Progressive Network Streaming of Textured Meshes in the Binary glTF 2.0 Format

Peer-reviewed author version

LEMOINE, Wouter & WIJNANTS, Maarten (2023) Progressive Network Streaming of Textured Meshes in the Binary glTF 2.0 Format. In: Posada, Jorge; Moreno, Aitor; Jaspe, Alberto; Muñoz-Pandiella, Imanol; Stricker, Didier; Mouton, Christophe; Mohammed, Ayat; Elizalde, Ane (Ed.). Proceedings of the 28th International ACM Conference on 3D Web Technology, Association for Computing Machinery, (Art N° 7).

DOI: 10.1145/3611314.3615907
Handle: http://hdl.handle.net/1942/41606

Progressive Network Streaming of Textured Meshes in the Binary glTF 2.0 Format

Wouter Lemoine
Hasselt University – tUL
Expertise centre for Digital Media
Diepenbeek, Belgium
wouter.lemoine@uhasselt.be

Maarten Wijnants
Hasselt University – tUL – Flanders Make
Expertise centre for Digital Media
Diepenbeek, Belgium
maarten.wijnants@uhasselt.be

(a) No objects loaded. SSIM: 0.13 **(b) Partial FOV loaded. SSIM: 0.85** **(c) Full FOV loaded. SSIM: 1**

Figure 1: Progressive loading of the Sponza scene.

ABSTRACT

The glTF 2.0 graphics format allows for the API-neutral representation of 3D scenes consisting of one or multiple textured meshes. It is currently adopted as one of two file formats for 3D asset interoperability by the Metaverse Standards Forum. glTF 2.0 has however not been designed to be streamable over the network; instead, glTF 2.0 files typically first need to be downloaded fully before their contents can be rendered locally. This can lead to high start-up delays which in turn can lead to user frustration. This paper therefore contributes a methodology and associated Web-based client, implemented in JavaScript on top of the three.js rendering engine, that allows to stream glTF 2.0 files from a content server to the consuming client up to the level of individual glTF bufferviews. This in turn facilitates the progressive client-side rendering of 3D scenes, meaning that scene rendering can already commence while the glTF file is still being downloaded. The proposed methodology is conceptually compliant with the HTTP Adaptive Streaming (HAS) paradigm that dominates the contemporary market of over-the-top video streaming. Experimental results show that our methodology is most beneficial when network throughput is limited (e.g., 20Mbps). In all, our work represents an important step towards making 3D content faster accessible to consuming (Web) clients, akin to the way platforms like YouTube have brought universal accessibility for video content.

CCS CONCEPTS

• **Information systems** → **Multimedia streaming**; • **Computer systems organization** → *Client-server architectures*; • **Networks** → *Application layer protocols*; *Network performance analysis*; *Public Internet*.

KEYWORDS

HTTP Adaptive Streaming (HAS), MPEG-DASH, three.js, Sponza, resource scheduling, Web browser, WebGL, Metaverse, glTF

ACM Reference Format:
Wouter Lemoine and Maarten Wijnants. 2023. Progressive Network Streaming of Textured Meshes in the Binary glTF 2.0 Format. In *The 28th International ACM Conference on 3D Web Technology (Web3D '23), October 9–11, 2023, San Sebastian, Spain.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3611314.3615907

1 INTRODUCTION

The metaverse, an Internet-connected collection of linked 3D worlds, is ever-increasing in size and scope. Each such 3D world is populated with distinct 3D objects, often in the form of textured meshes, that are located throughout 3D virtual space. The user can view such worlds on a 2D screen or a VR headset, by rendering the objects that are currently in the Field Of View (FOV). This FOV is dependent on the 6 Degrees Of Freedom (6DOF) the user has, which means it typically changes over time (e.g., due to user navigation in the virtual world).

Due to the sheer storage requirements this would impose, it is unrealistic to assume that clients will be able to locally store all the composing assets of all linked metaverse worlds. As such, solutions are needed to dynamically transmit 3D assets over the Internet from a content server to the consuming client. The prototypical approach to handle this, is to download all involved assets and to

wait for all those downloads to complete before starting the rendering process. However, depending on the file size of the 3D assets and the prevailing network conditions, this approach might entail a long start-up delay, which is undesirable from a user experience perspective and thus holds customer churn risks. A better solution therefore is to progressively load the 3D world and its composing assets, such that scene rendering can already start before the 3D world has been downloaded entirely. Ideally, such progressive loading is FOV-sensitive (i.e., it should prioritize the network delivery of those 3D assets that are currently visible to the user).

The progressive loading of 3D graphical worlds shows similarities to the way video content streaming is approached in the HTTP Adaptive Streaming (HAS) paradigm. With HAS, a video file is temporally segmented in small chunks (typically holding a few seconds worth of audiovisual content), with the client dynamically downloading the relevant chunks on an as-needed basis (e.g., due to playback progress or due to the user seeking in the video playback). HAS has been designed to be scalable as well as maximally compliant with existing Internet infrastructure. The scalability benefit derives from the fact that all streaming logic resides at client side, which allows the video content to be hosted on stateless vanilla HTTP servers. The Internet infrastructure compliance on the other hand allows to maximally capitalize on prevailing Web infrastructure such as Content Delivery Networks (CDN) and load balancers.

The primary contribution of this paper is a conceptual mapping of the HAS paradigm to the progressive network streaming of 3D worlds consisting of textured meshes described in the glTF 2.0 file format[1]. This mapping is not straightforward due to two essential differences between HAS video streaming and the progressive streaming of 3D graphical worlds. First, whereas video is a linear medium offering only 1 Degree Of Freedom during consumption, 3D graphical content allows 6DOF consumption (see also earlier); these additional degrees of freedom considerably complicate the client-side logic that must decide which content piece to fetch next from the server. Secondly, with video, once frames have been displayed, they can often be discarded, as the user will typically not re-visit them. For 3D meshes on the other hand, client-side asset caching will likely be beneficial since downloaded assets might be needed at multiple time instances (e.g., when previously downloaded assets re-appear in the FOV). Our secondary contribution is the validation of our proposed methodology via an open-source[2] Web implementation based on three.js[3]. Via an empiric, testbed-based assessment spanning a variety of emulated network conditions, we objectively evaluate our Proof-of-Concept (PoC) Web implementation by comparing its performance against the bulk download approach (i.e., where the glTF 2.0 scene is fully downloaded before being rendered).

2 RELATED WORK

This section will briefly review related work. For the sake of comprehensiveness, we will first discuss prior art on the non-HAS-like progressive network streaming of textured meshes described in graphics formats other than glTF 2.0. Then, we will zoom in on the HAS-like streaming of (non-glTF) textured meshes before concluding this section with glTF streaming prior art.

