Master's thesis

Master's Programme in Computer Science

Optimizing WebGL application performance by identifying and tackling bottlenecks

Kim Toivonen

December 19, 2022

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Kim Toivonen

Työn nimi — Arbetets titel — Title

Optimizing WebGL application performance by identifying and tackling bottlenecks

Ohjaajat — Handledare — Supervisors

Dr. Gopika Premsankar and Dr. Ashwin Rao

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	December 19, 2022	0 pages

Tiivistelmä — Referat — Abstract

Browser based 3D applications have become more popular since the introduction of the Web Graphics Library (WebGL). However, they have some unique characteristics, such as the inability to access the local file system and the requirement to be executed in the browser's scripting environment. These characteristics can introduce performance bottlenecks, and WebGL applications are also vulnerable to the same bottlenecks as traditional 3D applications.

In this thesis, we aim to provide guidelines for designing WebGL applications by conducting a background survey and creating a benchmarking platform. Our experiments showed that loading model data from the browser's execution environment to the GPU has the biggest impact on performance. Therefore, we recommend focusing on minimizing the amount of data that needs to be added to the scene when designing 3D WebGL applications. Additionally, we found that the amount of data rendered affects the severity of performance drops when loading model data to the GPU, and suggest actively managing the scene by only including relevant data in the rendering pipeline

ACM Computing Classification System (CCS)
General and reference → Document types → Surveys and overviews
Applied computing → Document management and text processing → Document management → Text editing

Avainsanat — Nyckelord — Keywords

WebGL, glTF, JSON, Three.js, 3D rendering performance

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Networking study track

Contents

1 Introduction

Web browsers offer a convenient environment for software developers to create cross-platform applications. Technologies that web browsers support are well standardized and widely supported by different browsers. This has resulted in more and more applications being implemented as completely browser based. As the popularity of browser as an application environment has risen, so has the amount of functionalities that browsers are capable of performing. One example of this is WebGL (Web Graphics Library) [42]. It is an API that allows a web browser to render 2D and 3D graphics to HTML5's canvas-element, using GPU. It is supported by all modern browsers, and it has several libraries built on top of it.

Creating a 3D rendering application in the browser does create some additional challenges. Instead of having just a single codebase for a single application, one has to consider both the client application running in the browser as well as the server side application that runs on the server. These two applications need to communicate with each other constantly. This situation is fundamentally different when comparing it to a desktop application where the application reads and writes data to a local storage device. Transmitting data over the network is much slower than reading the same data from the local device. Slow network might make the application unresponsive from the users perspective if the data to be rendered takes a long time to be transmitted from the server storage to the application. 3D data is usually large in nature, meaning that the amount of data that the browser needs to load over the network is considerably larger than in standard web applications. In this regard, they are similar to multimedia web applications like video streaming sites.

Unlike multimedia web applications, a 3D rendering web application has also an additional challenge of having a cost for displaying the downloaded data after it has been received by the client. A WebGL application sends several draw calls per second to the client machines GPU which then draws the result to the browser window. This requires considerably more computing resources than simply displaying stylized text on a web page. The rendering process is executed several times each second meaning that a large amount of data is a possible bottleneck in the applications performance during the loading and also for the whole time it is visible to the user. Because of this additional cost, a WebGL application needs to actively manage the content that is loaded to keep the performance of the application at an acceptable level. As mentioned, downloading data through the network is also an expensive operation so there are several reasons why all redundant loading of 3D data should be kept to a minimum.

A browser application is usually run so that a piece of JavaScript code is fetched from the server when the user enters a web page. JavaScript is tokenized and interpreted by the browsers on runtime and it is not possible to compile the code beforehand [36]. Browser's JavaScript engines also force the JavaScript to be single threaded. These characteristics of JavaScript when compared to 3D desktop applications which can be run on native code introduce yet more performance penalties. JavaScript does allow multithreading with the help of web workers but in a more limited manner than traditional threading [41]. For example the only way to pass information between threads can be done by serializing and deserializing data to either strings or numbers back and forth making it ill-suited for handling processing of large number of 3D data.

When an application runs on a web browser it can not make many assumptions about the hardware it is running on as opposed for video games and other desktop applications where the systems compatibility can be verified during the installation phase. Virtually every computer including mobile devices run a modern web browser which is the only requirement for running WebGL applications.

In addition of the specific challenges just mentioned, a WebGL based 3D application shares all the common challenges with traditional 3D applications. These are for example how to remove unnecessary complexity and detail from the scene based on the users interactions, how to make the graphics as good as possible while keeping the application performance on a high enough level etc. This might need to be addressed by for example creating a system that depending on the user's actions we might want to only show a coarse representation of a model instead of rendering it entirely. Or we might want to sacrifice precision for performance in cases where the hardware in use can not handle rendering a more detailed model. This is usually achieved by having multiple levels of detail (LOD) for each model.

To summarize there are at least the following possible performance bottlenecks for WebGL applications:

- The computing capacity of the clients device on which the WebGL applications are running varies.
- Transferring a large amount of content over the network.
- Distributed nature of the client-server architecture.

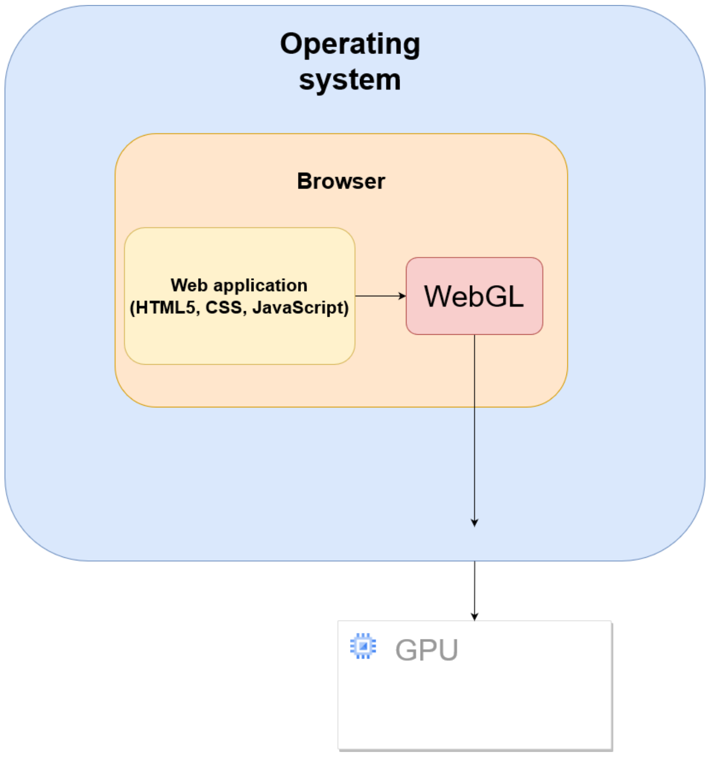


Figure 1.1: A graph describing the relation of WebGL, the browser and the operating system. Source: https://developer.ibm.com/tutorials/wa-webgl1/

- Performing computationally heavier tasks than a standard web application.
- Having to be implemented by an interpreted language at the client side.
- All the inherited performance issues from traditional 3D rendering applications.

A WebGL application design needs to account for both client and server sides. Even if the models have been stored optimally and the server provides a flexible interface for extracting only the relevant data, it would got to waste if the client side implementation can not take advantage of these various methods of optimization.

A server side implementation needs to offer some type of interface for the client side application to communicate with. They need to be able to exchange information about what is relevant data to the client by for example determining the subset of the scene the client sees *i.e.* the clients viewport. Using this information the server side application needs to be able to perform some type of spatial querying to the dataset. As 3D model data is generally stored in disk storage, reading each file to determine its location in the world and comparing it to the viewport is not a viable option. Instead some information about the model should be stored in a high availability storage *e.g.* a database. This

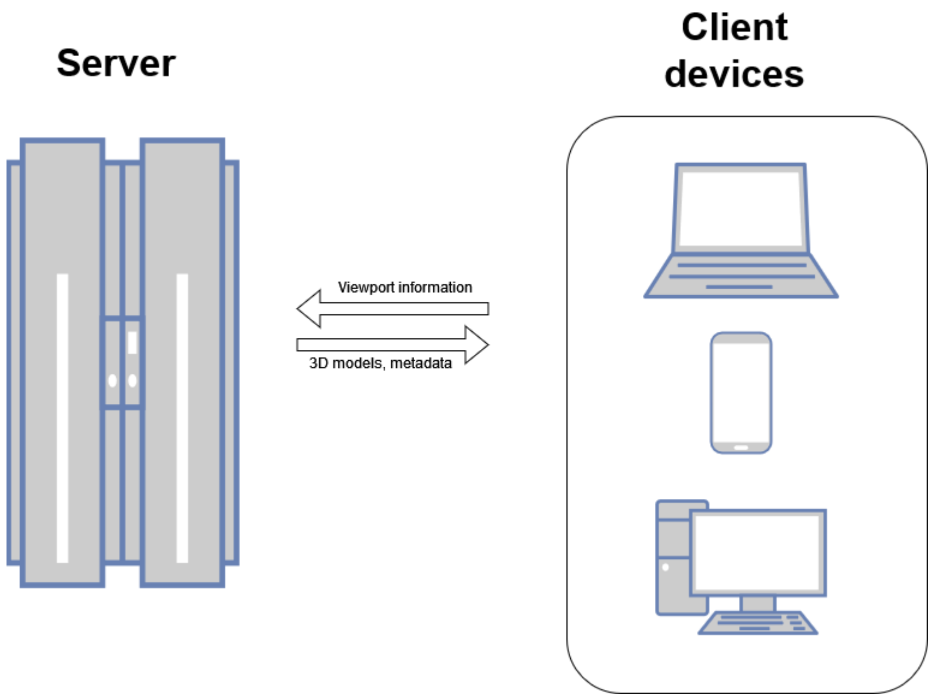


Figure 1.2: An example of a client-server architecture in WebGL context

information could be for example some type of bounding volume. Bounding volumes can be represented for example as a bounding sphere or a bounding box. Both are small and simple structures with a bounding sphere consisting only of a center point in the form of a three-dimensional vector and a radius which is a single numeric value. A bounding box can be represented as a pair of three-dimensional vectors describing the coordinates of two corners of the box. Having a simple enough representation allows the application to have location and size data of each model even in larger scenes so that it can be queried and filtered.

The server side application needs to store the models in a way that makes it convenient for the client side application to download them. The storage implementation might need to support *e.g.* level-of-detailing and splitting the dataset into small subsets to make extracting only relevant easier.

Client side application needs to be able to determine the area it wants to see models from. It needs to be able to download the models efficiently to the scene in the format that is used to store the models. Most of the traditional 3D rendering performance handling, such as managing the amount of data rendered, is also a responsibility of the client side application.

The goal of this thesis is to create a set of guidelines for creating WebGL applications.

