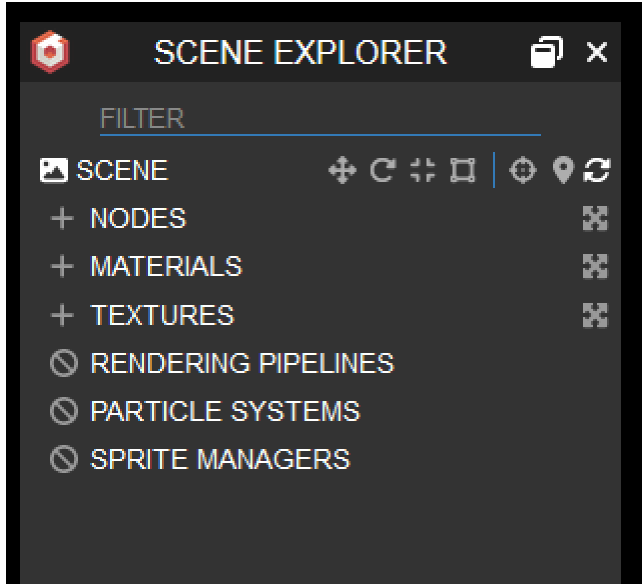


**Figure 49** – Debugging menu.

[24] Solved in #135bdf6

The debugging menu as seen above within **Figure 49** is what will be presented to you after implementing the code from **Figure 48**. Here the user can in real time manipulate values and see the changes from these manipulated values. This menu was used to traverse the node-tree and discover that one of the primary nodes was named "Sketchfab_model", which was then used to declare to the variable model as seen in **Figure 47**. It's worth noting that the "transformNodes" refers to the type of node in question, while there was a node above the "Sketchfab_model" node, this one was of the type "Mesh" and direct manipulation of it didn't yield any working results.

### 8.4.2 Quaternion

Another interesting issue that was solved by using the menu presented in **Figure 48** was the handling of quaternions, which is explained in documentation as a class used to store quaternion data (Babylon.js, n.d. c). Essentially, quaternions are a number system that extends complex numbers and is used in three-dimensional space. This was above my understanding of mathematics but the issue that arose was that with this turned on, normal rotation wouldn't work. It's explained in the documentation regarding rotation quaternions that these cannot be used in combination with a traditional rotation as this will produce wrong orientation or incorrect results unless "rotationQuaternion" is set to null. The issue arose as they describe it because many imported models already have a pre-defined "rotationQuaternion".

```
model.rotationQuaternion = null;
```

**Figure 50** – Code line rotationQuaternion[25] to turn off "rotationQuaternion".

**Figure 50** shows the exact line that is used to turn "rotationQuaternion" off, this is done within the "initializeModel()" function in Babylon.js shown in **Figure 33**.

### 8.4.3 Scaling Models

One of the issues that took up most of the development time was scaling each individual model to fit the same size relative to other models. My initial thought was to use a wide array of different models but scaling these to somehow all fit within the same sizing and distance from camera without some being perceived as too small or too big was quite difficult. The implementation of a solution to handle this can be seen within **Figure 27** and **Figure 34**, for Three.js and Babylon.js respectively. The implemented solution of using the bounding box to calculate relative size works, except individual measurements having to be tried and tested in case there is a difference in model sizing for future experiment. But this is a necessity to consider.

### 8.4.4 GM_Value

For the usage of TamperMonkey/GreaseMonkey scripts, reloads might be necessary but for security reasons localStorage isn't usable from these environments. This is explained under **Figure 43** and **Figure 45**. Essentially, the "GM" system works similarly, where "GM_setValue" is used to allow values to persist across page loads and origins (Greasespot, 2018a) and "GM_getValue" is used to retrieve a value set with "GM_setValue" (Greasespot, 2018b).

[25] Solved in #78c71e7

# 9 Pilot Study

The pilot study will consist of an example of the eventual experiment being conducted to assess the possibility of generating usable measurements and establish potential differences before going into the primary generation. A simple set of measurements will be taken, where the FPS Tracker will run for one minute and the Load Time Tracker will run for 50 runs, this will be done across the Three.js implementation using WebGL, the Babylon.js implementation using WebGL and the Babylon.js implementation using WebGPU. This will only be done on the catalogue of data, so not the individual models as well. No statistical analysis will be done on this data, but simply a visual interpretation, this will also not be done with throttling on network speeds but simply ran through a localhost using the build tool Vite.js.

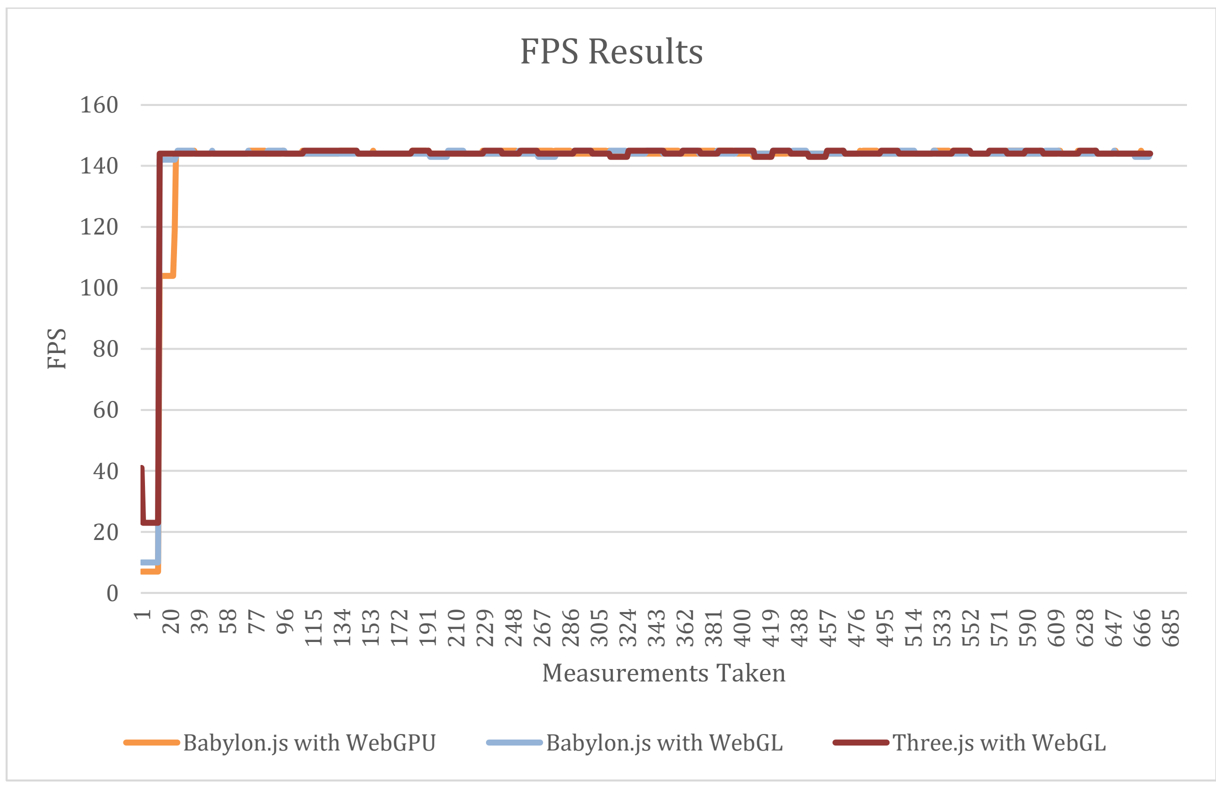


**Figure 51** – FPS Results in Pilot Study.

The results shown in **Figure 51** is taken from the FPS Tracker and shows some interesting results. Primarily that they all reach the 144 FPS max which is the refresh rate that my screen can tolerate, this presents an issue in statistical analysis and presents the necessity to use other artifacts or methods to collect measurements, this will be handled in the discussion later. One other interesting result is with Babylon.js using WebGPU is that it takes a little bit longer for the average canvas with model to reach the highest possible FPS before it stabilizes at that value. Although there are some minor differences in the Babylon.js with WebGL and Three.js with WebGL, the Babylon.js with GPU measurements appear to be more consistent after reaching the possible top value.