2.1 Progressive network streaming of meshes

The use of network streaming to achieve progressive loading of 3D meshes (both textured and non-textured) is not a new research topic; see, for example, the review paper by [Maamar et al. 2013] on this topic. Pioneering work was performed by Hoppe, who introduced the concept of *progressive meshes* (PM) that use edge collapsing and vertex splitting to respectively simplify and refine 3D mesh fidelity at run time [Hoppe 1996]. Numerous scientific approaches have adopted Hoppe's PM concept to achieve dynamic Level of Detail while streaming 3D models or environments. Three notable examples are a three-stage progressive transmission scheme that strives to minimize the distortion of the visible part of the scene [Liu et al. 2019], a greedy packetization heuristic that decides on the transmission order of individual vertex split operations such that intermediate decoded mesh quality is maximized [Cheng and Ooi 2008; Cheng et al. 2007], and a bit-allocation algorithm that aims to optimally distribute network bandwidth between the geometry and its mapped texture to maximize the quality of the model displayed on the client's screen [Tian and AlRegib 2004]. Over time, there also appeared progressive loading solutions for textured meshes that are not based on Hoppe's *progressive meshes* concept. For example, the progressive *P3DW* file format is optimized for fast client-side decompression while still achieving decent rate-distortion performance [Lavoué et al. 2013], while the *SRC* container format allows for interleaved transmission of respectively geometric and texture data while minimizing the necessary quantity of required HTTP requests [Limper et al. 2014].

The vast majority of the cited works focus exclusively on the progressive streaming of singular 3D models. In contrast, our work is concerned with the progressive loading of 3D scenes that are composed of multiple textured meshes. Our approach currently has no Level of Detail support; the progressive scene loading is achieved by dynamically deciding on the transmission order of the composing scene assets based on run-time heuristics (see Section 3.3).

2.2 HAS(-like) streaming of textured meshes

As stated in Section 1, HTTP Adaptive Streaming (HAS) has become the de facto paradigm for over-the-top network streaming of audiovisual content due to its ability to run-time adapt to prevailing networking conditions. HAS has been standardized by ISO/IEC in the form of MPEG-DASH [ISO/IEC 23009-1 2019; Sodagar 2011]. Zampoglou et al. were the first to appropriate MPEG-DASH for achieving the quality-adaptive network delivery of textured meshes, which were in their case specified in the open X3D standard [Zampoglou et al. 2018]. This seminal work was improved upon by the *DASH-3D* approach by allowing MPEG-DASH clients to selectively allocate network bitrate to respectively the geometry and textures of a 3D virtual world represented as a (textured) polygon soup in the OBJ graphics format [Forgione et al. 2018]. By differentiating geometry from textures in the client-side network streaming logic, it becomes feasible to make their bit allocation dependent on their relative contribution to perceptual scene quality. The *Relevance*

[1] Our choice for the glTF 2.0 format is driven by its recent adoption as a metaverse standard for 3D asset interoperability [Forum 2023].
[2] https://github.com/EDM-Research/progressive-gltf2

ABR technique extends the *DASH-3D* streaming logic by prioritizing the download of salient 3D scene assets that will likely attract most visual attention [Lievens et al. 2021]. Finally, very recently, an open-source MPEG-DASH-based Web framework for evaluating metrics and scheduling algorithms in the context of 3D content streaming has been contributed [Farrugia et al. 2023].

Unlike the just cited works, our approach is only *conceptually* compliant with HAS and does not explicitly use the HAS syntax and semantics under the hood. This is most notably exemplified by the fact that we do not adopt the HAS manifest format (e.g., MPEG-DASH MPD files). Instead, we directly work with the glTF 2.0 file format (see Section 3.2) without the need for an intermediate format or manifest file. This yields performance benefits by eliminating overhead. We do however mimic HAS' defining traits (e.g., use of HTTP for network transfer, streaming logic residing at client side, server-side scalability, firewall friendliness, etc). A second discriminator of our approach compared to the cited works is that the latter all use graphics formats other than glTF.

2.3 glTF-based 3D network streaming

Just like we do, the VIA (VIsibility-Aware Web-based VR) approach aims to mitigate the latency penalties that 3D graphics on the Web incur in naive bulk download scenarios [Slocum et al. 2021]. VIA does so by splitting the glTF 2.0 file into multiple (binary) files, where each file groups exactly those data buffers that are needed to render a single 3D object in the scene. Via an FOV-aware scoring mechanism, VIA determines what objects to prioritize and then fetches their dedicated data files first, this way decreasing the FOV rendering latency. VIA mostly works as an offline pre-processing step, yielding meta-data that must be stored at server side per individual scene and per individual FOV. In contrast, our approach works fully online and can therefore easily and directly adapt to FOV changes during 3D scene consumption. Three additional yet less relevant pieces of prior art on glTF network streaming are the *HyperVerse Transfer Protocol (HVTP)* that employs glTF 2.0 as a neutral scene graph to facilitate real-time communication between heterogeneous rendering engines [Dhanjan and Steed 2021], a glTF 1.0-based solution for transmitting BIM-like data to X3DOM clients [Scully et al. 2016], and a Cesium.js-based solution that exploits glTF 1.0 to stream 3D GIS data that has been spatially subdivided into 3D cuboids [Schilling et al. 2016].

3 PROOF-OF-CONCEPT

This section provides an overview of the PoC, focusing on how it achieves progressive scene loading and rendering using the binary glTF format.

3.1 Binary glTF

The binary glTF format is composed of three parts as shown in Figure 2. The first part, 12 bytes in size, consists of three uint32's specifying (in order), a magic value equal to the ASCII string 'glTF', a version identifier equal to '2' and a length equal to the total length of the file. These values describe the file as a binary glTF version 2 file of a certain length. Next is the *JSON-encoded meta-data chunk*, followed by the *binary data chunk*. Both these chunks have an eight byte sub-header, describing two uint32's specifying the length and type of the chunk, respectively. The type header field contains the ASCII representation of either 'JSON' or 'BIN' (to represent JSON meta-data and binary data, respectively). A chunk sub-header is always followed by the chunkdata (with a length equal to the number of bytes specified in the length field of the sub-header).

For the JSON chunk, the chunkdata encodes a set of lists that describes a virtual 3D world. These lists are structured in a tree-like fashion, as shown in Figure 3. Each list contains objects that have named values containing indexes pointing to objects of other lists. This index-based structure allows for re-use of, for example, a texture by two different objects.