This is done by building a benchmarking platform that can be used to assess the impact of the possible performance bottlenecks mentioned, and by using that platform create a some reference guidelines on what to focus on when designing a WebGL application. The thesis is organized as follows. In chapter 2, we go through related work related to the problem. Chapter 3 is an overview of the benchmarking system created. Chapter 4 explains the evaluation methodology used to conduct the benchmarking and visualizes and presents the results. In chapter 5 we discuss the results and contemplate on possible future improvements in regards of this thesis and possible limitations of the benchamrking. In chapter 6 we summarize our conclusion and lay out our reference guidelines for creating WebGL applications.

2 Background

There is some research regarding WebGL specifically as the platform and a lot of research on some relevant subjects such as level-of-detail and spatial querying which have been subjects of research in the realms of video games and general 3D rendering for several decades.

2.1 Client-server communication in WebGL applications

Limper *et al.* [25] performed a case study on fast delivery of 3D web content. They showed that for devices with subpar processing capabilities, such as mobile devices, the decoding time could quickly grow larger than the actual download time. They recommended avoiding additional compression and utilizing the browser's and HTTP's built-in compression capabilities.

Li *et al.* researched an on-demand loading mechanism for animated 3D models in mobile web 3D applications [22]. They separated the animation data from the model and applied their on-demand tool to both separately. In addition, they proposed using an asynchronous request-response mechanism. They noted that since modern browsers' JavaScript engines are single-threaded, loading a large amount of model data synchronously blocks the execution of the program which in turn affects the user experience negatively. They managed to improve the user experience by reducing the amount of data that was loaded initially. Even though their research focused on dynamic 3D models and animations, the same on-demand asynchronous loading should apply to static 3D models.

2.2 Storage formats

Several file formats exist for storing 3D models. These include for example X3D [9], STL [34], FBX [10], glTF [12], and Wavefront OBJ [39]. These different formats were designed with different use cases in mind and thus have different capabilities and performance attributes.

Lee *et al.* [21] conducted a study on the performance of OBJ, STL, FBX, and glTF file formats. According to the comparison, glTF was the most efficient format in terms of downloading and decoding time for fetching and rendering a 3D model. Therefore, we decided to use glTF as the storage format for 3D models in this thesis.

glTF was developed by the Khronos group as a general-purpose 3D data storage format that is compatible with WebGL. It stores the mesh data in binary format so that it can be directly loaded into the GPU's buffers on the client side without any decoding. Additionally, glTF allows storing the scene structure and model metadata in JSON format within the same file as the mesh data. Currently, it is supported by most WebGL libraries. One limitation of glTF is that it does not have built-in support for progressive loading.

Cesium [6] has published a format known as 3D tiles, that has been built on top of glTF. It supports progressive loading of the meshes with different level-of-detail representations. A model based on 3D tiles is represented as a JSON file and a list of model files. The JSON file stores metadata that the application can use to determine what models should be loaded. Even though 3D tiles is an open specification, and other WebGL libraries can theoretically use it, it does not have wide support outside CesiumJS. It is tightly developed together with Cesiumjs which is shown by Kraemer et al. [19] to be well-suited only for larger-scale geospatial applications. At the time of the writing of this thesis several projects are adding 3D tiles support for Three.js such as NASA's and New York Times' Three.js based 3D tiles library [27] [37]. They are both still under development and neither of them has the support for everything defined in the 3D tiles specification.

Schilling *et al.* [29] investigated the optimal streaming of 3D city model data. They utilized the B3DM (Batched 3D Model) format to bundle several models into a single file which reduced the amount of HTTP calls the client has to make to the server. This format is also supported by Cesium 3D Tiles [6].

2.3 JavaScript performance

Theisen *et al.* performed a benchmark study of JavaScript in 2019 [36]. They compared its effectiveness in scientific computing to Java, and concluded that JavaScript can never be faster than the native code that runs it. One interesting observation was that JavaScript performance varies greatly depending on the browser it runs on. For example, in some tasks, JavaScript on Chrome was 8-9 times faster than JavaScript on Safari. Nonetheless, Java performed better than JavaScript on any browser in all of the tested tasks.

Web workers were introduced as a means to enable multithreading in browser applications [41]. They can be used to offload computation from the main thread, but they have some notable weaknesses. For instance, workers do not have access to the document object model (DOM), which means that any work performed by a worker can only be made visible to the user by transferring the result to the main thread. This also implies that the only way to share data between the main thread and workers is to serialize it to JSON in the source thread and deserialize it in the destination thread. This can be a challenge for 3D rendering applications, which handle large amounts of data several times per second and may not benefit from using workers.

Web Assembly is a technology that tries to bring native speed code execution to browsers [40]. It runs on a virtual stack that can be embedded into a browser. At the time of writing this thesis Web Assembly does not support direct access to the DOM and needs to communicate with the main JavaScript thread in a similar manner with workers.

2.4 WebGL performance

One can develop a browser-based 3D application by utilizing WebGL API directly which is widely supported in virtually every modern browser. Kramer *et al.* [19] performed a case study on three different open-source WebGL frameworks, CesiumJs, three.js, and X3DOM. They concluded that each had its use case. Cesium is meant to be used with strictly geospatial application, Three.js offers a direct access to WebGL and is very general purpose and X3DOM allows the development of 3D applications in a declarative high-level manner. In this thesis, we are going to focus on Three.js.

Three.js has a built-in loader for glTF. In addition Three.js' built-in glTF-loader has support for asynchronous loading which was recommended by Li *et al.*, to keep the program in a usable state during the time it is loading a large model [22].

Alatalo *et al.* [2] suggested using web workers to offload computation away from the main program. Even though the JavaScript engine in the browser is by default a single-threaded process, multi-threaded computation can be achieved with web workers. This helps the system to avoid a situation where the renderer is blocked by the processing of the 3D models before they are added to the scene. The authors noticed that by implementing loading and parsing of the models in a web worker, performance could be improved significantly with high-end devices. They mentioned that they were not using an optimized file format such as glTF in their system.

2.5 Level of Detail and progressive loading

Progressive Mesh was introduced by Hoppe *et al.* [14] back in 1996. It focuses on the effective compression of mesh data in a way that allows progressive decompression. When Limper *et al.* carried out a case study in 2013, they noticed that the original algorithm created by Hoppe could decompress more triangles per second than any other algorithm in their testing set [25]. However, Progressive Meshes are less-suited for web-based applications. This is because it only supports progressive rendering of the mesh, but not progressive transmission of the mesh.

Limper *et al.* [24] introduced a data structure of their own called POP buffer. POP buffers can create several level-of-detail (LOD) representations for a general triangle mesh to enable progressive loading. One of its main advantages is the ability to reuse lower LOD representations when constructing a higher LOD representation. With POP buffers, several LOD representations can be saved with zero memory overhead from having multiple versions of the same 3D model. Since POP buffers are a generic representation of any triangle mesh, they can also be represented as a set of glTF-models [25]. However, because the glTF-specification lacks support for progressive loading, all the different levels of detail representations need a separate glTF file. This creates memory overhead since one models needs several representations instead of one..

Scully *et al.* [31] tackled glTF's problem of not supporting progressive loading and implemented an extension of glTF that allowed a 3D scene to be partitioned so that it could be streamed progressively in the modified glTF format. Their work is part of a software ecosystem known as 3D Repo [31]. 3D Repo is sold as a propriety product. Additionally, since it uses a modified version of the glTF standard, the whole server and client side of the application needs to be able to utilize it. This means that the server and client implementations cannot be agnostic about how the other is implemented, as they could if a well-specified and supported standard was used.

If we allow some overhead that comes from representing each level of detail as a separate glTF-file we could use several different mesh simplification strategies. Going into detail about mesh simplification is not in the scope of this thesis but some well-known triangle mesh simplification strategies are mesh decimation [30] and quadratic error metrics-based simplification [11]. Some mathematical polygon approximations such as convex hulls and concave hulls can also be used as a lower level of detail model as shown by Selçuk *et al.* [32].

This thesis does not focus on which level-of-detailing method is the best for which use case. The idea behind level-of-detail and progressive loading is to have a way to download and process some data fast so that the application seems responsive from the users point of view as well as visualize as large areas as possible with the application staying usable.

2.6 Spatial querying

Filtering out irrelevant data is important when designing 3D applications. In WebGL applications it could be done both to the data already loaded to client side to exclude redundant data from the rendering pipeline or to determine from the scene graph what data should be loaded from the server to the client. The former can be done with operations based on the relation of the user's interactions and the actual geometry while the latter requires some secondary representation from the whole scene since it needs to evaluate which models are relevant before the actual geometry has been loaded to the client.

A technique known as view frustum culling has been implemented and improved on various occasions, for example by Zhang *et al.* [44] and Assarsson *et al.* [3]. In view frustum culling, the user's view is represented as a 3D object consisting of six planes and the 3D models that need to be rendered can be filtered out if they intersect with the frustum. View frustum culling however usually assumes that the geometry is already accessible when performing the computation. This is not the case in a web application where the geometry needs to be downloaded first for it to become available for further processing. Downloading every model that can be rendered at once is not a feasible alternative. It can however, be utilized for geometry that has been loaded to the scene

Cesium's 3D tiles have built-in functionality for loading only the relevant data to the scene [6]. Similar work has been also carried out by Chaturvedi *et al.* [7]. 3D Tiles' metadata contains bounding volumes of each tile, which can be then used to extract the relevant data from the tileset. Both, the work of Chaturvedi *et al.* and Schilling *et al.* [29], which has been implemented in the Cesium library, are designed to work with the CityGML standard, a standard specifically designed to describe virtual city models [18].

Alatalo *et al.* [2] describe a system for virtual city models that uses dynamic loading and unloading of assets based on the user's viewport. The models in the system are represented as blocks with textures applied to them, and the system has rules in place stating that all models should be blocks and that holes are not allowed. The system also makes the assumption that the blocks do not have any internal details.

Their system handled the dynamic loading and unloading of models by prioritizing models based on the view frustum and distance from the user. This was done by utilizing a k-d tree [4] to organize and query nearby models efficiently. When the system detected that a model was left behind in a scene, it disposed the model completely to save the client machine's memory resources. The authors noticed that with their dataset, during the dynamic loading and unloading of assets, the frame rate dropped to below five even on a gaming laptop. They identified the cause to be the synchronous loading and parsing of models on the client side, which is in line with the findings of Limper *et al.* [25] and Li *et al.* [23].

Slocum *et al.* [33] did work regarding performance optimization in a WebGL-based virtual reality environment. They wanted to provide a more general solution in comparison to the city model-based optimization presented by Schilling *et al.*, which works best when there are a large number of relatively simple 3D models to be rendered. Slocum *et al.* introduced a library named VIA. VIA's goal is to reduce the amount of data the VR environment has to initially load to a scene and provide a prioritization mechanism for the application to use when deciding what data to load next. It works by giving each object in a scene two scores – a visibility score and an angle score. The visibility score is given based on how much of the object is in the user's field of view, and the angle score represents how close the object is to the center of the user's field of view. Based on these two scores the system then prioritizes the models to be rendered for the user. Even though VIA was designed with virtual reality applications in mind, the concept is fully compatible with traditional 3D applications. Similar to 3D tiles, VIA needs to have some bounding volume information for each 3D model that the application might render at some point. This means that *e.g.* a database is needed to store the bounding volume information so that it can be accessed separately from the geometry itself. As mentioned earlier, the overhead of storing bounding volume information is low.