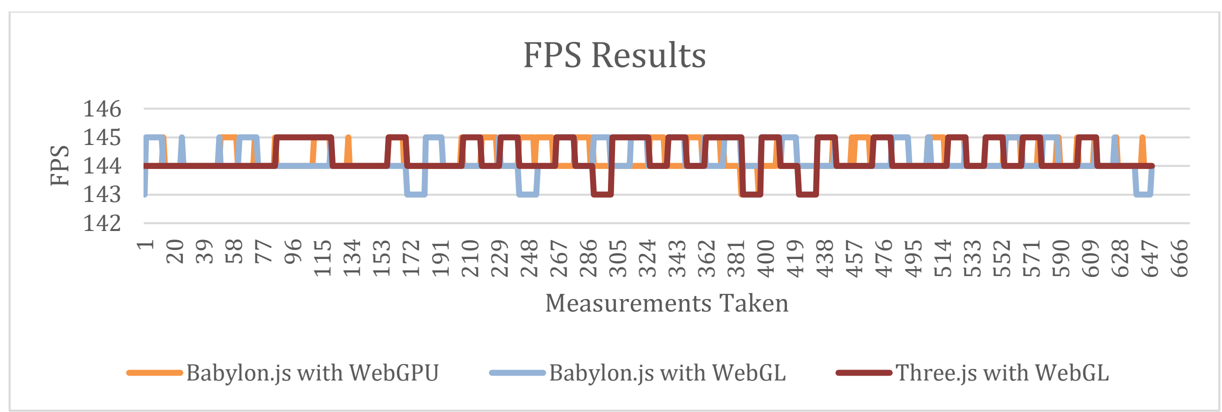


**Figure 52** – FPS Results in Pilot Study with earlier measurements removed.

The diagram in **Figure 52** is to exemplify the point that Babylon.js with WebGPU seems from a visual standpoint to be the most stable, only showing one dip below 144 FPS while Babylon.js using WebGL and Three.js using WebGL showing multiple more dips below 144 FPS. Although these aren't necessarily statistically significant since the dip is only from 144 to 143. These measurements are based on values collected after all three versions have reached 144 FPS for the first time, which means the first 23 measurements were removed from all versions.

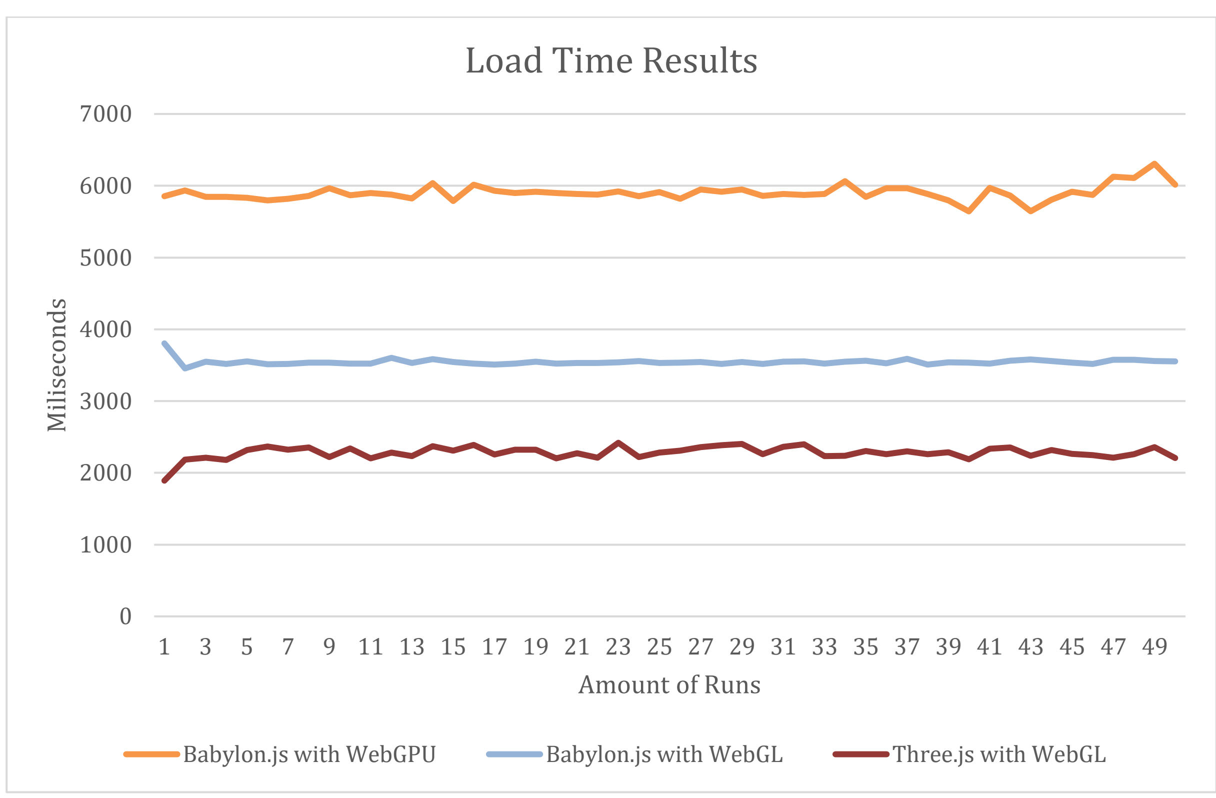


**Figure 53** – Load Time Results in Pilot Study.

The results from the test using the Load Time Tracker shows some interesting results that correlates to the results found in **Figure 51**, here we can see that Babylon.js using WebGPU takes a longer time to load all models, which correlates to the difference in how long it takes to come up to the same maximum FPS relative to other measurements. There is also a distinct difference between Three.js with WebGL and Babylon.js with WebGL, where Three.js shows a much faster loading time.

## 9.1 Discussion

The results presented within **Figure 51** shows a very limited difference in measurements; to specify, the measurements were taken with the system implemented to read one canvas at a time and then continue with reading the next one on a time-based lockout as presented within **Figure 29** and **Figure 36**. This means that there is a constant checking but not an overflowing of data. The slowdown showed in **Figure 50** presented around 100 FPS showed through data that looks like the one within **Figure 54** below. This showed that it took the equivalent time of 10 measurements to go from 104 FPS to 119 FPS, afterwards it went up to 144 FPS. These results are inconclusive and presents an issue for future statistical analysis. To solve this potential problem, an additional artifact of measurement will be introduced in the form of another computer of lower performance to take measurements from, potentially to see dips in quality. There is also the possibility to test with higher quality models, something that has been implemented with the "modelinformationHQ" version that is the same as what is presented within **Figure 21** but with models consisting of 2k and 4k textures instead of the 1k textures tested against in this pilot study.

```
104,
104,
104,
104,
104,
104,
104,
104,
104,
104,
119,
```

**Figure 54** – Example measurements from FPS Tracker.

# 10 Analysis

The analysis will use a similar approach to the Pilot Study although with changes to time and number of measurements taken as well as taking measurements from the individual versions mentioned within **Figure 30** and **Figure 31**. All these measurements were gathered while having caching turned off to make sure that each run is a realistic representation of an initial opening of the website. To maintain similarity to the experiments mentioned in the Method chapter throttling was implemented at 100MB/s, following the implementation of with Li et al. (2020) as well as Sheng et al. (2017). This number was decided to maintain a speed enough to mathematically be able to generate enough measurements to conclude an answer to the research question while also being a realistic value of network speeds.

## 10.1 FPS Values

Based on the findings from the Pilot Study, specifically the results presented within **Figure 52** it was obvious that to generate measurements that are statistically interesting and capable of being analysed properly, something had to be changed. The initial idea was to introduce another computer in the form of a laptop, although this proved futile as it demonstrated an equally unvaluable result.

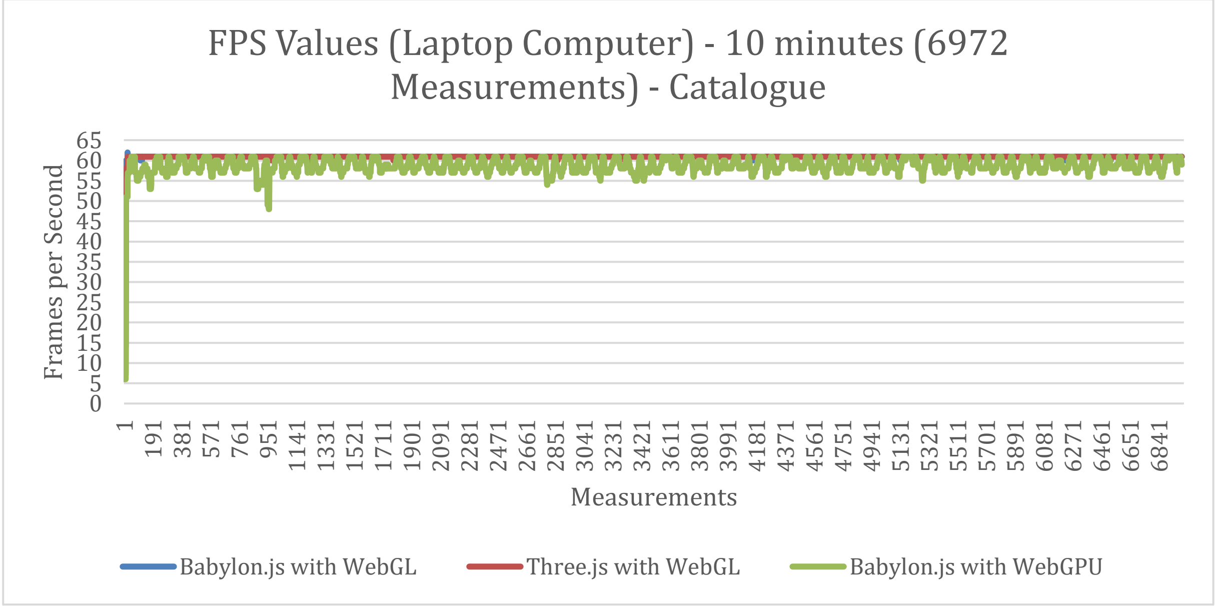


**Figure 55** – FPS Values from Laptop Computer.

The line chart within **Figure 55** shows that Babylon.js with WebGPU consistently dips below the maximum 60 FPS, which is the maximum value based on the refresh rate of the monitor. This isn't necessarily statistically significant as there is a massive cut-off of measurements based on the refresh rate alone. This led to investigating different methods of approaching this, specifically how to disable the frame rate limit in Google Chrome.