The binary chunk contains the mesh and texture data necessary to render the scene. Because all the structure and meta-data is contained inside the JSON chunk, no structural information is included in the binary chunk, apart from the 8 byte sub-header.

In glTF, a discrete object inside the 3D world is called a *node*. Every node requires a *mesh*, which in turn requires mesh data and optional texture data. This mesh and texture data can be found in the binary chunk (or in external data, but this is not used in our implementation), in what is known as a *bufferview*. But, due to the structurelessness of the binary chunk, we require the JSON meta-data to point to where the information is stored, and how to interpret it. Now, because the JSON meta-data tells us what parts of the binary chunk are needed to render an object, we can exploit this to only download precisely those parts. This knowledge allows us to progressively download and draw selected objects instead of having to wait until the entire scene has been downloaded and then rendering it as a whole.

3.2 3D mesh streaming pipeline

Using what the binary glTF (glb) file provides, we enumerate below the details of our proposed pipeline to stream a 3D world that is described by a single binary glTF 2.0 file. The pipeline starts when the Web browser has fetched the necessary HTML and JavaScript files containing the PoC code and rendering engine code (i.e., three.js). For reference, Figure 4 gives a more visual indication of the flow.

(1) The client makes a partial HTTP GET-request to fetch the first 20 bytes of the involved glTF file, spanning the 12-byte file header and the eight-byte sub-header of the first chunk (containing the JSON data).
(2) The client uses the length value in the sub-header of the first chunk to fetch all the JSON data as well as the eight-byte sub-header of the next chunk (containing the binary data).
(3) The client allocates a buffer for storing the binary data based on the length value in the sub-header of the binary data chunk.
(4) The client parses the JSON data into an efficient data structure, calculates the bounding boxes of every node from the JSON data and sets up data structures to start fetching binary data, marking all nodes and bufferviews with the *To-Do* status. The client is now ready to fetch binary data.
(5) The list of nodes that have status *To-Do* are retrieved from the data structure and a filtering heuristic (see Section 3.3) is applied to this list. This decides which nodes will be retrieved from the server in this pass of the pipeline.
(6) Every node in the filtered list gets marked with the status *Doing*. A custom JSON chunk is constructed in memory by modifying

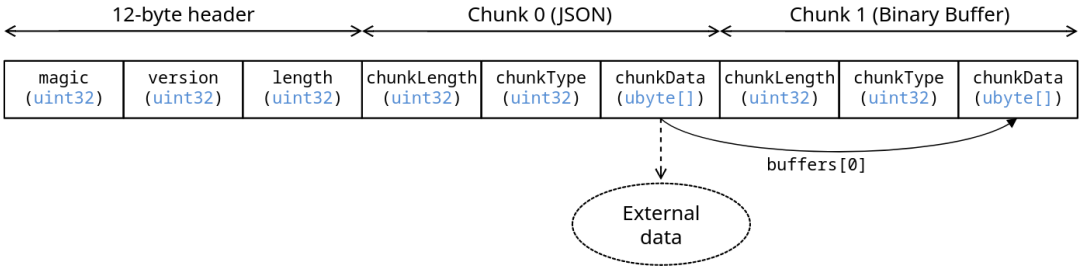

Figure 2: Binary glTF layout [Khronos Group 3D Formats Working Group 2021].

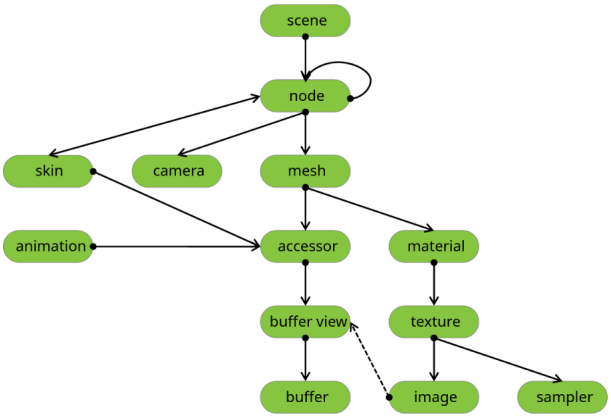

Figure 3: Relations between top-level arrays in glTF assets [Khronos Group 3D Formats Working Group 2021].

the scene description such that it lists only those nodes that are marked with the status *Doing*. If this modified JSON chunk (together with the required binary data) is fed to a rendering engine, only the listed subset of nodes will be added to the rendered scene.

(7) Using the data structure from step (4), the necessary parts of the binary buffer are calculated and put into an HTTP multipart byterange request. These bufferviews are marked with the status *Doing*.

(*) **The path branches here, steps 5-10 repeat until no more nodes have the status *To-Do*. The iterations of steps 5-10 continue asynchronously.**

(8) After the HTTP request from (7) returns, it is parsed and the received data is copied into their respective place described by their bufferviews in the buffer allocated in (3); the involved bufferviews are marked as *Done*.

(9) A binary glTF file is constructed in memory by concatenating a modified 12-byte header with the correct length, the modified JSON from (6), and the buffer allocated in (3). The resulting memory block contains precisely the JSON data as well as binary data needed to render the nodes that were selected in (5).

(10) The outcome of (9) is passed to the client-side render engine. After rendering, the nodes themselves are marked *Done*.

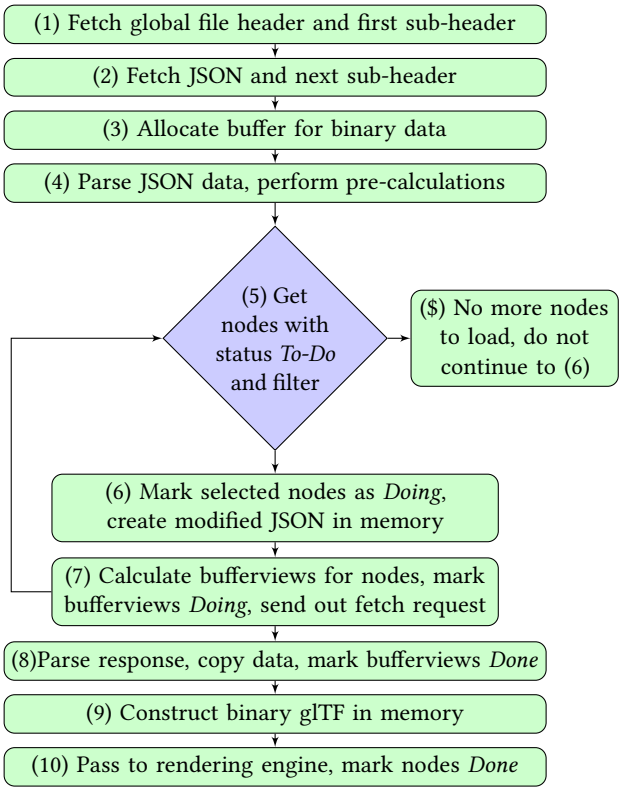

Figure 4: glTF-based 3D mesh streaming pipeline.