3 System Architecture

We present an overview of the system developed in this thesis. The goal of the system is to evaluate the impact of different optimization techniques (reviewed in Chapter 2) on the performance of rendering large 3D objects in a web-based application. Our system allows to evaluate the performance gain of different configurations separately as well as together. Our system performs the benchmarking by animating automatically around a set of *checkpoints*. Each checkpoint consists of an identifier, a three dimensional vector describing its position in the scene and has one or several models attached to it as well. The animation sequence is the same regardless of the configuration. Each configuration affects the behaviour of the system in its own way. Figure A describes how each checkpoints appears in the benchmark platform.

3.1 Configurations

After performing a survey on the related work we selected the configurations to assess the impact of the following optimizations in regards of performance and usability:

- Usage of optimized file format against unoptimized one
- Downloading and displaying only relevant data in detail *i.e.* filtering
- Downloading models progressively

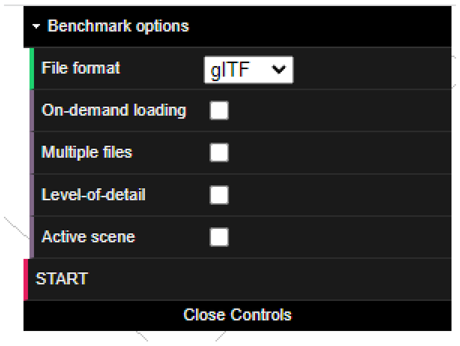


Figure 3.1: The configuration menu presented before benchmarking.

- Managing the amount of data that is rendered *i.e.* reducing the amount of operations the CPU and GPU has to do on each render call.

The configuration menu shown to the user can be seen in figure 3.1. Below is a more detailed explanation on what each configuration option does.

File format The first configuration menu allows the user to switch between two file formats. These are glTF and JSON. They were chosen since glTF is the recommended format for 3D data and JSON can be seen as a standard way of transmitting data to a web application [21]. As mentioned in the description of the viewer, the application will initially load direct download links to both file formats. Depending on the selected option, the application will load the models to the scene using the selected format's download URLs.

GlTF is loaded by using Three.js' GLTFLoader. It offers functionality to download and parse glTF asynchronously from a URL. After the file has been loaded and parsed by the loader, the resulting Three.js objects are added to the scene.

In case JSON is selected, the application simply downloads the JSON data from S3 and then creates the Three.js objects during runtime. The data is a list of objects where each object represents a single submesh of the model. Each object consists of two flat lists and color information. The first list is structured so that six subsequent items represent one vertex on the mesh. The first three represent the position of the vertex as a three-dimensional vector and the remaining three represent the normal of the vertex as a three-dimensional vector. The second list is a flat list of integers that represent the triangle faces so that three subsequent values on the list form one single face. A Three.js BufferGeometry is composed of this information. A material is created for each mesh according to the color information and the resulting THREE.js object is then added to a common parent object. After all the meshes have been created the parent object is added to the scene.

On-demand loading If this option is selected, the system will calculate the distance between the camera and each checkpoint in 500 millisecond intervals. The system will start to load the models for a checkpoint only if this check determines that a checkpoint is close enough. If the option is not selected, the system will just start to load all the models into the scene when the benchmarking sequence begins.

This is a much simpler version of spatial querying than those discussed in Chapter 2. In our benchmarking system, the number of models is relatively low, the camera is constantly moving and the checkpoints are placed far from each other. In a larger and more interactive 3D application some type of spatial indexing and a more refined function for determining models that need to be loaded should be implemented.

Multiple files To overcome glTF's limitation of not supporting progressive loading, we split each model into multiple small parts and save each of those as its own glTF-file. When this option is selected, the system will load all the models using the list of download URLs. This allows the system to already render a part of the model before the whole model has been downloaded from the server. Additionally it allows finer detail in querying and filtering only relevant data, since only one subset of the each model could be loaded if needed.

Level-of-detail In addition to the standard glTF representation, each model has a simplified version of its geometry stored in a separate glTF file. The process for generating this file is described in the system overview. When this option is selected, the system will check the distance between the camera and each model in 500ms intervals. Based on this distance, the models will be divided into two groups: "low" and "full". The system will then load either the low-detail or the full-detail version of each model, depending on the group it is in. If a model that was previously in the "low" group moves into the "full" group, the system will first load the full-detail version of the model from S3 and then make the low-detail version invisible and add the full-detail version to the scene. The low-detail model is kept in memory but excluded from the rendering pipeline. If a fully loaded model is marked as low by the check, the full model will then have its visibility flag set to false and the low model's visibility flag is set to true. This approach was chosen based on the recommendation of Alatalo *et al.* [2] who noted that the loading and unloading of models from the scene was a big performance bottleneck. In a more large-scale system some type of dynamic unloading is required to prevent the memory usage from growing as the application is used.

Active scene When this option is active, a third periodic check is activated. This check goes through all individual meshes in the scene and determines based on their size and distance to the camera if they should be visible or not. Like the other checks, this one is too performed in 500 millisecond intervals. Each mesh is either categorized as visible or not visible. Each mesh's visibility flag will then be changed accordingly. As mentioned, Three.js does perform view frustum culling meaning that it will automatically clip geometry outside the camera's view frustum from the rendering pipeline but it will not perform occlusion culling. This means that with highly detailed all the finest details (for example, a single bolt with a diameter of one centimeter) contained within the view frustum are rendered fully even if the camera is hundreds of meters away from it. Active scene management is meant to reduce the load on the GPU by reducing the amount of geometry it needs to render each frame. This of course adds load to the CPU because it has to calculate the distance between the camera and each mesh in the scene. As Alatalo *et al.* suggested we offload this computation to a web worker which allows us to perform the additional computation without it hurting the performance of the main execution thread.

3.2 System overview

Our system comprises three distinct components. These are the converter, server, and viewer. They are described in detail next. The source code for all components is available at GitHub [20].

3.2.1 Converter

The converter is a JavaScript script that reads source files of 3D models and converts them into files that can be visualized in the viewer. The converter takes in a file path, parses the source file, and creates a set of JSON and glTF files from the geometry data. The converter's job is simply to generate files in all the desired formats, split the source file into a set of result files, and create a simplified version of the source file in some format that is used in the benchmarking. All of these goals could be achieved in multiple ways and some choice of source file format or the nature of the model might require them to be implemented differently than the system used in this thesis.

In this system, the converter reads an Industry Foundation Classes (IFC) file [15]. We chose this as the format since IFC had a Three.js compatible library called IFC.js and IFC files are usually used in the construction industry to contain high-detail models for different structures [17]. This high-detail nature offered a good starting point for benchmarking since the amount of geometry on each model is large. The converter could be implemented on any source file format that could be read to a list of vertices and triangle faces.

The file is then parsed by IFC.js which offers an API to read all the triangle meshes that are present in the file. IFC files might consist of hundreds or thousands of small triangle meshes and these meshes together form the actual structure that is represented in the file. After parsing the file and creating Three.js Geometries, two different lists are initialized. One list stores all the meshes as Three.js' Mesh-objects [26], and the other is a two-dimensional list that stores groups of meshes so that each group can be exported as its file.

The converter iterates over each triangle mesh that IFC.js parses from the source file. Each mesh will be first converted to a Three.js Mesh object and stored in the mesh list. Simultaneously the number of faces on each mesh is added to a cumulative value. After each mesh is converted into a Three.js mesh, the converter will check if a preconfigured threshold of 20 000 faces is reached. This is a threshold chosen simply to give us a set of fairly small files. If it is, the models up to that point are collected into an array and that array is added as an item to the multiple files array. The counter is then set to 0. In addition, we also calculate the bounding box of every individual mesh in the model at this point and keep track of the largest bounding box.

After the meshes have been converted and collected, the converter iterates through each mesh again and compares the size of its bounding box to the size of the largest mesh found in the previous iteration. If the size of the mesh's bounding box is at least 20% of the size of the largest bounding box, the mesh will be included in the simplified version. All the meshes that have a large enough bounding box are included in a second model which is then saved as the simplified version of the model.

At this point the converter creates three Three.js Group-objects [13] for the full model, the simplified model, and for each list in the multiple file list. All the Groups are then exported to either JSON by using JavaScript's built-in method for JSON serialization, or to glTF by using Three.js' GLTFExporter. This process is visualized in Figure 3.2.

3.2.2 Server

The server simply serves JSON data to the viewer. It has an endpoint for each checkpoint. The data that those endpoints return consists of a URL for the full JSON model, a URL for the full glTF model, URLs for multiple glTF models, a URL for the simplified glTF model, and the position of the checkpoint as a three-dimensional vector. The server-viewer relationship is depicted in figure 3.3.