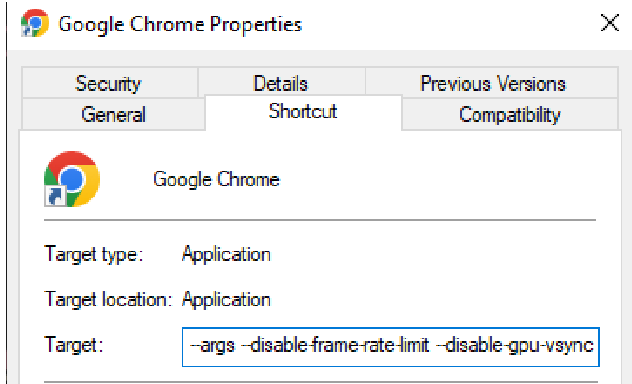


**Figure 56** – Adding custom properties to Google Chrome.

Based on a post on the Google Chrome Help forums by a user named "Mete", this is accomplished by adding commands to the properties of a Google Chrome shortcut, specifically "—args --disable-frame-rate-limit --disable-gpu-vsync" to the end of target property as shown above in **Figure 56** (Mete, 2019).

Using this information, this was implemented on the desktop version using the same specifications as mentioned previously within 5.3.1 "Technical". These measurements were gathered over a period of 20 minutes for each combination of library and API.

### 10.1.1 Catalogue Version

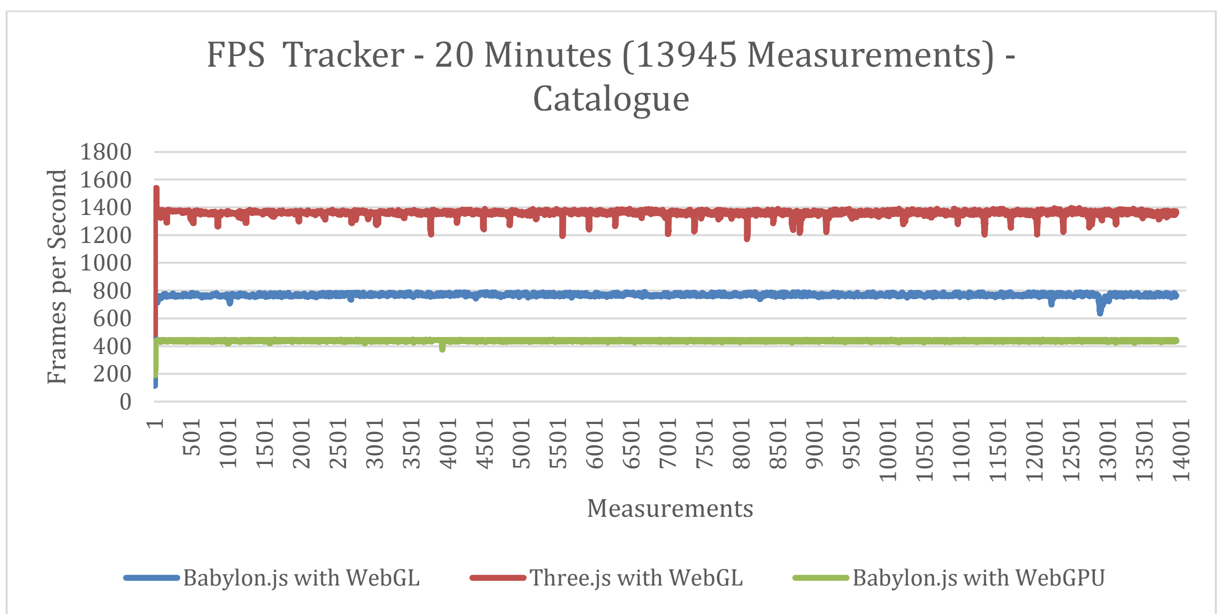


**Figure 56** – Line Chart of FPS Values (Catalogue).

The line chart within **Figure 56** represents the final measurements taken using the approach mentioned below **Figure 56**, this ran over a course of 20 minutes and generated in total 13945 measurements for each of the three artifacts. There is a drastic difference between the results shown, where Three.js with WebGL performs consistently better than the other artifacts although presents more dips than the other alternatives. Babylon.js using both APIS performs worse than Three.js with WebGL, although Babylon.js with WebGL performs better than with WebGPU. Babylon.js with WebGPU performs the worst in relation to the others, although maintains a much more consistent result without many dips.

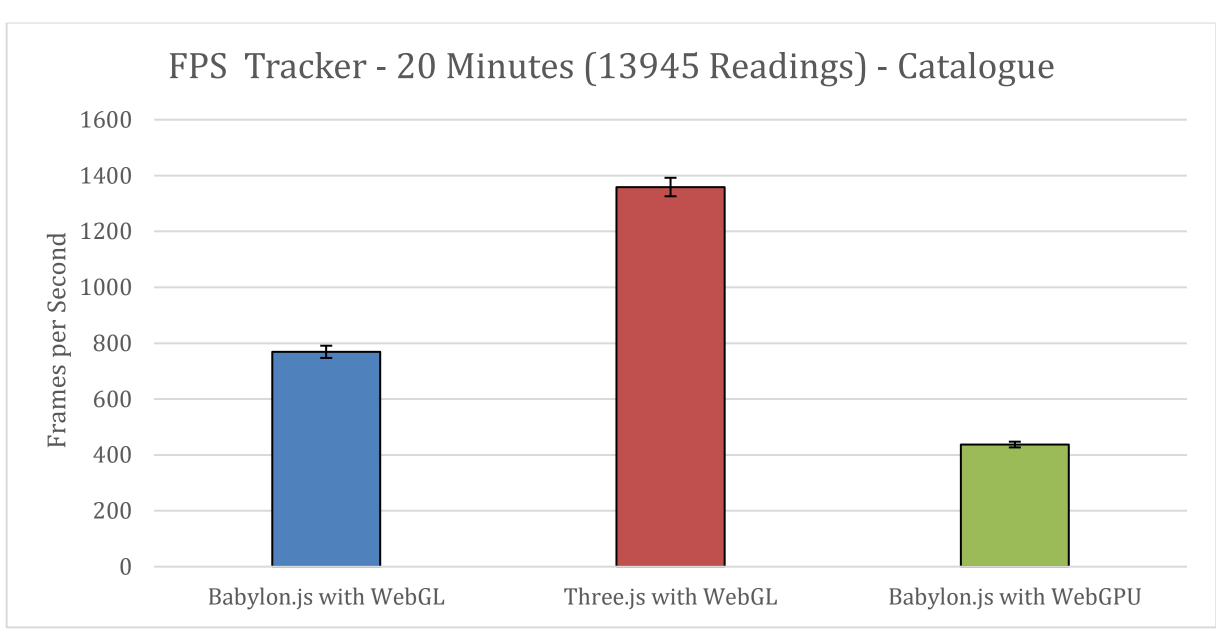


**Figure 57** – Bar Chart of FPS Values (Catalogue).

The bar chart within **Figure 57** presents the same findings as **Figure 56**, although with error bars to represent the standard deviation. Here the claim that Babylon.js with WebGPU remains more consistent holds true based on the standard deviation, while Three.js with WebGL has a higher standard deviation, represented by the regular dips within **Figure 56**. These measurements resulted in the Anova p-value $< 0.001$ which indicates that it's statistically significant, showing that Three.js with WeBGL performs the best in regard to frames per second with Babylon.js with WebGPU performing noticeably worse in comparison to the other two.

| **Babylon.js with WebGL** | | **Three.js with WebGL** | | **Babylon.js with WebGPU** | |
|---|---|---|---|---|---|
| Mean | 769,0953 | Mean | 1359,027 | Mean | 437,2086 |
| Standard Error | 0,185257 | Standard Error | 0,280286 | Standard Error | 0,087179 |
| Median | 769 | Median | 1363 | Median | 436 |
| Mode | 768 | Mode | 1363 | Mode | 436 |
| Standard Deviation | 21,8768 | Standard Deviation | 33,09872 | Standard Deviation | 10,29488 |
| Sample Variance | 478,5945 | Sample Variance | 1095,525 | Sample Variance | 105,9845 |
| Range | 676 | Range | 1052 | Range | 256 |
| Minimum | 113 | Minimum | 487 | Minimum | 190 |
| Maximum | 789 | Maximum | 1539 | Maximum | 446 |
| Sum | 10725034 | Sum | 18951632 | Sum | 6096874 |
| Count | 13945 | Count | 13945 | Count | 13945 |