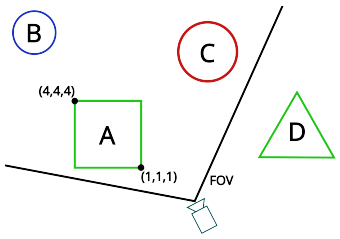

Figure 5: Top-down view of a 3D scene. Object A is a green cube, B a blue sphere, C a red sphere and D a green tetrahedron.

3.3 Node scheduling heuristics

Our proposed approach achieves progressive 3D scene loading by dynamically fetching individual assets (i.e., glTF nodes) as opposed to integral glTF files. This necessitates client-side heuristics to quickly calculate in which order nodes (and subsequently bufferviews) must be downloaded from the content server. It is apparent that these heuristics have a major impact on the user experience. For example, an anti-heuristic might prioritize the loading of nodes that fall outside the user's current FOV in the 3D scene, which would lead to a maximal time to interactive and hence a poor user experience.

In this work, we discern two types of heuristics: *ordering* and *rate limiting*. By using these two types of heuristic in tandem and tuning them properly, an optimized set of nodes to load can be found without much computational overhead.

An ordering heuristic sorts the set of not-yet-loaded nodes in descending order of priority. For example, not using an explicit ordering heuristic results in implicitly using the ordering in which nodes are listed in the original glTF file. This typically will not result in a pleasant experience for the user, since the ordering of the nodes in the glTF file is often quite random, resulting in nodes popping in at random places in the 3D scene. Using distance as an ordering heuristic does allow for a better experience. Users mostly interact with objects close by in the virtual world; therefore, it makes sense to install an inversely proportional relationship between an object's loading priority and its distance to the virtual camera. In effect, by calculating the object-to-camera distance and ordering the objects in terms of increasing distance, users will be able to more quickly interact with the objects that matter most. Other ordering heuristics could include the spatial surface of the node in 3D virtual space (cf. DASH-3D [Forgione et al. 2018]), or semantic importance as assigned by the author of the scene (cf. [Lievens et al. 2021]).

A rate limiting heuristic takes the list of not-yet-loaded nodes and filters out the nodes by means of a selection function. This type of heuristic is paramount in enabling progressive loading, as it only selects the next needed resources. An example of a relevant rate limiting heuristic is FOV-based filtering. This means all nodes that are not currently in view are removed from the list. Calculating whether a node is in view can be done using either the node's bounding box or its center-point (the former is more accurate but requires more computing power). A FOV-based heuristic can be tuned by changing the frustum angle to be either narrower or wider than the camera FOV. With a more narrow-angled frustum, less objects would be loaded, allowing for less bandwidth usage at the cost of only showing objects in the center of the view. In contrast, a wider-angled frustum would load more objects, slightly (pre-)loading objects that are just outside the view of the camera FOV; this of course leads to a higher bandwidth usage. In our PoC, the rate limiting frustum is chosen equal to the camera frustum to strike a good balance between bandwidth usage and number of objects loaded. Another example of a rate limiting heuristic is a simple node cardinality threshold. This heuristic will only keep the first N items in the list of (ordered) nodes. This heuristic can be tuned by using a dynamic value for N, depending on application characteristics or contextual factors like prevailing network conditions. In our PoC, this N value dynamism is achieved by introducing a second parameter M which act as a multiplier. In particular, after every download iteration, N is replaced by $N \times M$, allowing for more nodes to load in the next pass of the pipeline (e.g., for $N = 50$ and $M = 2$, the first pass would download 50 nodes, the second 100, the third 200, and so on).

To see how these heuristics interact with each other, consider the simple scene shown in Figure 5. Assume we use the following heuristics pipeline: FOV-based rate limiting, distance-based ordering, and object count rate limiting with $N = 2$ and $M = 1$. In a first pass, objects A-D (each corresponding to a separate glTF node) are passed to the FOV heuristic. Since object D falls outside the FOV, it is filtered out, leaving objects A, B and C. This list is then passed to the distance ordering heuristic, yielding the following prioritization: A, C, B. Finally, given that the maximum object count is two, B will be removed from the queue, leaving objects A and C to be queued for download. In the next pass, only B will be queued for download. In a third pass, no object would pass the heuristics (due to D falling outside the FOV); however, a FOV-based heuristic could disable itself once it detects it would return nothing. This would lead to D passing the heuristics and getting queued for download.

3.4 Practical example

We now provide a practical example of how the streaming pipeline functions according to the simple scene shown in Figure 5, using the same heuristics pipeline as described in Section 3.3 but this time using parameters $N = 3$ and $M = 1$. The objects in this scene are built from three simple meshes: a cube, a sphere and a pyramid. The textures are simple colors: green, blue and red. Assume each mesh and texture takes up 16kB of data in the binary chunk of the binary glTF file and is ordered in the file as mentioned (e.g., the cube is stored in bufferview 0, the color green in bufferview 3). Therefore, the total size of the binary chunk is 96kB. Also, assume the JSON file and the sum of all headers amount to 32kB in total (i.e., the data of the binary chunk starts at byte 32.000), yielding a complete file size of 128kB. Figure 5 also visualizes the virtual camera pose and associated FOV, which we assume to remain static for this example (the actual implementation supports a dynamic camera). It can be seen that only objects A-C are visible while object D is not in view.