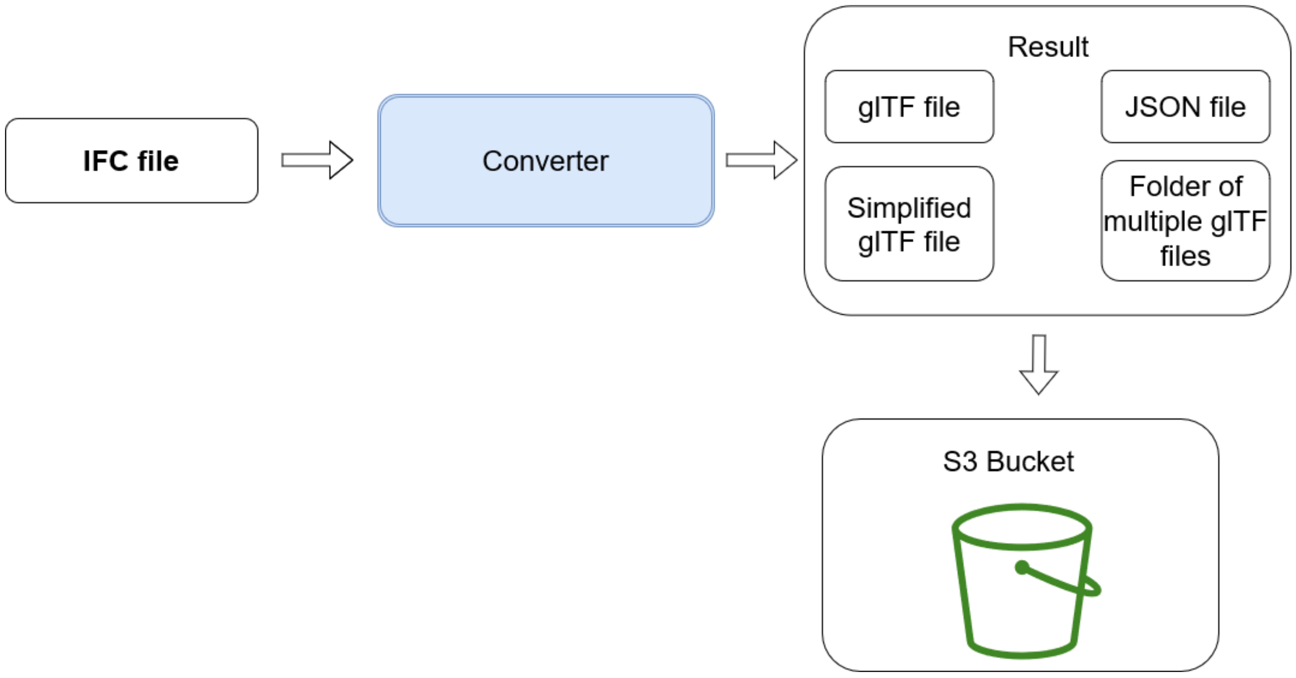


Figure 3.2: The converter

3.2.3 Viewer

The client side of the application is a standard JavaScript-based web application. It uses Three.js to render 3D scenes to the HTML5 canvas. As described by Limper *et al.* [24], Three.js is a general-purpose imperative framework built on top of WebGL. Initially, the user is presented with a configuration menu and a button to start the benchmarking (Figure 3.1). When start is pressed, the program first requests the data from all the checkpoints. After receiving the locations and download-URLs for each checkpoint the application will start the actual benchmarking. It will automatically move to each checkpoint in order. The application will turn the camera so that it faces a checkpoint and approach it in an animation that always takes 10 seconds. After reaching a checkpoint the camera will start to rotate itself in one-second-long animations around the checkpoint. After each animation, it checks if all of the data related to that checkpoint has been loaded and when it determines that it is, it will perform 20 more animations around the fully loaded model. This means that at minimum the application will rotate around a checkpoint for at least 20 seconds. This procedure is performed at every checkpoint. The program halts after each checkpoint has been handled.

Three.js has built-in view frustum culling [35] which can be taken advantage of for all the models that are in the scene. The models excluded from the rendering pipeline will still

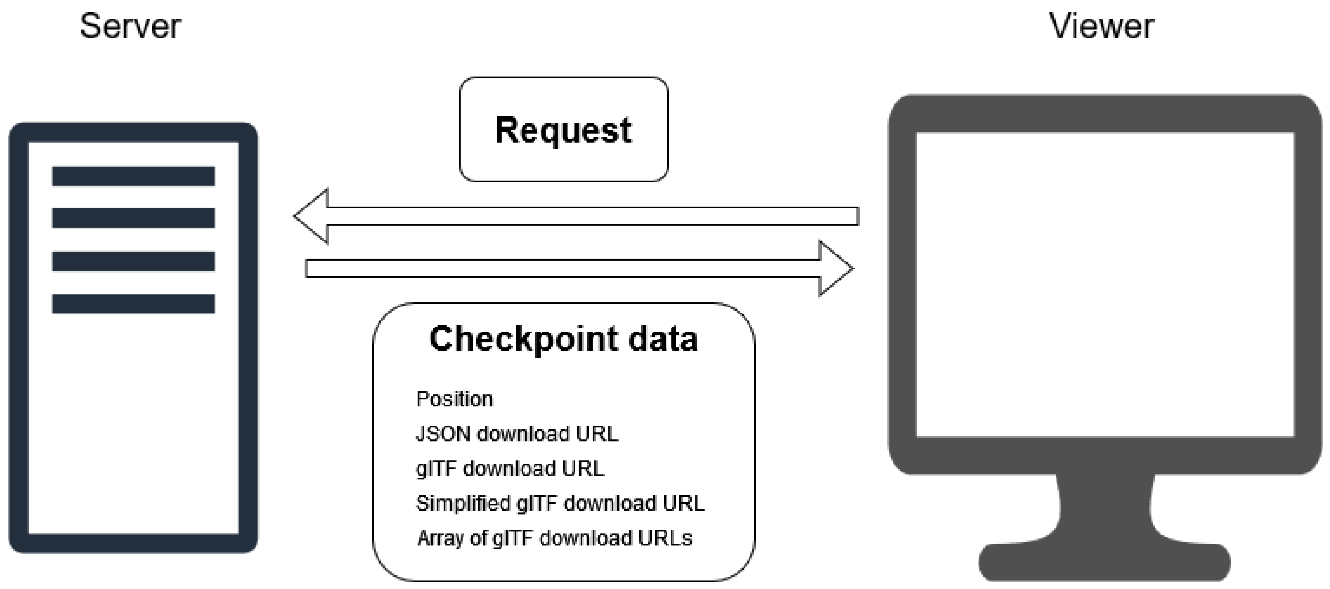


Figure 3.3: The relationship between the server and the viewer

reside in the memory but will not consume GPU's processing power.

4 Performance evaluation

This section describes the performance evaluation carried out with our benchmarking system described in Chapter 3. This section first describes the experimental setup, including the datasets and performance metrics. It then presents the results from the experiments, and discusses the impact of different configurations on rendering large 3D objects in a web application.

4.1 Evaluation methodology

4.1.1 Test dataset

Our benchmarking is carried out with five different checkpoints. Each checkpoint consists of one or several IFC files. All the source files used are listed in Table 4.1. In our experimental evaluation, we simply extract the geometry out of the source models. Since IFC is a standard used mainly in construction industry, IFC files generally have a lot of metadata and custom properties used by the designers and architects. Additionally a single geometry described in an IFC file can be represented multiple times in the visualization of a model. For example, a single bolt might appear in the visual representation hundreds of times even though it is only described once in the source file. Because of this the resulting geometry can be smaller or larger than the source depending on the amount of metadata and reused geometry.

Checkpoint	**File name**	**File size (MB)**	**Source**
1	0912102010-03-01 Project.ifc	49.5	IFC Repository [38]
2	ifcbridge-model01.ifc	14.8	ifcinfra.de [16]
3	IFC Schependomlaan	48.1	buildingSmart [5]
4	Duplex_A_20110907.ifc	2.3	buildingSmart [5]
	Duplex_M_20111024_ROOMS_AND_SPACES.ifc	8.4	
	Duplex_Plumbing_20121113.ifc	30.1	
5	Clinic_Architectural.ifc	12.4	buildingSmart [5]
	Clinic_HVAC.ifc	25.7	
	Clinic_Structural.ifc	18.2	

Table 4.1: List of the IFC models that were used in the benchmarking

4.1.2 Metrics

CPU usage We monitor the percentage of CPU resources that the process that runs the application uses. This is done by running a separate Python script simultaneously with the benchmarking. The script uses the `psutil` library [28] to monitor the number of CPU resources that the benchmarking application is using. It will collect this information every second until the script is terminated.

Memory usage. The memory usage statistics are collected by the same script used to collect CPU usage statistics, with values reported every second. However, instead of collecting the percentage of memory utilized, it collects the absolute amount of memory that the benchmarking process uses.

Frames-per-second (FPS). When the benchmarking is ongoing, we collect the FPS information in the browser. When the application is rendering the 3D scene, it calls the browser's built-in `requestAnimationFrame` function [43] and gives its render call as a callback to the function call. The browser then renders the scene when it determines that it has sufficient resources to do so. The FPS values are collected as follows. Each time the `requestAnimationFrame` is called, we increment a counter. Simultaneously we have an interval running once every second that saves the value of the counter and resets it to zero. FPS is good metric for evaluating the user experience. Generally, if FPS drops below 10 the application becomes noticeably unresponsive and harder to use. We consider an FPS of over 30 to be a good value where the user experience is still smooth.

Time to process. From the user experience perspective, it is important to get visual feedback when using an application. In the case of 3D applications, the feedback comprises visible changes in the scene after loading a model or moving around it. If the user tries to perform any action on the scene without seeing any changes, the user experience suffers. Loading new models to the scene either because the user explicitly requested it or by reacting to something that the user did such as moved in the scene, needs to be as fast as possible. As mentioned in Chapter 1, WebGL applications have to generally transmit a lot of data and use a lot of computing resources. These may lead to an extended period of inactivity in the scene from the user's perspective since a model has to be downloaded and processed on the client side before it can be added to the scene. The amount of time it takes to show a change in the scene is an important metric regarding the user experience. Accordingly, we capture this metric as follows. When the application starts to download data for that particular checkpoint a start time for that checkpoint is logged. After the data is received from the server the application then processes the data to create the geometry. That geometry is then loaded to the scene and at this point the application logs the time it took to download and process the model. If a checkpoint consists of several models the time is calculated from the point in time when the application starts to download the first model for that checkpoint to the point in time where every model of that checkpoint has been loaded to the scene.

4.1.3 Benchmark setting

For the benchmarking, the server was configured to return the positions of the checkpoints so that they form a pentagon. The placement of the checkpoints is depicted in Figure A.2. The benchmarking was run on a computer with an AMD Ryzen 5 3600 6-Core CPU running on a frequency of 3.59 GHz, an NVIDIA GeForce GTX 1660 Super GPU, 16GB RAM and running on a 64-bit Windows 10 Pro operating system. According to Speedtest.net, the network download bandwidth was 39.38Mbps during the time of benchmarking. Each configuration is run ten times and results are gathered from each iteration.

4.2 Results

We run experiments to evaluate the impact of the different configurations listed in Section 3.1. In total, there are six tested configurations with different combinations of the on-demand, multiple files, level-of-detail and active scene management optimizations. Initially we benchmarked only the difference between the file formats. The difference in the processing speed became apparent immediately, as can be seen in figure 4.6, so we performed the benchmarking by adding optimization techniques sequentially to form each configuration.

Table 4.2 describes the abbreviations for the different configuration settings.

Table 4.3 shows the size of the geometry files produced by the converter. A lower value is desirable for this metric, as it represents a smaller amount of data transferred over the network. For the remaining metrics, the experiments are carried out 10 times. Figure 4.1 shows the average FPS as the function of time over all the iterations. Minimum and maximum values are visualized as the shaded area behind each line. For FPS higher

OD	On-Demand
MF	Multiple files
LOD	Level-of-detail
AS	Active scene management

Table 4.2: Abbreviations used in the plots

Checkpoint	JSON size (MB)	glTF size (MB)	Multiple glTF size (MB)	Low LOD glTF Size (MB)
1	74.5	21.9	19.5	0.2
2	40.6	12.4	12.1	6.8
3	43.9	22.3	21.4	1.3
4	210.5	52.0	52.9	3.9
5	292.1	100.9	96.4	1.4

Table 4.3: List of the geometry file sizes that the converter produces

result is better. Figure 4.2 plots the average CPU usage as the function of time and the minimum and maximum values similar to the FPS results. In this case the lower result is better. Average memory usage is visualized in figure 4.3. Figure 4.6 shows the average time taken for each configuration to both download and process the model before it was added to the scene for each checkpoint. The bar represents the average over all iterations and minimum and maximum values are represented as the error bar. We describe each configuration separately and evaluate their performance for the considered metrics next.