**Figure 58** – Statistical Data for FPS Tracker (Catalogue)

The statistical analysis presented within **Figure 58** presents that all measured combinations of libraries and APIs have the same amount of measurements, it also supports the claim that Babylon.js with WebGPU had the lowest standard deviation, remaining the most consistent of the artifacts.

| Group 1 | Group2 | Meandiff | P-adj | Lower | Upper | Reject |
|---|---|---|---|---|---|---|
| Babylon.js WebGL | Babylon.js WebGPU | -331.886 | 0.00 | -332.712 | -331.060 | True |
| Babylon.js WebGL | Three.js WebGL | 589.931 | 0.00 | 589.106 | 590.757 | True |
| Babylon.js WebGPU | Three.js WebGL | 921.818 | 0.00 | 920.992 | 922.644 | True |

**Figure 59** – Tukey HSD Data for FPS Tracker (Catalogue).

While the Anova test presented a P-value of less than 0.001, indicating that this data is statistically significant the scope of the thesis is to analyze differences specifically between WebGPU and WebGL, a Tukey HSD test was done to conclude differences between each of these measured variables. This test was done using a Python script with the statsmodels module and specifically the "pairwise_tukeyhsd" function included in said module. This Tukey HSD test was done with an alpha value of 0.01, meaning a 1% risk of producing a false positive, but also 99% confidence in the results. This same approach will be performed for all future measurements where a Tukey test will also be used to analyze differences. The findings from this test indicate that the null hypothesis is rejected showing that there is significant difference when comparing Babylon.js with WebGL to Babylon.js with WebGPU, Babylon.js with WebGL compared to Three.js with WebGL and with Babylon.js with WebGPU compared to Three.js with WebGL. Showing a statistically significant difference in each of these comparisons, the conclusion can be drawn that all these groups are different in comparison to each other.

### 10.1.2 Individual Version

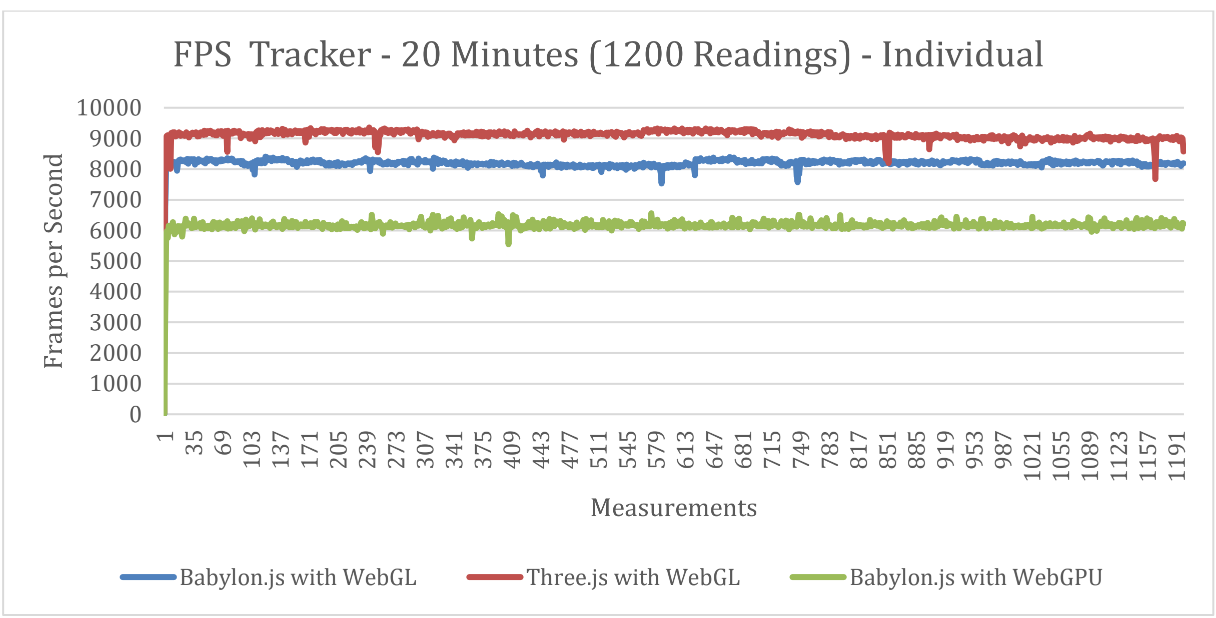


**Figure 60** – Line Chart for FPS Values (Individual).

The measurements from the individual models present the same finding as **Figure 56** does although with a less noticeable difference, where the difference between Three.js with WebGL and Babylon.js with WebGL is barely noticeable, with dips in the measurements in Three.js with WebGL going below the average measurement for Babylon.js with WebGL.

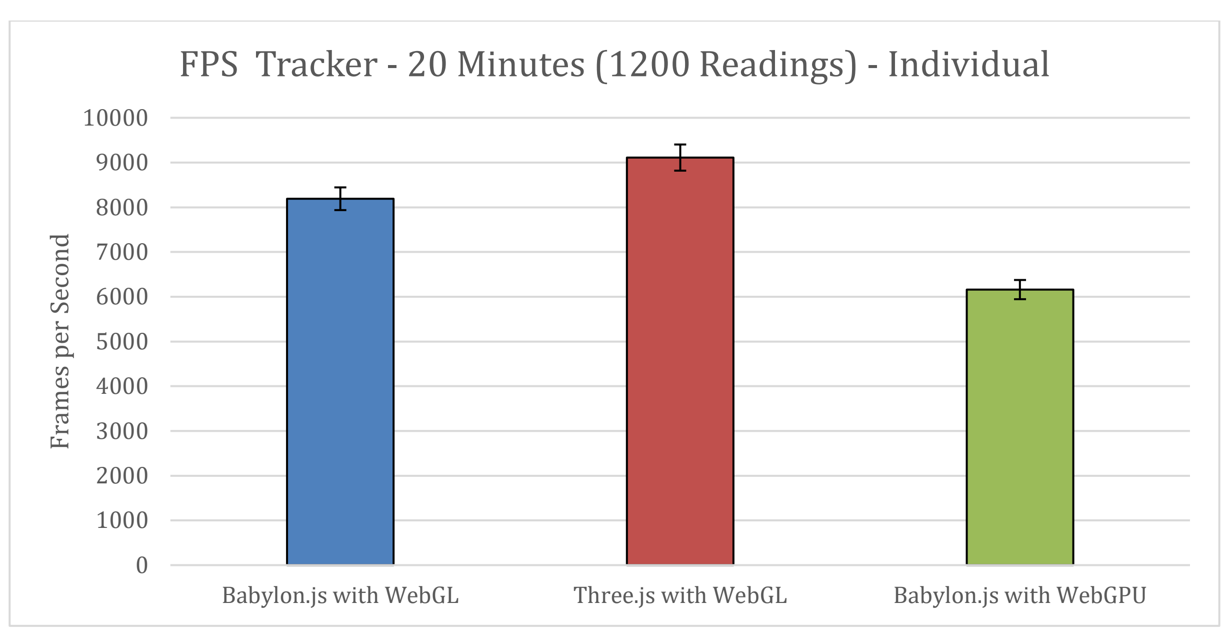


**Figure 61** – Bar Chart for FPS Values (Individual).

**Figure 61** presents the findings from **Figure 60** with the addition of standard deviation in the form of error bars, as done in **Figure 57** as well. Here the difference in the bottom part of the error bars for Three.js with WebGL moves much closer to the top part of the error bars for Babylon.js with WebGL. Though this still presents an Anova P-value of < 0.001, making the findings statistically significant.

| **Babylon.js with WebGL** | | **Three.js with WebGL** | | **Babylon.js with WebGPU** | |
|---|---|---|---|---|---|
| Mean | 8190,56 | Mean | 9111,139 | Mean | 6159,814 |
| Standard Error | 7,330714 | Standard Error | 8,44523 | Standard Error | 6,195635 |
| Median | 8204 | Median | 9140 | Median | 6154 |
| Mode | 8229 | Mode | 9080 | Mode | 6147 |
| Standard Deviation | 253,9434 | Standard Deviation | 292,5514 | Standard Deviation | 214,6231 |
| Sample Variance | 64487,24 | Sample Variance | 85586,3 | Sample Variance | 46063,07 |
| Range | 8398 | Range | 9297 | Range | 6542 |
| Minimum | 6 | Minimum | 58 | Minimum | 15 |
| Maximum | 8404 | Maximum | 9355 | Maximum | 6557 |
| Sum | 9828672 | Sum | 10933367 | Sum | 7391777 |
| Count | 1200 | Count | 1200 | Count | 1200 |

**Figure 62** – Statistical Data for FPS Tracker (Individual).

The statistical analysis within **Figure 62** presents the basis of **Figure 60** and **Figure 61**, here we can see the statistical difference from the catalogue version presented within **Figure 58**. One noticeable difference is that the standard deviation on average is much higher than in the catalogue version, but the standard deviation remains the lowest for Babylon.js with WebGPU. Here it’s also noticeable the mean value for Babylon.js with WebGL and the mean value for Three.js with WebGL is much closer than in the catalogue version.

| Group 1 | Group2 | Meandiff | P-adj | Lower | Upper | Reject |
|---|---|---|---|---|---|---|
| Babylon.js WebGPU | Babylon.js WebGL | 2030.745 | 0.00 | 2000.313 | 2061.178 | True |
| Babylon.js WebGPU | Three.js WebGL | 2951.325 | 0.00 | 2920.892 | 2981.757 | True |
| Babylon.js WebGL | Three.js WebGL | 920.579 | 0.00 | 890.147 | 951.011 | True |

**Figure 63** – Tukey HSD Data for FPS Tracker (Individual).

The results from this Tukey HSD test presents similar findings as for the catalogue version presented within **Figure 59**, showing that for all comparisons within the three artifacts the null hypothesis was rejected, and a significant difference was presented.