The execution of our streaming pipeline proceeds as follows: initial steps (1) and (2) are performed as previously described, leading to HTTP fetch operations resulting in the retrieval of 20 bytes and 31.980 bytes, respectively. In step (3), a binary buffer with a size of 96 kB is allocated. Step (4) involves parsing the JSON data. For example, for object A, the bounding box ((1,1,1);(4,4,4)) is calculated, and a reference to the necessary bufferviews (i.e., 0 and 3) is cached as well. Similar calculations are performed for the other objects to determine their respective bounding boxes and necessary bufferviews. Moving to step (5), since no nodes have been loaded yet, all nodes are eligible for downloading. Following the heuristics described in Section 3.3, objects A, B and C are prioritized and queued for download. Continuing with step (6), a modified JSON chunk is generated in memory, containing a scene solely consisting of objects A, B, and C. In step (7), the necessary bufferviews are looked up in the metadata resulting from step (4); in this case, these are bufferviews 0, 1, 3, 4, and 5 corresponding to the cube and sphere meshes, and the colors green, blue, and red. Subsequently, a

multi-part byterange request is constructed, specifying the ranges 32.000-63.999 and 80.000-127.999. Once this request is scheduled, the next request can be determined. Returning to step (5), all objects within the FOV are already being processed; only object D remains, and is thus queued for download. Similarly, in step (6), another modified JSON chunk is generated, containing only object D. In step (7) then another request is created; considering that bufferview 4 is already being downloaded as part of the fetching of object B, only bufferview 2 (related to the pyramid and corresponding to byterange 64.000-79.999) needs to be retrieved. In the meantime, the completion of the first request is awaited. Upon completion, step (8) involves parsing the HTTP response and copying the bufferviews into their appropriate positions within the buffer generated in step (3). For instance, the first byterange (32.000-63.999) is copied to the corresponding byterange (0-31.999) in the buffer. In step (9), the binary glTF is constructed as described in Section 3.2. Finally, step (10) involves passing the binary glTF file to the rendering engine, which triggers the rendering of objects A, B, and C. The same steps (8-10) are repeated once the second network request has been fully processed, pertaining to the data necessary for rendering object D.

3.5 Browser and server considerations

As our PoC uses three.js as rendering framework, it relies on WebGL support in the browser to render the 3D meshes. No browser plugins are needed to make use of our PoC. On the server side no modifications are necessary, but support for HTTP multi-part byterange requests is required to be able to fetch parts of a file (as opposed to integral files). HTTP servers that do not support multi-part HTTP byterange requests will return the entire file, which would completely eliminate the progressive streaming benefit.

4 EXPERIMENTAL EVALUATION

This section goes into detail of how our PoC was experimentally evaluated, followed by a description of the ensuing results.

4.1 Experimental setup

Models. We tested our PoC with the following two 3D scenes:

- *forest*: a simple scene with flat color textures and 2.372 objects taken from https://www.kenney.nl/assets; total size: 23MB.
- *Sponza*: the well-known Sponza scene with detailed textures and 424 objects; total size: 43MB.[4]

These scenes were chosen as they allow us to evaluate our PoC with scenes that contain many objects with little textures and vice-versa. Both model geometries were compressed using Google Draco [Zhang et al. 2021].

Benchmark. To enable comparative evaluation of our approach, we utilize a benchmark (BM) test. This BM case is a bulk download of the entire scene (i.e., glTF file). This data is then passed in its entirety to the rendering engine, to be displayed in the Web browser.

Node scheduling heuristics. The following heuristics pipeline is used: FOV-based rate limiter, distance-based ordering, N-based object count rate limiting with multiplier M. This pipeline has been empirically chosen to have the best overall performance. However, we will also show the effect of using different values for parameters N and M. The values 50, 100, 150, 200 and 250 are chosen for N, while the values 1, 2 and 3 are chosen for M.

Apparatus. The experiments were run in a testbed consisting of two PCs that were connected via a gigabit Ethernet link (1Gbps) by means of a network switch. Those two PCs represented respectively the content server and the consuming client running a Web browser. The server PC was a Dell Latitude 5411 laptop with an Intel(R) Core(TM) i5-10400H CPU and 16GB RAM. The server ran a H2O HTTP server (version 2.3.0-DEV) on Debian 12. The client PC consisted of a Lenovo Legion desktop with an Intel(R) Core(TM) i7-9700 CPU, 16GB RAM and an Nvidia GeForce RTX2070 GPU. Software-wise, the client PC used the Debian 11 Operating System to run the Firefox Web browser via Playwright V1.27.1. Playwright is a Web browser automation framework, enabling scripting of common actions such as loading a webpage, key inputs and taking screenshots. The latter is necessary for the evaluation metric described below. The client-side rendering (i.e., inside the Web browser) was done using three.js; the rest of the client code is written in JavaScript (transpiled from TypeScript).

Network conditions. Using Linux's traffic control network emulator (`tc-netem`), we emulated various network conditions to test our PoC under. To provide a wide range of conditions, we use 20, 50 or 100 Megabits per second (Mbps) of throughput in combination with 10, 20 or 50 milliseconds (ms) of latency.

Metrics. To evaluate the performance, we measure the Structural Similarity Index Metric (SSIM) [Wang et al. 2004] over time. SSIM is a well-known predictor of perceived quality of digital media by comparing an image to a reference image. In our case, this image is a screenshot of the virtual camera view (which will increase in fidelity as more nodes are being loaded), while the reference image corresponds with the final resulting view (i.e., once the FOV has been fully loaded). By taking multiple screenshots during the streaming process, a plot can be generated showing how the SSIM evolves over time. The PoC client also reports the times when the first nodes, the full FOV and the complete scene have been downloaded and rendered.

Evaluation summary. By testing every combination of parameter, using 3 repetitions to rule out anomalies, a total of 864 tests are performed (2 scenes $\times$ 3 network throughputs $\times$ 3 network latencies $\times$ ((5 N values $\times$ 3 M values) + 1 BM) $\times$ 3 repetitions). The selection of scenes and network conditions gives a broad spectrum of possible operational conditions of this PoC. In both the PoC and BM, we keep the camera FOV static to simplify the result analysis.

4.2 Results

We now give the results of our experiments and explain what can be seen in the SSIM graphs. Figure 1 shows an example loading sequence of the *Sponza* scene. The associated SSIM of each image is mentioned in the caption to aid in comprehending the SSIM graphs. To further interpret an SSIM plot, the following notions should be taken into account:

[4] https://github.com/KhronosGroup/glTF-Sample-Models/tree/master/2.0/Sponza, edited so that each conceptual scene object is represented as a separate glTF node.

Table 1: Average loading times for *forest* (PoC vs BM) with $N = 150$, $M = 1$.

Network		PoC			Benchmark
Throughput (Mbps)	Latency (ms)	First load (ms)	Full FOV (ms)	Full file (ms)	Full file (ms)
20	10	3079	5999	11944	11673
	20	3229	6204	12243	11757
	50	4957	7058	13331	12315
50	10	1682	3070	5851	5458
	20	1872	3297	6134	5578
	50	2588	4232	7419	6221
100	10	1278	2264	3957	3421
	20	1412	2394	4176	3523
	50	2240	3384	5504	4231

Table 2: Average loading times for *Sponza* (PoC vs BM) with $N = 150$, $M = 1$.