4.2.1 JSON

As we can already see from Table 4.3, the JSON files have a substantially larger size than their glTF counterparts even though they describe the same geometries and materials. In JSON everything is represented as normal text, as opposed to glTF where geometry data is encoded in binary format. This effect alone causes the JSON files to be over three times as large as the glTF files on average. This directly affects the amount of data that needs to be transmitted over the network.

From the FPS results (Figure 4.1a), we can see that the FPS drops close to 10, several times during the experiments. In the worst cases, it goes all the way down to 0. The difference between the average and the minimum and the maximum is the largest with JSON as compared to the other configurations. This is expected since it is the only configuration where the application needs to process the data on the client side. Because every checkpoint is loaded to the scene at once in this configuration, each individual checkpoint processing is competing with one another. Depending on the CPU time each processing gets from the JavaScript engine the average process times might differ a lot between iterations.

Since this configuration simply starts loading every checkpoint to the scene, the memory usage (Figure 4.3) rises as long as models are being loaded to the scene. After all the models have been loaded, the memory usage lowers as the application can free some buffers that are in use when loading and parsing the JSON data to a Three.js Mesh. The peak memory usage is higher than in any other configuration.

The CPU usage (Figure 4.2a) follows the same pattern as the FPS statistics, with the difference being that instead of having dropped, it has spikes. The spikes behave similarly to the drops in the FPS statistics so that they appear when models are being loaded and are more significant when more checkpoints have been loaded. The variance in CPU usage is much higher with JSON than it is with the other configurations.

Even though this configuration did not result in an increased amount of computational resources to be used, the largest weakness this configuration has is the amount of time that it takes to show a model after starting to download it. In addition to the larger amount of data that needs to be downloaded, the parsing phase, where a Three.js Mesh is constructed from the JSON data, takes considerably longer than in any other configuration. This causes the user to see an extended period of inactivity where nothing is loaded on the scene. Figure 4.6 shows that the difference in processing times compared to glTF was considerable, with JSON taking several times as long as glTF to be loaded. Since JSON was only benchmarked in a configuration without on-demand loading, multiple files, active scene, or level of detail, each checkpoint was competing against each other for computing resources making the minimum and maximum result significantly different from the average.

4.2.2 glTF

In addition to having significantly lower file size when compared to JSON (Table 4.3), we can see from Figure 4.6 that glTF is significantly faster to load the scene then JSON. Interestingly even though the processing times were significantly shorter with glTF, the CPU usage spikes are similar when loading glTF even though it can be loaded to the GPU without any additional processing in between.

After the model is loaded there does not seem to be much difference in the computing resource utilization (Figure 4.2b) when compared to JSON. Sometimes the performance drops were even more significant with glTF than with JSON. This is possibly due to the fact that since geometry stored in glTF does not need any preprocessing the application was loading much more data to the GPU's buffers at a single points of time which seems to be the heaviest part of the processing as can be seen from figures 4.4 and 4.5.

When loading all of the files simultaneously without any scene management, Figure 4.1b shows that the FPS drops at the beginning when all of the checkpoints are loaded. Similarly to the JSON configuration the checkpoints are loaded simultaneously and therefore are competing for computing resources. The variance is still quite large with a lot of points in time where in the worst scenario the FPS dropped below 10.

4.2.3 glTF On-demand

This configuration requires periodically checking each checkpoint's distance to the camera; however, this overhead is not noticeable in the CPU usage. As we can see from Figure 4.4 most of the spikes seem to be from the loading of the model to the GPU buffers. We can also see that the CPU utilization (Figure 4.2c) is generally lower than what it was with plain glTF and the spikes are not as high. The spikes are naturally spread out more evenly throughout the runtime of the benchmarking since each checkpoint is loaded separately.

When comparing this configuration to the plain glTF solution, we can see that the memory usage (Figure 4.3) is rising in a much more gradual manner. This is expected since now checkpoints are loaded to the application one at a time, when necessary. The FPS drops (Figure 4.1c) do not happen in the same extent as with plain glTF or JSON until much later when the larger checkpoints are being loaded.

To visualize the correlation between events, such as checkpoints being loaded to the scene, and performance metrics we chose one individual iteration to serve as a test run. The configuration we used to plot this was glTF On-Demand since it is the most basic configuration where each checkpoint is loaded individually. The annotated test run's CPU usage can be seen in figure 4.4 and its FPS statistics are presented in figure 4.5.

We can see clearly in both figures, that the largest drops in performance happens exactly when the application loads the models to the scene. After downloading the glTF model from the server, the application has the geometry in memory. From there it will transfer the data to the GPU so that it can be rendered on the canvas, and this seems to be the heaviest part of the application process.

4.2.4 glTF On-demand Multiple files

Splitting the 3D data to multiple small files seemed to also bring some improvements when compared to the previous configurations where each model was represented as one large file. The reduction in the amount of time it took to download and process the models is small but noticeable in Figure 4.6. The FPS metrics are pretty much the same when compared to the previous configuration with on-demand loading and a single file representations. We do see in figure 4.2d that the average CPU usage peak is lower when using multiple files to represent single models.

One not easily quantifiable benefit from this configuration is that some parts of the model can be made visible to the user before the whole model has been loaded. This makes the application feel more responsive since the user can see the scene changing faster.

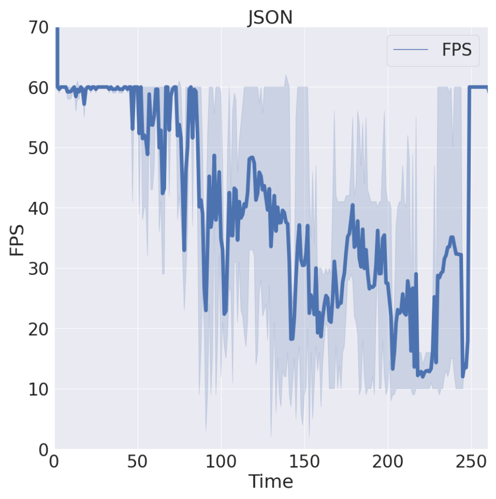
4.2.5 glTF - On-demand - Multiple files - Level-of-detail

In this configuration, full models are only shown when the camera is close enough to the models. This reduces the amount of data that goes through the rendering pipeline each frame with the goal of improving FPS. This is the first configuration where we can clearly see (Figure 4.1e) that the FPS stays at a high level consistently. We do still see some drops presumably when the larger models are loaded but on average the FPS remained higher than 20 at all times during the benchmarking. The drawback of this configuration are that each model representation gets an additional file that is the lower level-of-detail version of the model and that the application has to continuously check which models need to be shown fully and which can be represented by the lower level-of-detail version. However these drawbacks are not noticeable in the CPU (Figure 4.2e) or memory usage (Figure 4.3e). The improvements of this configuration are clear in the FPS statistics, meaning that unsurprisingly having a level-of-detail mechanism is encouraged in a browser environment.

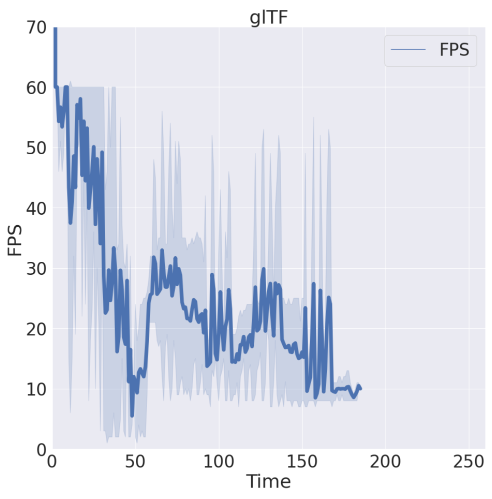
We could also see that on average, the drops in the FPS charts were not as significant with the configurations that managed the amount of models rendered to the scene.

4.2.6 glTF - On-demand - Multiple files - Level-of-detail - Active scene

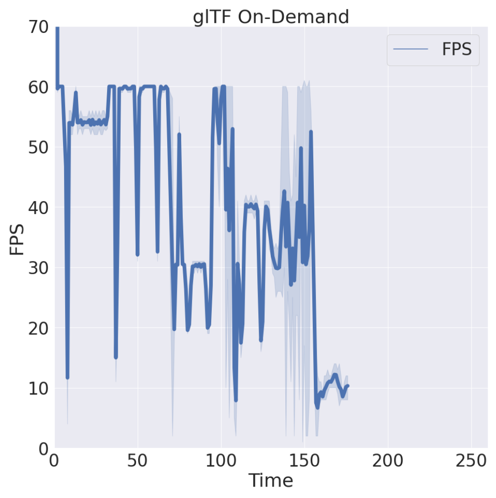
This configuration aims to reduce the number of triangles in the scene, similar to the previous configuration. It also uses level-of-detailing, but in addition, it removes small objects from the rendering pipeline on the client side after the model has been loaded. Since this is done on the client side, it does not affect download and processing performance, as can be seen from (Figure 4.6). The FPS charts show improvements when using the active scene management option, and there is virtually no difference in CPU utilization and memory usage compared to the previous configuration without this option. This configuration introduces additional calculations for determining which models are visible and which are not, but these do not significantly impact performance. There is a drop in FPS when loading the last checkpoint, but on average, the FPS remains higher than with any other configuration. However the additional improvement this configuration brings when compared to the previous one can only be seen on average during the loading of the last checkpoint and there the difference is on average around five frames per second. This configuration might impact the quality of the scene by hiding smaller objects. It is the decision of the application developer if this small performance improvement is worth the price of having an impact on the scene quality in terms of having objects hidden.