## 10.2 Load & Render Time Values

For the load and render tracker, it collected 300 measurements instead of the 50 as done in the Pilot Study, this was under the same configuration as mentioned previously.

### 10.2.1 Catalogue Version

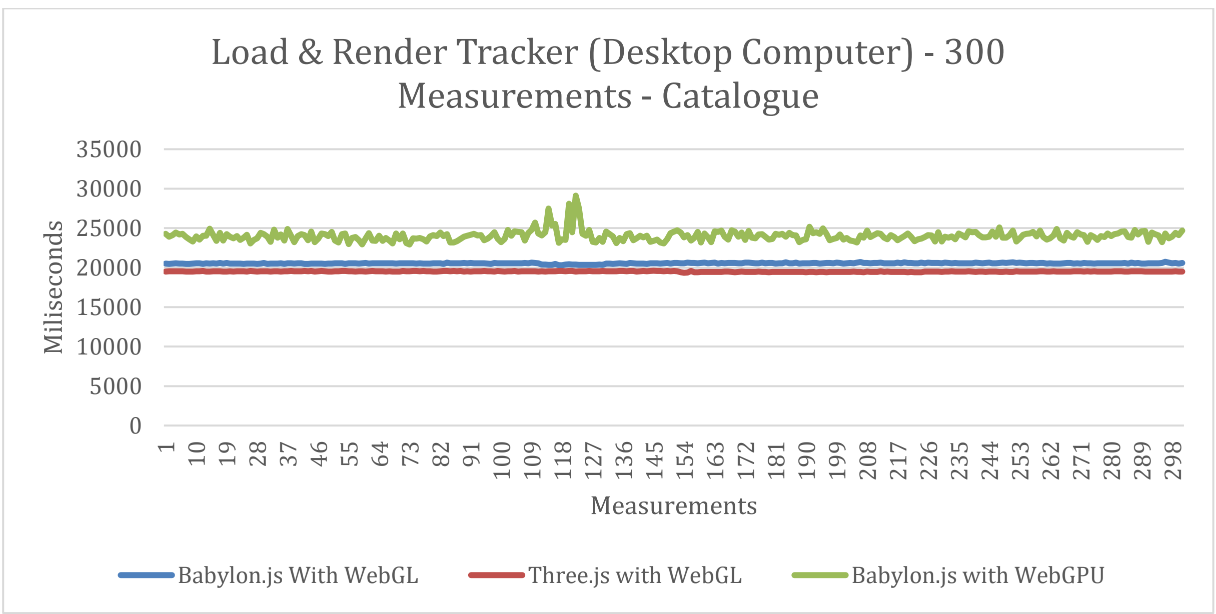


**Figure 64** – Line Chart of Load & Render Values (Catalogue).

For the results presented in **Figure 64**, there is a much smaller change than the one regarding FPS values presented within **Figure 56** and **Figure 57**. Although there is a noticeable difference with Babylon.js with WebGPU showing much less consistent results, going both up and down consistently. This can be compared to both Babylon.js with WebGL and Three.js with WebGL maintaining a relatively consistent value throughout the 300 measurements.

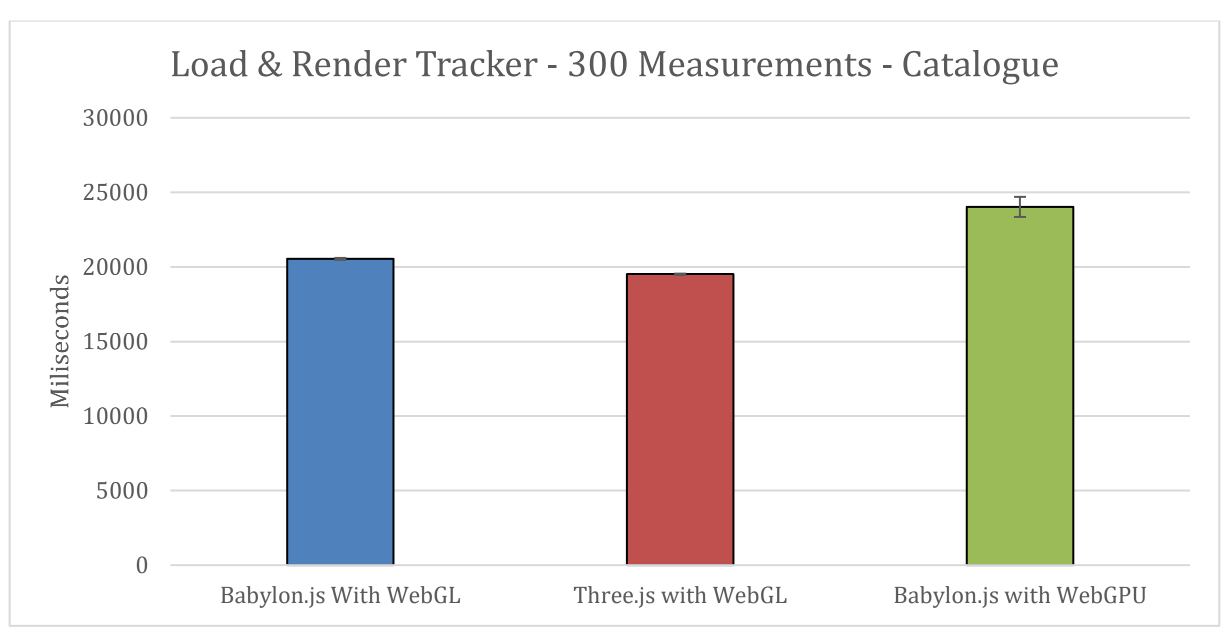


**Figure 65** – Bar Chart of Load & Render Values (Catalogue).

In the bar chart within **Figure 65**, based on the same measurements as **Figure 64**, we can see that the error bars representing standard deviation are barely visible in both the WebGL versions, but noticeable within the Babylon.js with WebGPU version. These findings present us with an Anova P-value < 0.001, making this a statistically significant difference showing that Three.js with WebGL is slightly faster than Babylon.js with WebGL, with both of these being slightly noticeably faster than Babylon.js with WebGPU. This proves that Three.js with WebGL is the fastest alternative of the three, while Babylon.js with WebGPU provides a much broader difference in measurements and therefor maintaining a much less consistent result.

| Babylon.js With WebGL | | Three.js with WebGL | | Babylon.js with WebGPU | |
|---|---|---|---|---|---|
| Mean | 20544,36 | Mean | 19510,38 | Mean | 24025,96 |
| Standard Error | 3,622128 | Standard Error | 2,580973 | Standard Error | 39,09571 |
| Median | 20550 | Median | 19514 | Median | 23979 |
| Mode | 20554 | Mode | 19497 | Mode | 24100 |
| Standard Deviation | 62,84158 | Standard Deviation | 44,77822 | Standard Deviation | 678,2851 |
| Sample Variance | 3949,064 | Sample Variance | 2005,089 | Sample Variance | 460070,7 |
| Range | 436 | Range | 259 | Range | 6192 |
| Minimum | 20305 | Minimum | 19347 | Minimum | 22929 |
| Maximum | 20741 | Maximum | 19606 | Maximum | 29121 |
| Sum | 6183852 | Sum | 5872623 | Sum | 7231814 |
| Count | 301 | Count | 301 | Count | 301 |

**Figure 66** – Statistical Data for Load & Render Values (Catalogue).

The statistical data presented within **Figure 66** presents the findings from **Figure 64** as well as **Figure 65**, here we can see that the standard deviation for Babylon.js with WebGPU is much higher than Babylon.js with WebGL as well as Three.js with WebGL, as opposed to what was presented within **Figure 62** and **Figure 58**. Here it's also very noticeable that the

difference in mean between Babylon.js with WebGL and Three.js with WebGL is barely noticeable, only being a difference of ~1000.

| Group 1 | Group2 | Meandiff | P-adj | Lower | Upper | Reject |
|---|---|---|---|---|---|---|
| Babylon.js WebGL | Babylon.js WebGPU | 3481.601 | 0.00 | 3387.760 | 3575.442 | True |
| Babylon.js WebGL | Three.js WebGL | -1033.983 | 0.00 | -1127.824 | -940.142 | True |
| Babylon.js WebGPU | Three.js WebGL | -4515.584 | 0.00 | -4609.425 | -4421.743 | True |

**Figure 67** – Tukey HSD Data for Load & Render Tracker (Catalogue).

A Tukey HSD test was performed here as well, showing similar results to the ones presented within **Figure 63** as well as **Figure 59** where the null hypothesis was rejected for all artifacts as there was a significant difference when compared to each other.