Network		PoC			Benchmark
Throughput (Mbps)	Latency (ms)	First load (ms)	Full FOV (ms)	Full file (ms)	Full file (ms)
20	10	17594	17595	20598	20908
	20	17742	17743	20811	20988
	50	18389	18389	21508	21441
50	10	7973	7973	9497	9451
	20	8140	8140	9658	9351
	50	8869	8869	10418	10011
100	10	4798	4798	6221	5542
	20	4931	4931	6356	5652
	50	5742	5742	7218	6277

Table 3: Impact of parameter N on average loading time for the *forest* scene with $M = 1$, compared to benchmark (BM).

Network	N	First load	Full FOV	Full file
Poor (20Mbps, 50ms)	50	3278	7829	15540
	100	3615	7247	13950
	150	3869	7058	13331
	200	4071	6921	13171
	250	4217	6841	12975
	BM	/	/	12315
Good (100Mbps, 10ms)	50	1111	2737	5159
	100	1234	2403	4256
	150	1278	2264	3957
	200	1318	2185	3847
	250	1362	2129	3703
	BM	/	/	3421

Table 4: Impact of parameter M on average loading time for the *forest* scene with $N = 50$, compared to benchmark (BM).

Network	M	First load	Full FOV	Full file
Poor (20Mbps, 50ms)	1	3280	9471	20492
	2	3277	7077	13148
	3	3276	6939	12980
	BM	/	/	12315
Good (100Mbps, 10ms)	1	1113	3714	7806
	2	1127	2338	3944
	3	1093	2159	3728
	BM	/	/	3421

- A higher SSIM value means the input image (i.e., rendered scene) more closely resembles the reference (i.e. the fully rendered FOV)
- Every discrete "step" in the SSIM plot is a direct result of our PoC having downloaded and rendered an additional set of nodes; this hence shows that a scene is being loaded progressively
- The SSIM plots do not start at 0 due the skybox having an influence on the SSIM.

Note that slight variations in the test results across repetitions are due to Operating System background tasks and/or quirks in the three.js rendering engine. This sometimes leads to longer processing time and thus a longer time to render than the theoretical lower bound. Also, three.js sometimes fails to load the full FOV instantly but instead requires two render steps to show the complete FOV even though all encompassing objects have already been passed to the rendering engine. This is exemplified in Figure 8 (right), where the $N = 250$ lines for the *Sponza* scene do not directly jump to SSIM=1 but rather have an intermediate data point at SSIM=0.85, even though the full FOV only contains 137 objects.

Impact of throughput and latency. Figure 6 (*forest* scene) and Figure 7 (*Sponza* scene) show the SSIM over time for the various network conditions tested. Parameters $N = 150$ and $M = 1$ were chosen empirically as they provide average performance for both scenes. For the *forest* scene, our PoC substantially outperforms the BM under every network condition. Outperforming means in this case: having the fastest full FOV paint, thus the time where SSIM reaches one. For the *Sponza* scene, the PoC only outperforms the BM in the slowest of network conditions; in the other cases, the BM beats our PoC slightly. However, the PoC still manages to paint a part of the scene before the BM manages to have the full file downloaded and rendered (i.e., our POC achieves a faster time to interactive). Generally speaking (i.e., for both scenes), the performance gains of our PoC (relative to the BM) is inversely proportional to the quality of the network conditions.

It is important to stress that our PoC (in contrast to the BM) typically has not yet downloaded the entire glTF file at the moment when it achieves a full FOV. The timings for when the PoC has the full glTF file downloaded and rendered can be seen in Table 1 and Table 2. It is clear that, for most cases, the BM outperforms the PoC on this metric. This is due to the additional network round trips that our PoC incurs compared to one monolithic file download in the BM approach. However, for the *Sponza* scene at network conditions (20Mbps, 10ms) and (20Mbps, 20ms), the PoC slightly beats the BM. This is due to efficient interleaving of rendering and ongoing network requests. As networking is asynchronous in JavaScript, the browser can already render part of the scene while it is waiting on the next part of the data. As such, the computational cost of rendering the scene is spread out over time, which offsets the increased time needed to download the glTF file in multiple network round trips.

Impact of tuning parameter N. Figure 8 shows the impact of tuning the N parameter for the *forest* and *Sponza* scenes. Of the three repititions tested, we show the median result. For each scene, the lower N is, the faster the first paint occurs (i.e., the first jump in SSIM). This is an expected result, because a lower N means less nodes are loaded per fetch iteration (yielding lower download sizes). The drawback is that more requests are necessary (and hence more network round trips) to progressively download the file and thus render the FOV and full scene. Therefore, by tuning the N value, one can strike a good balance between a fast first paint, a fast full FOV and a fast full file. Concrete timings for these three events

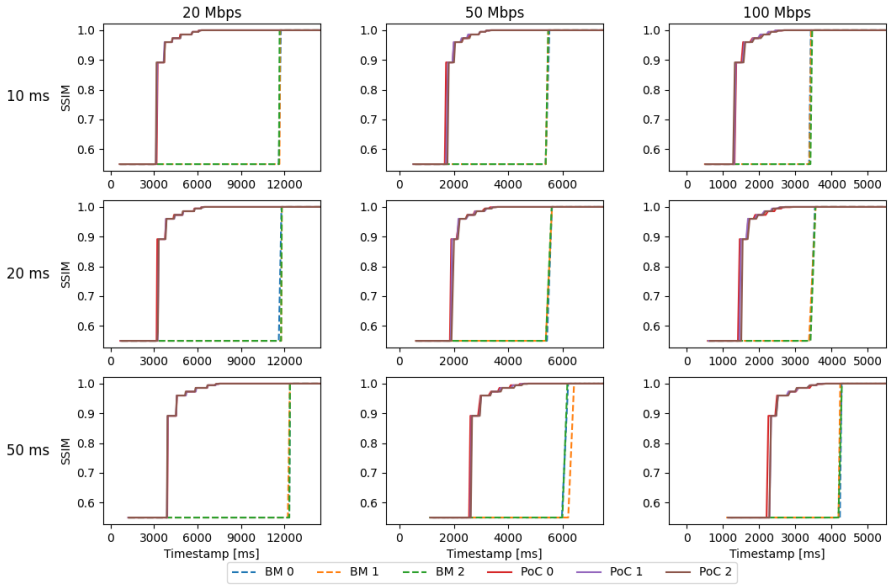

Figure 6: Proof-of-Concept (PoC) vs benchmark (BM) under different network conditions for the *forest* scene. Numbers 0-2 denote repetitions of the same test. Note that the x-axis is consistent in the columns only. Parameters: $N = 150$, $M = 1$.