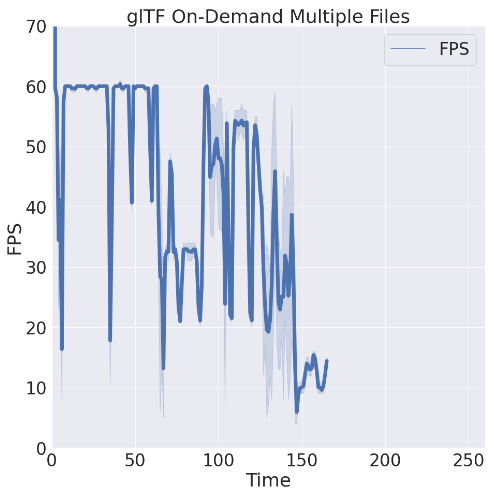
(a)



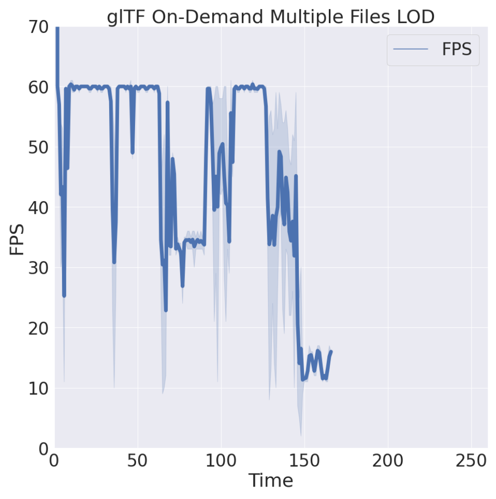
(b)



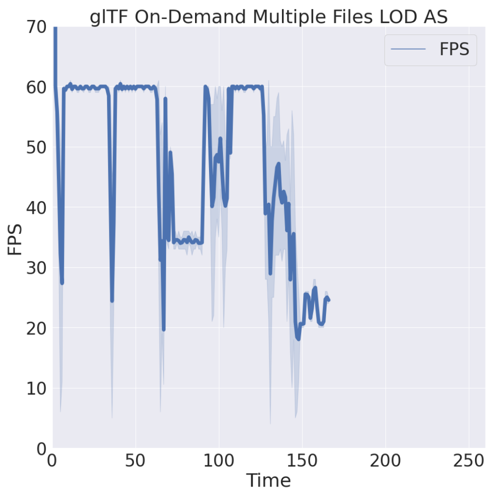
(c)



(d)

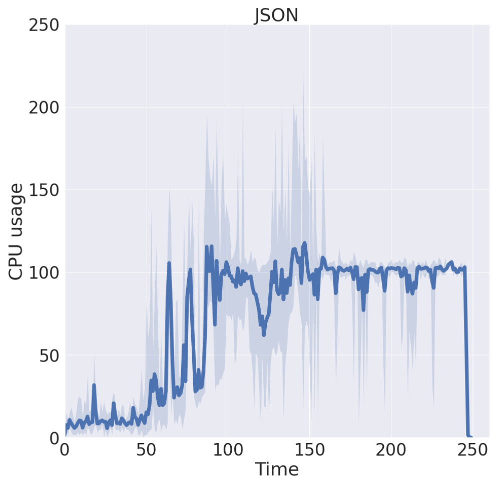


(e)

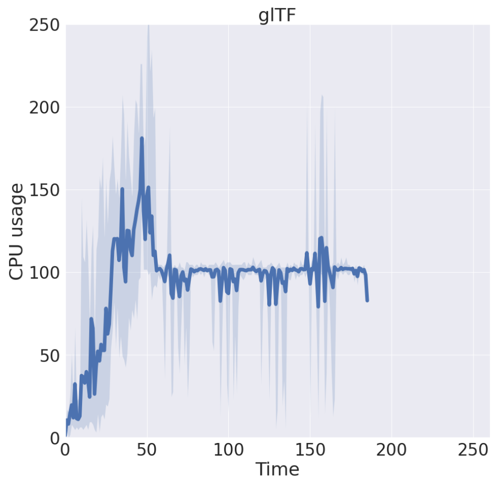


(f)

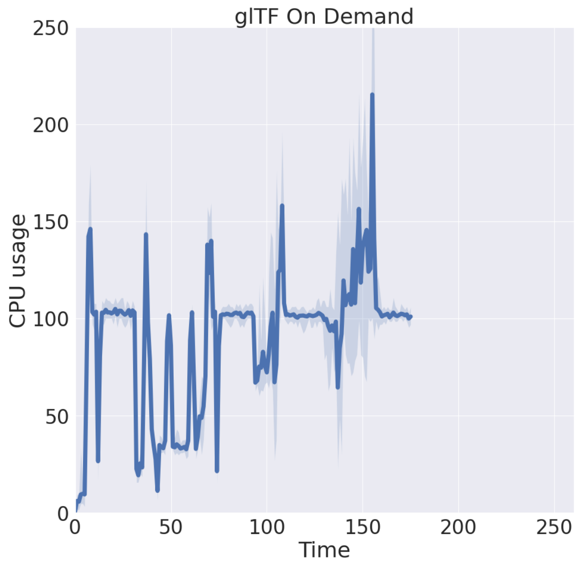
Figure 4.1: Average FPS statistics for each configuration



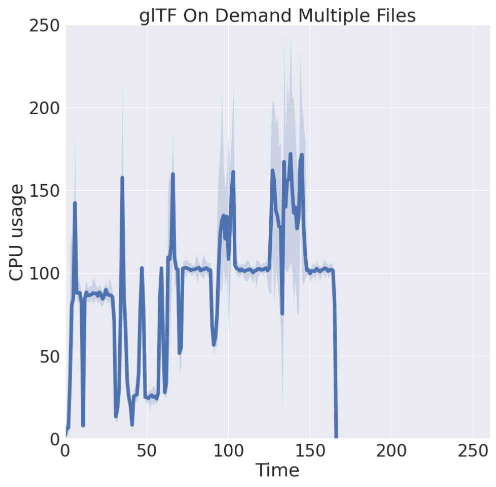
(a)



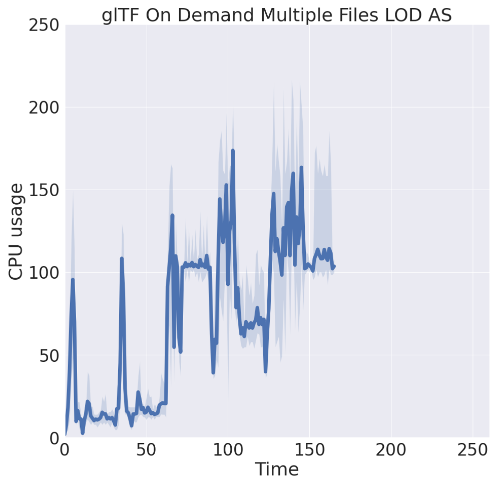
(b)



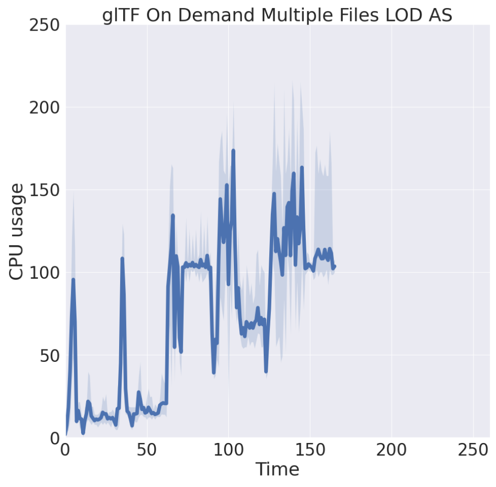
(c)



(d)

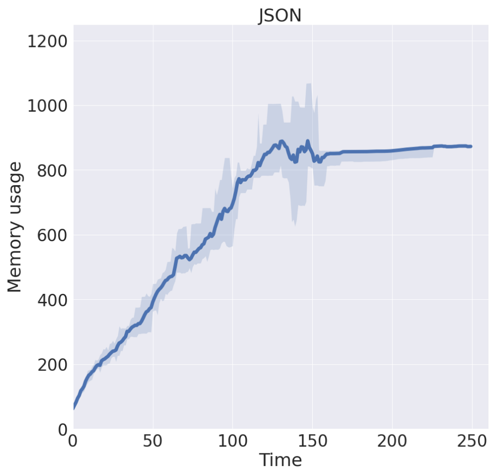


(e)

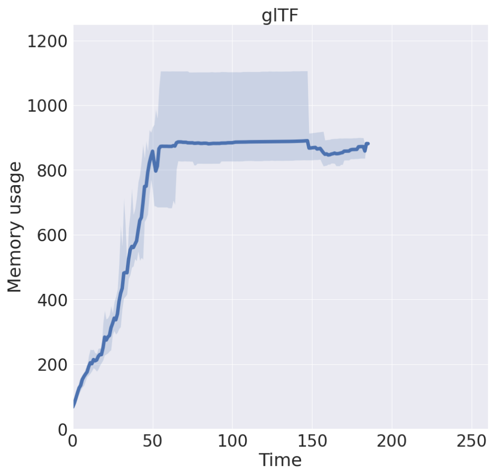


(f)

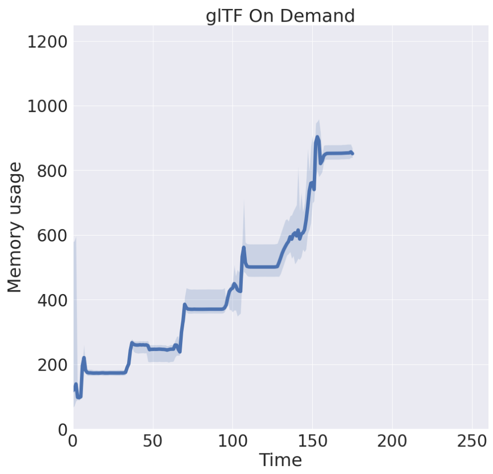
Figure 4.2: Average CPU usage for each configuration



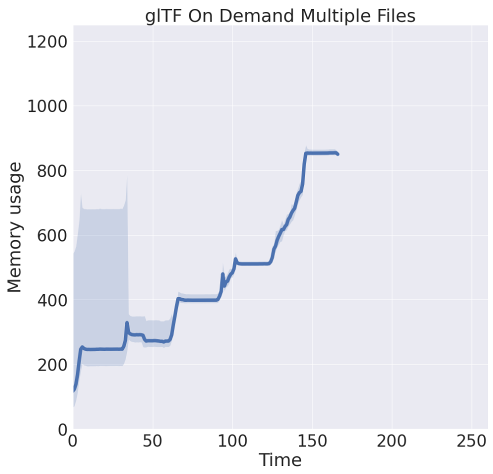
(a)



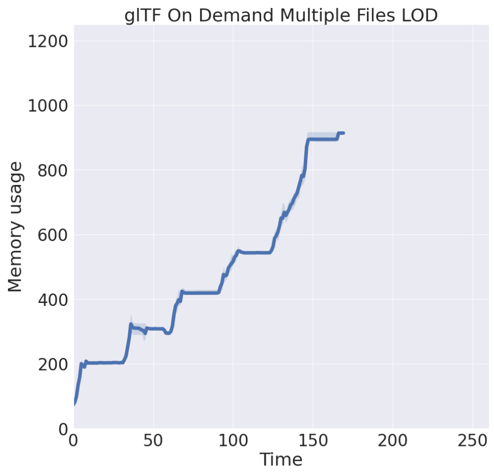
(b)



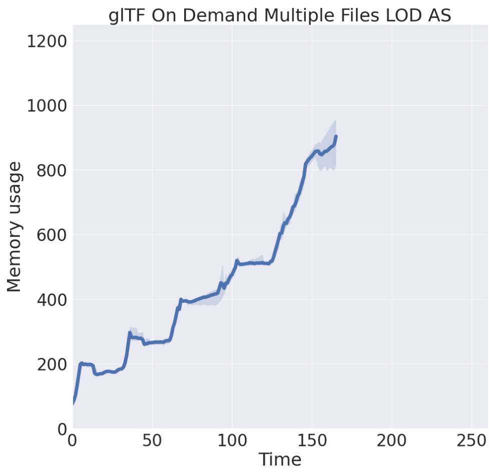
(c)



(d)



(e)



(f)