### 10.2.2 Individual Version

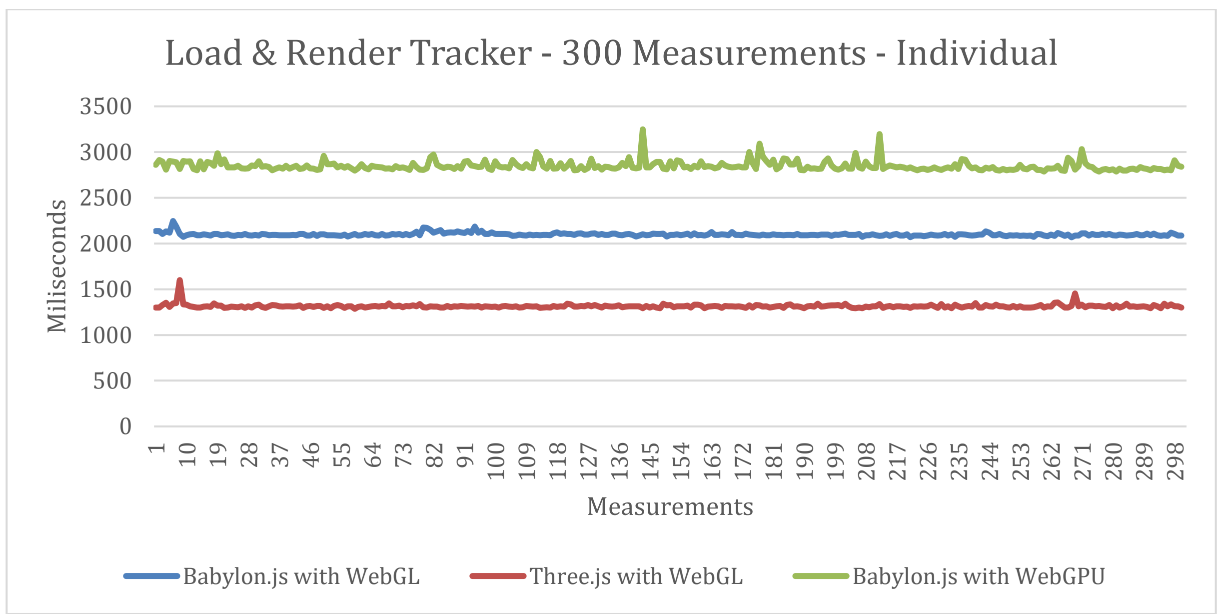


**Figure 68** – Line Chart for Load & Render Values (Individual).

The individual version presented within **Figure 68** shows a much more noticeable difference than the one for the catalogue version found within **Figure 64**. It's also noticeable that Babylon.js with WebGPU presents much more consistent measurements than the findings in **Figure 64**.

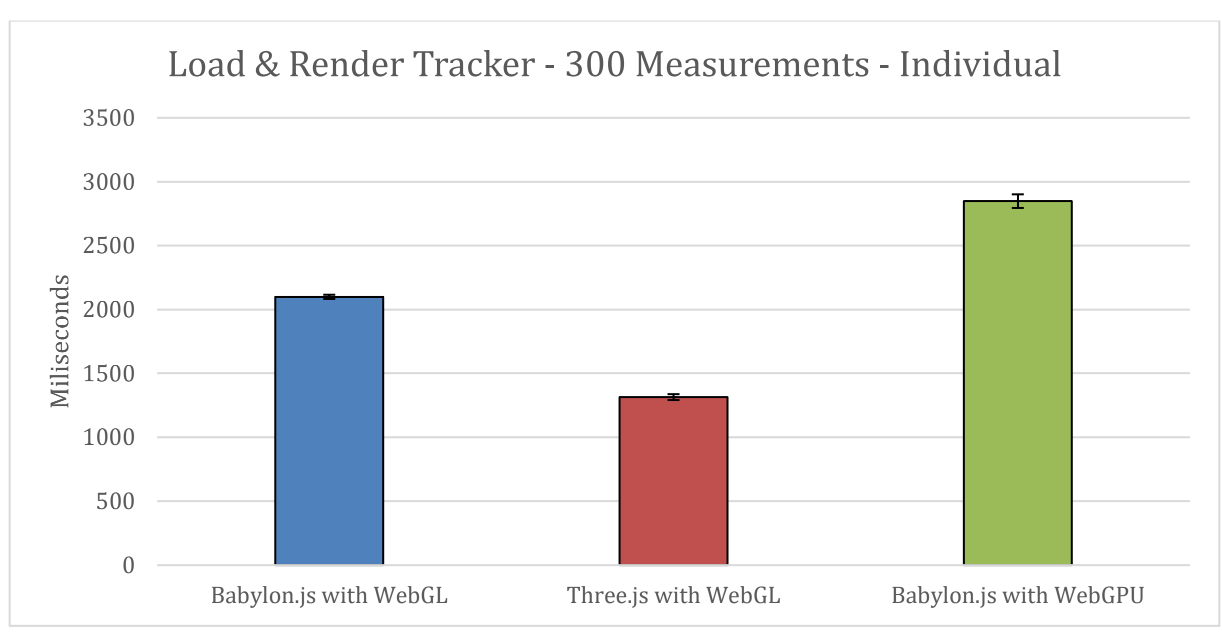


**Figure 69** – Line Chart for Load & Render Values (Individual).

The findings within **Figure 69** presents a much broader range than that found within **Figure 65** for the catalogue version, where even the error bars remain far from each other. Although proving that the standard deviation for Babylon.js with WebGPU remains lower than that found in the catalogue version. These findings present an Anova P-value of < 0.001, making these findings statistically significant.

| **Babylon.js with WebGL** | | **Three.js with WebGL** | | **Babylon.js with WebGPU** | |
|---|---|---|---|---|---|
| Mean | 2098,897 | Mean | 1314,287 | Mean | 2847,573 |
| Standard Error | 1,025058 | Standard Error | 1,278339 | Standard Error | 3,096795 |
| Median | 2094 | Median | 1312,5 | Median | 2833,5 |
| Mode | 2094 | Mode | 1313 | Mode | 2821 |
| Standard Deviation | 17,75453 | Standard Deviation | 22,14148 | Standard Deviation | 53,63806 |
| Sample Variance | 315,2234 | Sample Variance | 490,2453 | Sample Variance | 2877,041 |
| Range | 178 | Range | 314 | Range | 461 |
| Minimum | 2068 | Minimum | 1286 | Minimum | 2787 |
| Maximum | 2246 | Maximum | 1600 | Maximum | 3248 |
| Sum | 629669 | Sum | 394286 | Sum | 854272 |
| Count | 300 | Count | 300 | Count | 300 |

**Figure 70** – Statistical Data for Load & Render Values (Individual).

The statistical data presented within **Figure 70** strengthens the claim of there being a much bigger difference in mean value than the one presented within **Figure 66** for the catalogue version, here we can also see that the standard deviation for Babylon.js with WebGPU is noticeably higher than that of Babylon.js with WebGL as well as that of Three.js with WebGL.

| Group 1 | Group2 | Meandiff | P-adj | Lower | Upper | Reject |
|---|---|---|---|---|---|---|
| Babylon.js WebGL | Babylon.js WebGPU | 748.676 | 0.00 | 740.320 | 757.032 | True |
| Babylon.js WebGL | Three.js WebGL | -784.610 | 0.00 | -792.965 | -776.254 | True |
| Babylon.js WebGPU | Three.js WebGL | -1533.286 | 0.00 | -1541.642 | -1524.930 | True |

**Figure 71** – Tukey HSD Data for Load & Render Tracker (Individual).

Just like for the previous measurements, the Tukey HSD test was done and presented same findings as **Figure 59**, **Figure 59** and **Figure 67** where the null hypothesis was rejected and a statistically significant difference between each artifact was presented.