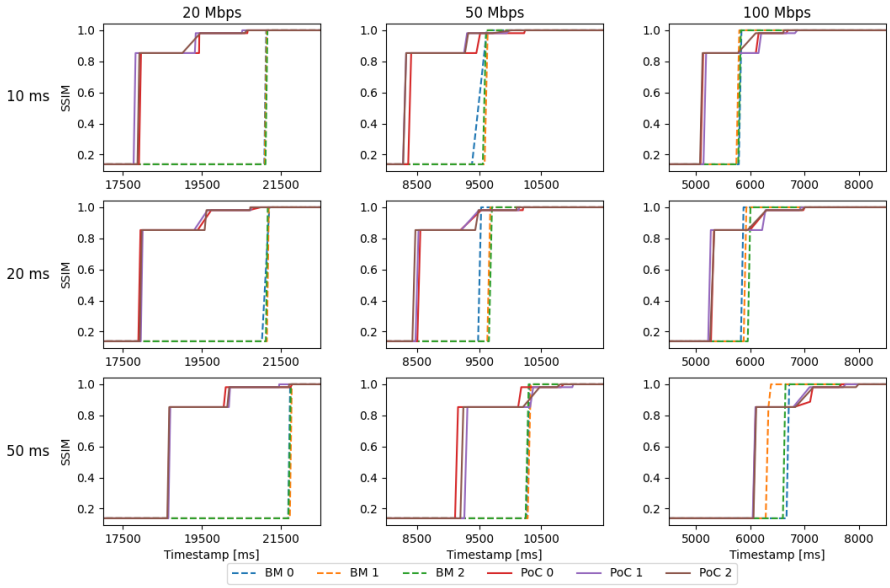

Figure 7: Proof-of-Concept (PoC) vs benchmark (BM) under different network conditions for the *Sponza* scene. Numbers 0-2 denote repetitions of the same test. Note that the x-axis is consistent in the columns only. Parameters: $N = 150$, $M = 1$.

can be seen in Table 3 for the *forest* scene (the table for the *Sponza* scene can be found in Appendix A). The table again emphasises how the first load time increases as N increases, whereas the times for a full FOV and full file decrease as N increases.

The network conditions shown in Figure 8 and in Table 3 are both extreme ends of the spectrum that we considered in our evaluation. This allows us to see how different N values behave under different network conditions. In both network situations, the overall shape of the plot is unaffected. The latency mostly increases the time between consecutive paints, whereas the throughput dictates the timeframe the paints happen in, while also affecting the time between consecutive paints.

Impact of tuning parameter M. In the previous paragraph, we showed that setting a low value for N allows for a fast first paint

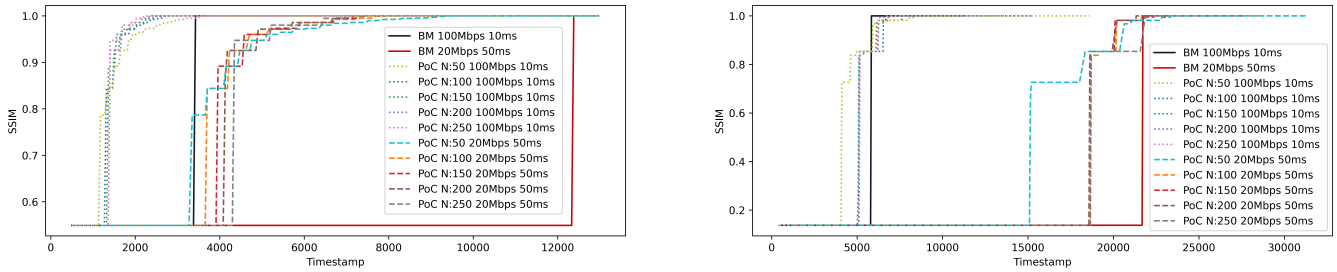

Figure 8: Impact of parameter N (with $M = 1$) for the Proof-of-Concept (PoC) vs benchmark (BM) under two extreme network conditions for the *forest* (left) and *Sponza* (right) scenes.

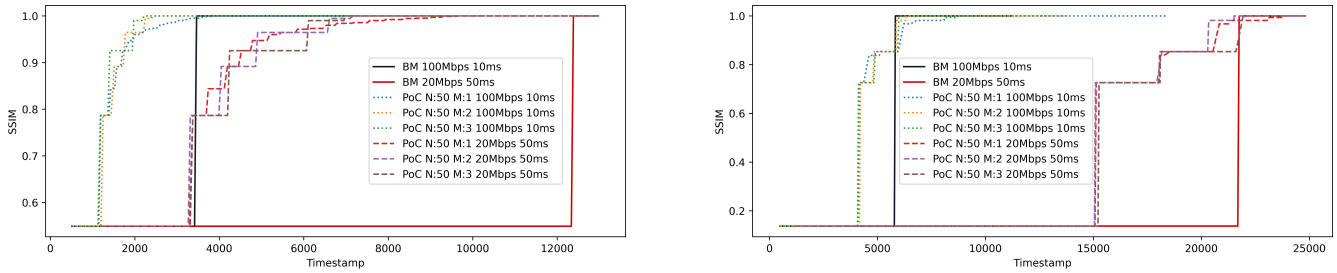

Figure 9: Impact of parameter M for the Proof-of-Concept (PoC) vs benchmark (BM) under two extreme network conditions for the *forest* (left) and *Sponza* (right) scenes with $N = 50$.

with the drawback of requiring a longer time to complete the full FOV and file. By tuning $M > 1$, we can have both a fast first paint time, and a fast full FOV and file time. Figure 9 shows this behavior for both scenes. For each network condition and scene, the first paint happens at nearly the same time. This is due to M not having an impact on the number of requested nodes in the first iteration (see Section 3.3). After the first paint, with $M > 1$, N does increase in subsequent iterations (i.e., $N \times M$). Subsequent paints will thus request more nodes, resulting in a higher step in SSIM value. Especially for the *forest* scene, where there are many nodes, the long tail can be eliminated by setting $M > 1$. Due to node instancing, only minimal extra data is required to render these numerous nodes from a limited set.

When comparing Table 3 and Table 4, we can see that, when $M = 3$ in the *forest* scene, the first load time of $N = 50$ can be attained, while only taking slightly longer than the full FOV and full file load times of $N = 250$. The *Sponza* scene also exhibits this behaviour, but it is less noticeable due it having less objects (the table for the *Sponza* scene can be found in Appendix A).