Figure 4.3: Average memory usage for each configuration

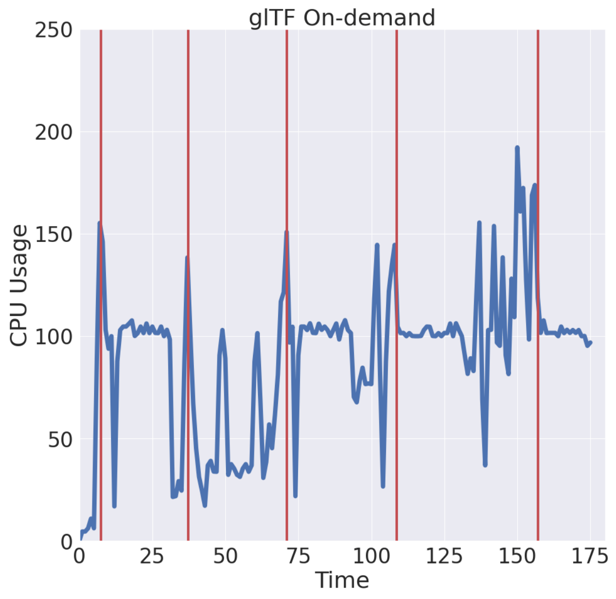


Figure 4.4: Individual test run's CPU usage. Red vertical lines represent the points when a model was added to the scene

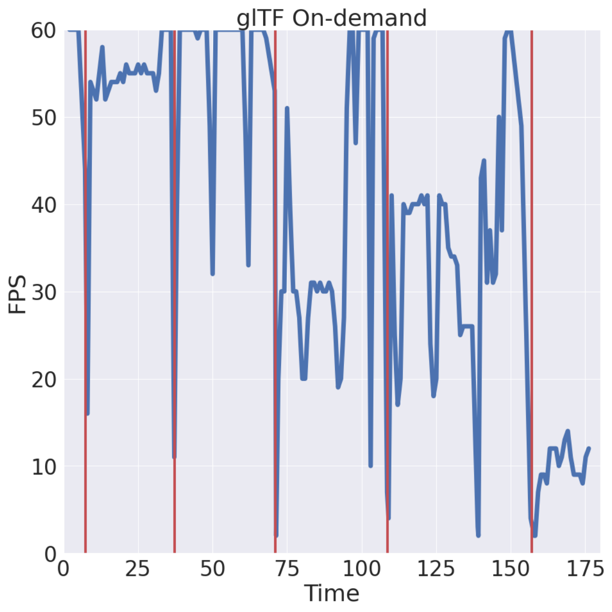
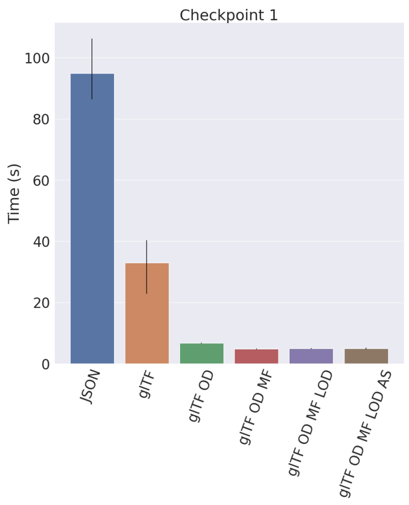
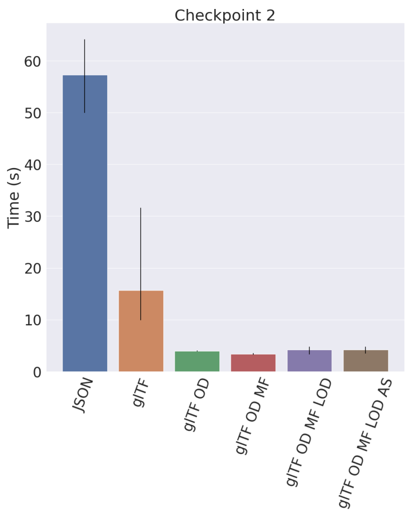


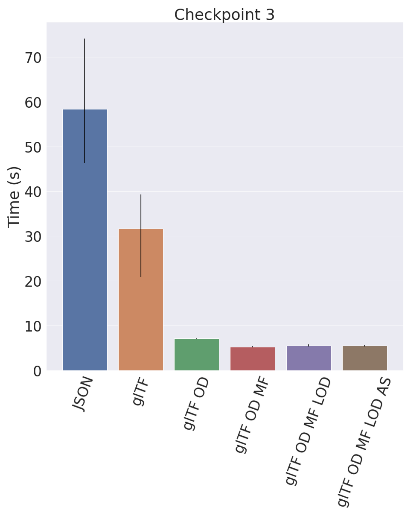
Figure 4.5: Individual test run's FPS statistics. Red vertical lines represent the points when a model was added to the scene



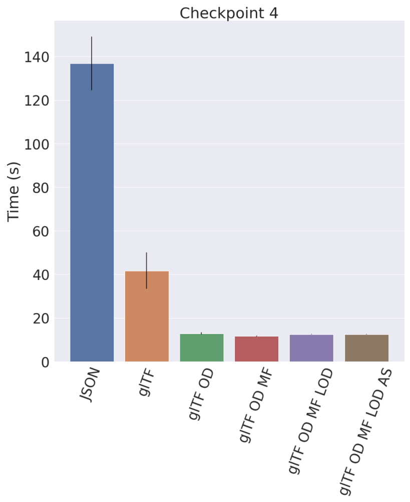
(a)



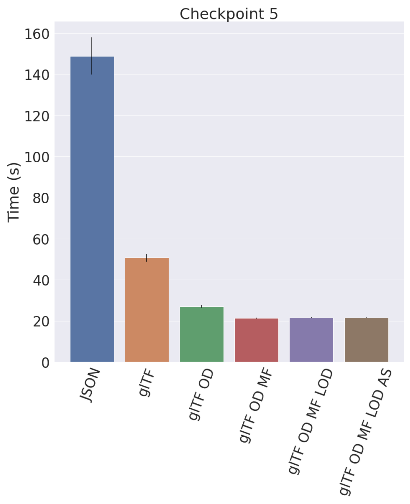
(b)



(c)



(d)



(e)

Figure 4.6: Average processing times for each checkpoint. Error bars represent the minimum and maximum time it took to process each checkpoint with each configuration.

5 Discussion and future work

5.1 Data format - transmission and processing

In terms of storing the data that is used by a WebGL application, our results show that an optimized format that allows as little processing on the client side, such as glTF brings significant reductions in the time that a user has to wait for a model to load. This was also observed by Limper *et al.* in their case study [25]. After the data has been loaded to the scene there seems to be no difference in performance between different file formats, which was to be expected since the data already resides in the GPU's buffers at this point.

We observed significant drops in performance just when models were loaded onto the scene. This effect was also reported by Alatalo *et al.* [2]. One thing to note is that the performance drop occurred with both file formats. Even though glTF was much faster to process, the number of computing resources it took to load the processed data to the GPU and the resulting drop in performance was as significant with it as when loading plain JSON. Because of this, we recommend designing WebGL applications in a way that minimizes the number of times it has to download new models to the scene. Especially we recommend that situations, where a model is removed from the scene and then loaded again soon after, are avoided.

Google has released a library called Draco, which is purely meant for compressing 3D data [8]. It can be wrapped around glTF to produce even smaller files to reduce the storage size and the amount of data that needs to be sent over the network. Draco requires an additional decoding process on the client side when compared to glTF. This goes against the suggestion of Limper *et al.* [25] of preferring data formats that require as little processing in the client application as possible, and letting the browser take care of compressing and caching data. However, the study was published before Draco was published. Since Draco is directly targeted to be used with both storage and transmission of 3D data, it would be beneficial to test how it impacts the download and process times.

5.2 Rendering performance

Even though the largest performance drops were detected when the application was loading the models to the scene, the number of triangles that needed to be drawn had a noticeable impact on the FPS in our experiments. This was expected since less data is going through the rendering pipeline. This creates a bit of a dilemma since it would be beneficial to reduce the number of triangles on the scene with *e.g.* level-of-detailing, switching between different versions of models dynamically every time the user's viewport changes. However, these mechanisms result in a large number of loading and unloading of models which were the cause of the largest drops in performance. In the benchmarking application used in this thesis, a scene's size was fixed and the number of separate models was low. This allowed the level of detailing to work in a way where everything was loaded to the scene upfront and the visibility flags of different versions were manipulated by the application. However, this approach would lead to higher memory usage if used in an application with a large number of models in the scene. We also noticed that the performance drops that were experienced during the loading of models, were not as significant when there was some scene management in place to limit the amount of data that is rendered.

There is a need for a scene management system that could dynamically manage the resources that are loaded to the GPU, in a manner that minimizes the amount of times data needs to be downloaded from the server and processed. 3D tiles specification, which was mentioned in Chapter 2, is the only WebGL-specific specification to our knowledge that tackles these issues. As of now, the 3D tiles specification is tightly coupled with the Cesium platform even though it is an open specification. However, there are several ongoing projects to create Three.js support for 3D tiles such as 3DTilesRendererJS by NASA [27]. Cesium has also published a new version of the specification, labeled 3D Tiles Next, where they plan to simplify the architecture by utilizing glTF directly without the need to wrap it in their B3DM format [1].

6 Conclusion

We found that using an optimized file format such as glTF can reduce the time it takes to display a 3D model to the user. However, it did not significantly decrease the computation resources used or improve FPS levels compared to using JSON. The greatest performance decline occurred during model loading. The impact of the performance drop was less significant when the amount of rendered data was managed using a level-of-detailing system. We observed that the performance decline was directly correlated with the size of the model being loaded, with larger models having a greater impact on performance than smaller ones

Here are our reference guidelines for tackling performance issues with WebGL applications:

- Utilize an optimized file format such as glTF.
- Load as little 3D data as possible at any given time. The more data is loaded, the more significant is the drop in the FPS and spike in the CPU usage. All excess loading should be avoided such as loading a detailed model that is far away from the user.
- Avoid situations where the same model is first unloaded and then loaded back to the scene.
- Manage the amount of data that is visible to the user. Hide unnecessary details when possible. This reduces the negative effect that loading models to the scene has to the performance.

Bibliography

[1] *3D Tiles Next*. https://cesium.com/blog/2021/11/10/introducing-3d-tiles-next/. Accessed: 2022-11-23.

[2] T. Alatalo, T. Koskela, M. Pouke, P. Alavesa, and T. Ojala. "VirtualOulu: collaborative, immersive and extensible 3D city model on the web". In: *Proceedings of the 21st International Conference on Web3D Technology*. 2016, pp. 95–103.

[3] U. Assarsson and T. Moller. "Optimized view frustum culling algorithms for bounding boxes". In: *Journal of graphics tools* 5.1 (2000), pp. 9–22.

[4] J. L. Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[5] *buildingSmart sample files*. https://github.com/buildingSMART/Sample-Test-Files/tree/master/IFC2x3. Accessed: 2022-10-26.

[6] *Cesium 3D Tiles specification*. https://github.com/CesiumGS/3d-tiles/tree/main/specification. Accessed: 2022-01-20.