# 11 Discussion

An important discussion topic to bring up early is the issue regarding optimization in software development, these implemented artifacts are without optimization that might come from experience or a deeper knowledge of these APIs and libraries. Because of this, the results can't be presented as undoubtable proof of the superiority of one combination in relation of others, this can only be presented as proof of this in the context of implementation without optimization. In other words, this is the non-optimal version of each artifact. A great example of this is regarding the catalogue version of Babylon.js with both WebGL and WebGPU. As shown in **Figure 32** where one engine is initialized for each canvas window, it's technically possible as of Babylon.js version 4.1 to initialize one global engine and a view for each canvas window, possibly benefitting performance. Though I couldn't get this working for unique models. Although while this is an important aspect to consider for the catalogue measurements presented within **Figure 56** through **Figure 59** as well as from **Figure 64** to **Figure 67** this does not account for differences in results presented within the individual measurements found within **Figure 60** to **Figure 63** as well as from **Figure 68** to **Figure 71**. But it can absolutely have influenced the catalogue measurements that shouldn't be dismissed. Another example of these optimizations could have been the model quality, simply viewing the models presented within Three.js shows that they're of lower quality than the ones in Babylon.js, even though they both use the same baseline models. This might mean that the graphical advantages of Three.js is not baseline but to be implemented individually, or that Babylon.js has better baseline graphical advantages that could be turned off to create more accurate measurements, though this would deviate from the goal of retaining as much similarity between the artifacts as possible, even if to a detriment to the graphical accuracy between the artifacts. Another important example is the effect of compatibility mode for WebGPU in Babylon.js, this is defined in the Babylon.js documentation as making all scenes in WebGPU also be capable of being rendered with WebGL without further modification, but this isn't necessarily a simple implementation as it can introduce different issues and might not even introduce performance improvements. Another important topic to mention is the issue regarding measuring frames per second, while I originally considered introducing another computer to measure on after the findings from the Pilot Study, the issue still appeared of cutting off values above the refresh rate of the screen. Unlocking the FPS introduced a minor issue as the "animation()" function presented within **Figure 29** and **Figure 36** where the animation in the form of the models spinning would trigger much more rapidly than it should, leading to the interpretation that this is tied to the frames per second. Given the unlocking of this and the models rapidly moving, this might have had a slight effect on the actual frames per second and therefore the measurements presented. Another point worth discussing is the usage of textured models, since the models' textures were in 1k this could have influenced the results, where for example Babylon.js with WebGPU could have performed much better with higher quality models. There was an implementation of "modelinformationHQ" as mentioned earlier with higher quality models, though due to lack of time I couldn't test using these models as well.

## 11.1 Ethics

There is no user information or user involvement within the thesis which avoids a lot of the ethical discussion, but the measurements itself must be mentioned. As only 300 measurements were taken for the load and render time measurements these aren't necessarily conclusive but serve as a solid basis to draw conclusions from. The lack of more measurements is based in the

time necessary for taking these measurements, as only the 300 measurements took multiple hours. As previously mentioned, all the code is on Github and can be followed through the footnotes within the implementation section for future development, re-development, external analysis and further understanding of the study. Although there is one major consideration to mention which is the inherent goal of this thesis was to be able to further develop e-commerce websites to replace physical stores with the goal of costing society less resources. This goal comes with a potential issue of having much more sources of personal information for individuals with ill intent to attempt to compromise and puts a much bigger strain on IT-security in general.

## 11.2 Sustainability

An important topic to consider is the sustainability conclusions to draw from this thesis. In practice, while one option might perform better technically the implementation time is of bigger concern as the sustainability efforts should aim to keep implementation time low and the generalized implementation simple for easier development and less time and therefore energy and resources to be used during this phase. The results point towards Three.js with WebGL being the better option for this type of implementation and from my own personal interpretation this option was also the easiest to develop and implement as Babylon.js with both WebGL and WebGPU suffered more issues during the development time and had both worse documentation and more confusing implementation to use in development. With this in mind, I'd argue that Three.js with WebGL is the more sustainable option at least for those lacking experience and the deep understanding that might prove necessary for Babylon.js and especially WebGPU, where many optimizations can be implemented.

Another important topic to discuss is how the results can be applied for sustainable development within society, while the ability to successfully supplement an e-commerce environment with the ability to render 3D models representing the respective articles there is a broader application where this knowledge can serve as a healthy tool for society. One field of application where this currently takes place and can be further developed is the field of medicine, where models representing organs as well as bone and muscle structures are used for doctors, nurses, students, and such to learn and take advantage of. While the result from this thesis isn't necessarily enough to use for the implementation of such a critical and important field, this can still serve as a way of helping society. An example of this is for the elderly in Sweden where the carers at retirement homes will handle the purchasing of foods and similarly, the elderly can be assisted by the visual element of a 3D model for deicing what they wish to purchase. Reusing knowledge like this is known as digitalization. Another example where this could be for example applicable can be in the CAD environment, where computer-aided design is used to create intricate models and designs for manufacturing, while the web-based options for 3D models probably isn't effective enough to handle such a task this can absolutely be applied for the visualization of such products and artifacts for sale or for displaying on the web.

## 11.3 Conclusion

Based on the findings presented within all the aforementioned figures, the hypothesis for this study can be rejected. As Babylon.js with WebGPU was shown to perform worse in both measured variables in both the catalogue and individual versions. This shows that Babylon.js with WebGPU does not perform better than Three.js with WebGL or Babylon.js with WebGL.

Based on these findings, the null hypothesis can also be rejected as there was a significant change. Through all of these findings, Three.js with WebGL performed better in all variables, while Babylon.js with WebGL performed better than Babylon.js with WebGPU. It's worth mentioning that the scope of the study was to conclude if there is a difference between WebGPU and WebGL and as a Tukey test was done to analyze differences between each measured artifact where a significant difference was shown in each test done, the conclusion that WebGPU performed worse than WebGL in all measurements can be concluded.

There are multiple reasons as to why this is the case, WebGPU might simply not be optimized enough yet, or it can simply present a focus on model quality before technical performance. This stands in contrast to studies done by Dyken et al. (2022), Hidaka et al. (2017) as well as Poudel, Usher & Petruzza (2023) that all showed performance improvements by using WebGPU. Their findings can be because of intensive optimization work done that wasn't applicable in this thesis, but these are also different applications and face difference issues and restrictions and can't be used to argue against their findings necessarily.

The purpose of this study was inherently to find a better approach to represent models on the web with the foundational idea that superiorly implemented methods for this will lower the necessity for physical stores in favor of digital stores. This to aim for a future where a lower environmental footprint can be achievable and a smaller number of detrimental resources have to be used to uphold these physical stores, this idea is based on the overwhelming positivity in regard to 3D models in e-commerce as discussed under the chapter "3.1.3 WebGL in E-commerce". This idealistic goal does present the ethical issue of increasing the sources of digital risk in the form of losing private information and privacy.

One important ethical aspect to mention for future studies involving a similar approach to the measuring of frames per second is that this proved to be very intense on the computer, while the implementation serves as a way to avoid having to introduce another computer for measurements, I still opted to attempt the approach of disable the frame rate limit on the laptop computer which almost instantly caused an operating systems crash every time I tried.

## 11.4 Future Work

To align with previously mentioned points within the discussion, future work should aim to measure the performance difference between WebGL and WebGPU in e-commerce with a bigger perspective and capabilities, taking multiple measurements at intervals to analyze the difference of long-term usage of the website as well as testing against multiple computers of different specifications to see a broader range of differences. Another possibility of future work should focus on the performance of WebGPU in different applications in comparison to WebGL, as e-commerce is a small application field and not necessarily the most applicable one to present the functionality, performance, and improvements of WebGPU over WebGL. It will also be of interest to analyze the potential of WebGPU in new environments as the opportunity arises, for example in other browsers and operating systems. An example could be web-browser based video games, where the benefits of using WebGPU could have the opportunity to present its true changes from WebGL.

# 12 References

Algharabat, S, R. & Dennis, C. (2010). Using authentic 3D product visualization for an electrical online retailer. *Journal of Customer Behaviour*: 97-115. DOI: 10.1362/147539210X511326

Angel, E. (2017). The Case for Teaching Computer Graphics with WebGL: A 25-Year Perspective. *IEEE Computer Graphics and Applications 37(2)*: 106-112. DOI: 10.1109/MCG.2017.26

Babylon.js. (n.d. a). UniversalCamera. https://doc.babylonjs.com/typedoc/classes/BABYLON.UniversalCamera [2024-04-16]

Babylon.js. (n.d. b). WebGPU. https://doc.babylonjs.com/setup/support/webGPU [2024-04-17]

Babylon.js. (n.d. c). Quaternion. https://doc.babylonjs.com/typedoc/classes/BABYLON.Quaternion [2024-04-19]

Barmar. (2013). Local Storage: MySQL vs JSON?. *StackOverflow*. https://stackoverflow.com/questions/16901920/local-storage-mysql-vs-json [2024-04-13]

Bauer, V., Heinemann, L. & Deissenboeck, F. (2012) A structured approach to assess third-party library usage. *28th IEEE International Conference on Software Maintenance*: 489-482. DOI: 10.1109/ICSM.2012.6405311

Borissova, D., Dimitrova, Z., Dimitrov, V., Yoshinov, R., Garvanova, M. & Garvanov, I. (2021) Multi-Attribute Decision-Making Model for Ranking of Web Development Frameworks. *25th International Conference on Circuits, Systems, Communications and Computers*: 3-8. DOI: 10.1109/CSCC53858.2021.00009

Collins, Camille. (2021). Is e-commerce really sustainable? Understanding its impact on the environment. *Sana*. https://www.sana-commerce.com/blog/impact-of-ecommerce-on-the-environment/ [2024-05-20]

Dethe, S, H. & Joy, E. (2023). Revolutionizing E-commerce with 3D Visualization: An Experimental Assessment of Behavioural Shopper Responses to Augmented Reality in Online Shopping. *International Conference for Emerging Technology*: 1-6. DOI: 10.1109/INCET57972.2023.10170472

DougM. (2013). Local Storage: MySQL vs JSON?. *StackOverflow*. https://stackoverflow.com/questions/16901920/local-storage-mysql-vs-json [2024-04-13]

Dyken, L., Poudel, P., Usher, W., Petruzza, S., Chen, Y. J. & Kumar, S. (2022). GraphWaGu: GPU Powered Large Scale Graph Layout Computation and Rendering for the Web. *22nd Eurographics Symposium on Parallel Graphics and Visualization*: 73-83. DOI: 10.2312/PGV.20221067

Eterhyte. (2014). How do I make functions added by Tampermonkey be available in console after script has been ran?. *StackOverflow*. https://stackoverflow.com/questions/27680230/how-do-i-make-functions-added-by-tampermonkey-be-available-in-console-after-the [2024-04-11]

Feng, L., Wang, C., Li, C. & Li, Z. (2011). A Research for 3D WebGIS based on WebGL. *2011 International Conference on Computer Science and Network Technology*: 348-351. DOI: 10.1109/ICCSNT.2011.6181973

Ferrarezi, J., Neto, M., Dias, D., Pilastri, A., Guimarães, M. & Brega, J. (2016). LibViews – An Information Visualization Application for Third-Party Libraries on Software Projects. *20th International Conference Information Visualisation*: 136-140. DOI: 10.1109/IV.2016.43

Ferraz, O., Menezes, P., Silva, V. & Falcao, G. (2021). Benchmarking Vulkan vs OpenGL Rendering on Low-Power Edge GPUs. *International Conference on Graphics and Interaction*: 1-8. DOI: 10.1109/ICGI54032.2021.9655285

Geelhaar, J. & Rausch, G. (2015) 3D Web Applications in E-Commerce – A secondary study on the impact of 3D product presentations created with HTML5 and WebGL. *International Conference on Computer and Information Science 14*: 379-382. DOI: 10.1109/ICIS.2015.7166623

Greasespot. (2018a). GM.setValue. https://wiki.greasespot.net/GM.setValue [2024-04-20]

Greasespot. (2018b). GM.getValue. https://wiki.greasespot.net/GM.getValue [2024-04-20]

Hewawalpita, S. & Perera, I. (2017) Effect of 3D product presentation on consumer preference in e-commerce. *Moratuwa Engineering Research Conference 1*: 485-490. DOI: 10.1109/MERCon.2017.7980532

Hidaka, M., Kikura, Y., Ushiku, Y. & Harada, T. (2017) WebDNN: Fastest DNN Execution Framework on Web Browser. *ACM International Conference on Multimedia*: 1213-1216. DOI: 10.1145/3123266.3129394

Hirose, K. & Koga, M. (2023). Development of a Web Application to Support Validation of Indoor Robots Using AR. *23rd International Conference on Control, Automation and Systems*: 694-699. DOI: 10.23919/ICCAS59377.2023.10316875

Hoetzlein, C, R. (2012). Graphics Performance in Rich Internet Applications. *IEEE Computer Graphics and Applications 32*: 98-104. DOI: 10.1109/MCG.2012.102

Imms, D. (2017). Fast and Simple JavaScript FPS Counter. *Growing with the Web*. https://www.growingwiththeweb.com/2017/12/fast-simple-js-fps-counter.html [2024-04-10]

Li, H., Daugherty, T. & Biocca, F. (2002). Impact of 3-D Advertising on Product Knowledge, Brand Attitude, and Purchase Intention: *The Mediating Role of Presence. Journal of Advertising*: 43-58. DOI: 10.1080/00913367.2002.10673675

Li, L., Qiao, X., Lu, Q., Ren, P. & Lin, R. (2020). Rendering Optimizations for Mobile Web 3D Based on Animation Data Separation and On-Demand Loading. *IEEE Access 8*: 88474-88486. DOI: 10.1109/ACCESS.2020.2993613

Li, Sing. (2019). Introducing WebGL. *IBM Developer*. https://developer.ibm.com/tutorials/wa-webgl1/ [2024-05-19] (a)

Lujan, M., Baum, M., Chen, D. & Zong, Z. (2019) Evaluating the Performance and Energy Saving Efficiency of OpenGL and Vulkan on a Graphics Rendering Server. *International Conference on Computing, Networking and Communications*: 777-781. DOI: 10.1109/ICCNC.2019.8685588

Manor, Eytan. (2021). The story of WebGPU – The successor to WebGL. *Medium*. https://eytanmanor.medium.com/the-story-of-webgpu-the-successor-to-webgl-bf5f74bc036a [2024-05-19]

Mehrara, M., Hsu, P-C., Samadi, M. & Mahlke, S. (2011) Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. *IEEE 27th International Symposium on High Performance Computer Architecture*: 87-98. DOI: 10.1109/HPCA.2011.5749719

Mete. (2019). How do I disable Vsync for Chrome?. *Google Chrome Help*. [2024-05-16] https://support.google.com/chrome/thread/22363727/how-do-i-disable-vsync-for-chrome?hl=en

Miao, R., Song, J. & Zhu, Y. 2017. 3D geographic scenes visualization based on WebGL. *6th International Conference on Agro-Geoinformatics*: 1-6. DOI: 10.1109/Agro-Geoinformatics.2017.8046999

Moloo, K, R., Pudaruth, S., Ramodhin, M. & Rozbully, B, R. (2016). A 3D Virtual Tour of the University of Mauritius using WebGL. *International Conference on Electrical, Electronic and Optimization Techniques*: 2891-2894. DOI: 10.1109/ICEEOT.2016.7755226

Movania, M, M., Chiew, M, W. & Lin, F. (2013). On-Site Volume Rendering with GPU-Enabled Devices. *Wireless Personal Communications*: 795-812. DOI 10.1007/s11277-013-1354-y

Palepu, K, V., Xu, G. & Jones, A. J. (2013). Improving efficiency of dynamic analysis with dynamic dependence summaries. *28th IEEE/ACM International Conference on Automated Software Engineering (ASE):* 59-69. DOI: 10.1109/ASE.2013.6693066

Panchal, S., Raval, P., Shetty, S. & Ambadekar, S. (2022). College 3D Model Rendering Using Three JS. *International Conference on Advances in Science and Technology 5*: 142-147. DOI: 10.1109/ICAST55766.2022.10039553

Poudel, P., Usher, W. & Petruzza, S. (2023). Multi-layer Caching and Parallel Streaming for Large Scale Cloud Optimized Point Cloud Visualization Using WebGPU. *IEEE International Conference on Big Data*: 360-365. DOI: 10.1109/BigData59044.2023.10386238

Resch, B., Wohlfahrt, R & Wosniok, C. (2014). Web-based 4D visualization of marine geo-data using WebGL. *Cartography and Geographic Information Science 41(3):* 235-247. DOI: 10.1080/15230406.2014.901901

Reydar. (n.d.). 3D Model Product Viewers: How Can Ecommerce Sites Benefit?. https://www.reydar.com/3d-model-products-for-ecommerce-sites-benefits/ [2024-05-20]

Shah, H., Tupe, V., Rathod, A., Shaikh, S. & Uke, N. (2021). A Progressive Web App for Virtual Campus Tour. *International Conference on Computing, Communication and Green Engineering 1*: 1-5. DOI: 10.1109/CCGE50943.2021.9776419

Sheng, B., Zhao, F., Zhang, C., Yin, X. & Shu, Y. (2017). 3D Rubik's Cube – online 3D modeling system based on WebGL. *IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference*: 575-579. DOI: 10.1109/ITNEC.2017.8284798

Spider Digital Group. (2024). The Rise of WebGL. https://www.spiderdigitalgroup.com/post/the-rise-of-webgl [2024-05-20]

Three.js. (2024). PerspectiveCamera. https://threejs.org/docs/?q=ca#api/en/cameras/PerspectiveCamera [2024-04-16]

Usher, W. & Pascucci, V. (2020). Interactive Visualization of Terascale Data in the Browser: Fact or Fiction? *IEEE 10th Symposium on Large Data Analysis and Visualization*: 27-36. DOI: 10.1109/LDAV51489.2020.00010

Wikipedia. (2024). WebGL. https://en.wikipedia.org/wiki/WebGL. [2024-05-19]

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B. & Wesslén, A. (2012). Experimentation in Software Engineering. Springer Heidelberg.

Wöllmann, S., Zink, R & Piser, M. (2020). 3D Mapping to Collect Volunteered Geographic Information. *International Conference on Advanced Computer Information Technologies 10*: 509-513. DOI: 10.1109/ACIT49673.2020.9208958

WWF. (n.d.) Why it's important that we value nature. https://www.wwf.org.uk/what-we-do/valuing-nature [2024-05-20]

Yan, Y., Tu, T., Zhao, L., Zhou, Y. & Wang, W. (2021). Understanding the performance of webassembly applications. *21st ACM Internet Measurement Conference*: 533-549. DOI: 10.1145/3487552.3487827

Yu, M. & Liu, F. (2016). Real-time rendering for realistic human skin on web-site. *International Conference on Audio, Language and Image Processing*: 51-56. DOI: 10.1109/ICALIP.2016.7846637

Zhang, T., Wang, X., Teng, Y., Fang, J., Zhang, Y. & Chen, X. (2023). Fast Realization of Robot 3D Simulation Based on WebGL. *35th Chinese Control and Decision Conference*. DOI: 10.1109/CCDC58219.2023.10326909