[7] K. Chaturvedi, Z. Yao, and T. H. Kolbe. "Web-based Exploration of and interaction with large and deeply structured semantic 3D city models using HTML5 and WebGL". In: *Bridging Scales-Skalenübergreifende Nah-und Fernerkundungsmethoden, 35. Wissenschaftlich-Technische Jahrestagung der DGPF*. 2015.

[8] *DRACO: 3D Data Compression*. https://github.com/google/draco. Accessed: 2022-11-22.

[9] *Extensible 3D (X3D) Graphics*. https://www.web3d.org/x3d/what-x3d. Accessed: 2022-01-16.

[10] *FBX - Adaptable file format for 3D animation software*. https://www.autodesk.com/products/fbx/overview. Accessed: 2022-05-12.

[11] M. Garland and P. S. Heckbert. "Surface simplification using quadric error metrics". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 1997, pp. 209–216.

[12] *glTF RUNTIME 3D ASSET DELIVERY*. https://www.khronos.org/gltf/. Accessed: 2022-01-16.

[13] *Group*. https://threejs.org/docs/?q=group#api/en/objects/Group. Accessed: 2022-06-12.

[14] H. Hoppe. "Progressive meshes". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, pp. 99–108.

[15] *IFC*. https://www.buildingsmart.org/standards/bsi-standards/industry-foundation-classes. Accessed: 2022-06-12.

[16] *IFC Infra*. https://ifcinfra.de/ifc-bridge. Accessed: 2022-10-26.

[17] *Ifc.js*. https://ifcjs.github.io/info/docs/introduction. Accessed: 2022-06-12.

[18] T. H. Kolbe, G. Gröger, and L. Plümer. "CityGML: Interoperable access to 3D city models". In: *Geo-information for disaster management*. Springer, 2005, pp. 883–899.

[19] M. Krämer and R. Gutbell. "A case study on 3D geospatial applications in the web using state-of-the-art WebGL frameworks". In: *Proceedings of the 20th international conference on 3d web technology*. 2015, pp. 189–197.

[20] *ktoiv/webgl-benchmarking*. https://github.com/ktoiv/webgl-benchmarking. Accessed: 2022-12-06.

[21] G.-h. Lee, P.-h. Choi, J.-h. Nam, H.-s. Han, S.-h. Lee, and S.-c. Kwon. "A Study on the Performance Comparison of 3D File Formats on the Web". In: *International journal of advanced smart convergence* 8.1 (2019), pp. 65–74.

[22] L. Li, X. Qiao, Q. Lu, P. Ren, and R. Lin. "Rendering optimization for mobile web 3D based on animation data separation and on-demand loading". In: *IEEE Access* 8 (2020), pp. 88474–88486.

[23] P. Li, X. Yu, and J. Wang. "Progressive compression and transmission of 3D model with WebGL". In: *2016 International Conference on Audio, Language and Image Processing (ICALIP)*. IEEE. 2016, pp. 170–173.

[24] M. Limper, Y. Jung, J. Behr, and M. Alexa. "The pop buffer: Rapid progressive clustering by geometry quantization". In: *Computer Graphics Forum*. Vol. 32. 7. Wiley Online Library. 2013, pp. 197–206.

[25] M. Limper, S. Wagner, C. Stein, Y. Jung, and A. Stork. "Fast delivery of 3d web content: A case study". In: *Proceedings of the 18th International Conference on 3D Web Technology*. 2013, pp. 11–17.

[26] *Mesh - Three.js docs*. https://threejs.org/docs/#api/en/objects/Mesh. Accessed: 2022-06-12.

[27] *NASA 3D Tiles Renderer JS*. https://github.com/NASA-AMMOS/3DTilesRendererJS. Accessed: 2022-11-23.

[28] *psutil documentation*. https://psutil.readthedocs.io/en/latest/. Accessed: 2022-12-06.

[29] A. Schilling, J. Bolling, and C. Nagel. “Using glTF for streaming CityGML 3D city models”. In: *Proceedings of the 21st International Conference on Web3D Technology*. 2016, pp. 109–116.

[30] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. “Decimation of triangle meshes”. In: *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*. 1992, pp. 65–70.

[31] T. Scully, S. Friston, C. Fan, J. Doboš, and A. Steed. “glTF Streaming from 3D Repo to X3DOM”. In: *Proceedings of the 21st International Conference on Web3D Technology*. 2016, pp. 7–15.

[32] A. Selçuk, U. Güdükbay, and B. Özgüç. “Walkthrough in Complex Environments at Interactive Rates using Level-of-Detail”. In: *Turkish Journal of Electrical Engineering and Computer Sciences* 10.1 (2002), pp. 57–72.

[33] C. Slocum, J. Huang, and J. Chen. “VIA: Visibility-aware Web-based Virtual Reality”. In: *The 26th International Conference on 3D Web Technology*. 2021, pp. 1–9.

[34] *STL (STereoLithography) File Format Family*. https://www.loc.gov/preservation/digital/formats/fdd/fdd000504.shtml. Accessed: 2022-05-12.

[35] I. Sukin. *Game development with Three. js*. Packt Publishing Ltd, 2013.

[36] K. J. Theisen. “Programming languages in chemistry: a review of HTML5/JavaScript”. In: *Journal of Cheminformatics* 11.1 (2019), pp. 1–19.

[37] *three-loader-3dtiles*. https://github.com/nytimes/three-loader-3dtiles. Accessed: 2022-11-23.

[38] *University of Auckland Open IFC Repository*. http://smartlab1.elis.ugent.be:8889/IFC-repo/http.openifcmodel.cs.auckland.ac.nz/. Accessed: 2022-10-26.

[39] *Wavefront OBJ*. https://www.fileformat.info/format/wavefrontobj/egff.htm. Accessed: 2022-01-16.

[40] *Web Assembly*. https://developer.mozilla.org/en-US/docs/WebAssembly. Accessed: 2022-09-12.

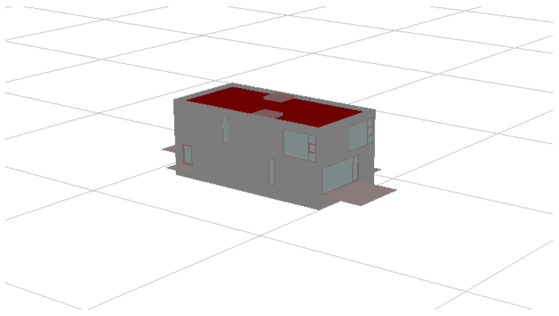
[41] *Web Worker*. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. Accessed: 2022-06-12.

[42] *WebGL*. https://www.khronos.org/webgl/. Accessed: 2022-02-11.

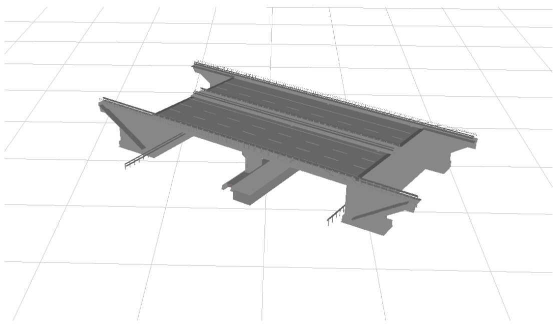
[43] *window.requestAnimationFrame: Mozilla MDN Docs*. https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame. Accessed: 2022-10-26.

[44] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III. "Visibility culling using hierarchical occlusion maps". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 1997, pp. 77–88.

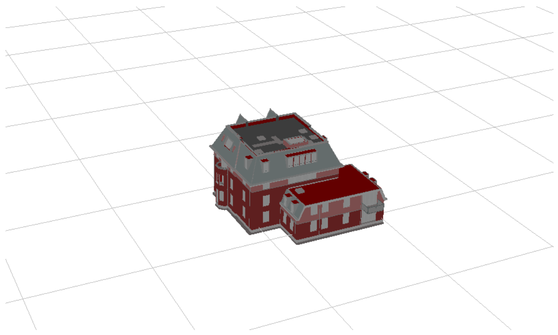
A All checkpoints visualized



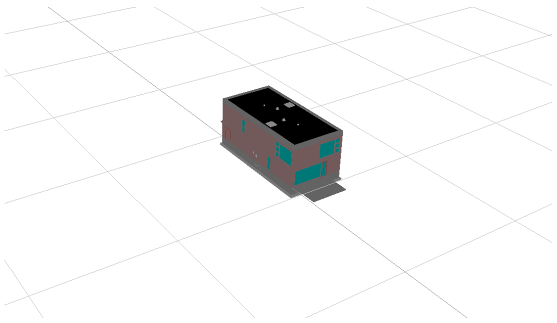
(a)



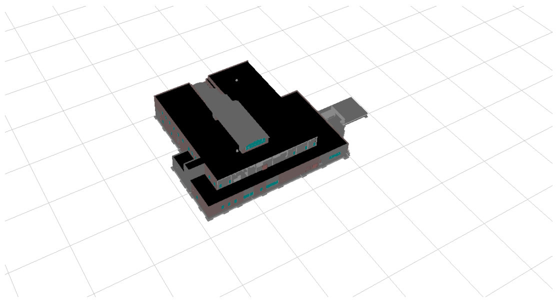
(b)



(c)



(d)



(e)

Figure A.1: Every checkpoint in order

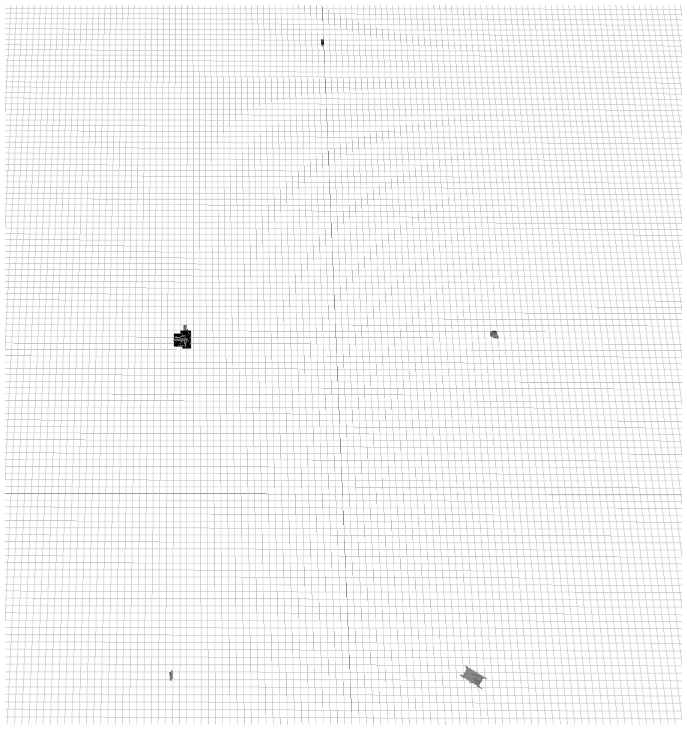


Figure A.2: Placement of the checkpoints in the benchmarking application. Checkpoint 1 is the one on bottom left. Checkpoints are placed in an anti-clockwise order