5 CONCLUSION AND FUTURE WORK

With the advent and proliferation of the metaverse, the need for 3D content delivery over the Internet increases. We have presented a set of heuristics that aid in efficiency calculating an order for loading nodes of a binary glTF 2.0 file. By combining these heuristics with HTTP multi-part byterange requests, we have devised a streaming pipeline that allows selective downloading and rendering of nodes within a glTF 2.0 encoded 3D world. Our implementation of this streaming pipeline is available as an open-source PoC client, built on top of the three.js library, and designed to run smoothly in any standard web browser. Our experimental results have shown that 3D worlds in the glTF 2.0 format are suitable for streaming using a HAS-like approach. Especially for networks with high latency and low bandwidth, such as mobile 4G networks, the PoC has shown itself to be the clear winner when comparing FOV paint times.

We propose three significant directions for future exploration. First, the networking code can be made rendering engine agnostic. This decoupling will allow for more flexibility when choosing an underlying render engine. This can be taken a step further, and the code can be rewritten in another language, to be compiled to WebAssembly for further performance improvements in the browser. Other targets could include Unity and Unreal Engine. Adapting the codebase for these platforms would facilitate seamless integration of our proposed 3D streaming pipeline. Second, the heuristics can be further improved upon. By introducing more refined heuristics, the manual tuning of the parameters N and M can be eliminated, or these parameters could be eliminated completely. Application-specific heuristics could also be devised and tested. Third, the networking code can include HTTP (re)-prioritization. By taking advantage of HTTP/2 and HTTP/3 enhancements in prioritization, network requests could be sent out in bulk without having to wait for a prior network request to return a response. Priority can be assigned and reassigned depending on more refined heuristics (i.e., up to the level of individual glTF nodes).

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union's Horizon Europe Programme under grant agreement 101070072, MAX-R (Mixed Augmented and eXtended Reality media pipeline).

REFERENCES

Wei Cheng and Wei Tsang Ooi. 2008. Receiver-Driven View-Dependent Streaming of Progressive Mesh. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video* (Braunschweig, Germany) *(NOSSDAV '08)*. Association for Computing Machinery, New York, NY, USA, 9–14. https://doi.org/10.1145/1496046.1496049

Wei Cheng, Wei Tsang Ooi, Sebastien Mondet, Romulus Grigoras, and Géraldine Morin. 2007. An Analytical Model for Progressive Mesh Streaming. In *Proceedings of the 15th ACM International Conference on Multimedia* (Augsburg, Germany) *(MM '07)*. Association for Computing Machinery, New York, NY, USA, 737–746. https://doi.org/10.1145/1291233.1291399

Jaspreet Singh Dhanjan and Anthony Steed. 2021. Revisiting the Scene-Graph-as-Bus Concept: Inter-networking Heterogeneous Applications Using glTF Fragments. In *Proceedings of the IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW '21)*. 342–346. https://doi.org/10.1109/VRW52623.2021.00068

Jean-Philippe Farrugia, Luc Billaud, and Guillaume Lavoué. 2023. Adaptive streaming of 3D content for web-based virtual reality: an open-source prototype including several metrics and strategies. In *Proceedings of ACM Multimedia Systems Conference (MMSys '23)*.

Thomas Forgione, Axel Carlier, Géraldine Morin, Wei Tsang Ooi, Vincent Charvillat, and Praveen Kumar Yadav. 2018. DASH for 3D Networked Virtual Environment. In *Proceedings of the 26th ACM International Conference on Multimedia* (Seoul, Republic of Korea) *(MM '18)*. Association for Computing Machinery, New York, NY, USA, 1910–1918. https://doi.org/10.1145/3240508.3240701

Metaverse Standards Forum. 2023. 3D Asset Interoperability using USD and glTF Domain Working Group Charter. Retrieved May 25, 2023 from https://portal.metaverse-standards.org/document/dl/5326

Hugues Hoppe. 1996. Progressive Meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. Association for Computing Machinery, New York, NY, USA, 99–108. https://doi.org/10.1145/237170.237216

ISO/IEC 23009-1. 2019. Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats.

Khronos Group 3D Formats Working Group. 2021. glTF 2.0 Specification. https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html. Version 2.0.1, 2021-10-11 23:01:57Z.

Guillaume Lavoué, Laurent Chevalier, and Florent Dupont. 2013. Streaming Compressed 3D Data on the Web Using JavaScript and WebGL. In *Proceedings of the 18th International Conference on 3D Web Technology* (San Sebastian, Spain) *(Web3D '13)*. Association for Computing Machinery, New York, NY, USA, 19–27. https://doi.org/10.1145/2466533.2466539

Hendrik Lievens, Maarten Wijnants, Mike Vandersanden, Peter Quax, and Wim Lamotte. 2021. Adaptive Web-Based VR Streaming of Multi-LoD 3D Scenes via Author-Provided Relevance Scores. In *Proceedings of the IEEE Conference on Virtual Reality and 3D User Interfaces (VR '21)*. 488–489. https://doi.org/10.1109/VRW52623.2021.00126

Max Limper, Maik Thöner, Johannes Behr, and Dieter W. Fellner. 2014. SRC - a Streamable Format for Generalized Web-Based 3D Data Transmission. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies* (Vancouver, British Columbia, Canada) *(Web3D '14)*. Association for Computing Machinery, 35–43. https://doi.org/10.1145/2628588.2628589

Yitong Liu, Jingfeng Guo, Ken Deng, and Yishi Liu. 2019. A Three-Stage Progressive Transmission Scheme for Virtual Environments Over Lossy Networks. *IEEE Access* 7 (2019), 184411–184422.

Haifa Raja Maamar, Azzedine Boukerche, and Emil Petriu. 2013. Streaming 3D meshes over thin mobile devices. *IEEE Wireless Communications* 20, 3 (2013), 136–142.

Arne Schilling, Jannes Bolling, and Claus Nagel. 2016. Using GlTF for Streaming CityGML 3D City Models. In *Proceedings of the 21st International Conference on Web3D Technology* (Anaheim, California) *(Web3D '16)*. Association for Computing Machinery, 109–116. https://doi.org/10.1145/2945292.2945312

Timothy Scully, Sebastian Friston, Carmen Fan, Jozef Doboš, and Anthony Steed. 2016. GlTF Streaming from 3D Repo to X3DOM. In *Proceedings of the 21st International Conference on Web3D Technology* (Anaheim, California) *(Web3D '16)*. Association for Computing Machinery, 7–15. https://doi.org/10.1145/2945292.2945297

Carter Slocum, Jingwen Huang, and Jiasi Chen. 2021. VIA: Visibility-Aware Web-Based Virtual Reality. In *Proceedings of the 26th International Conference on 3D Web Technology* (Pisa, Italy) *(Web3D '21)*. Association for Computing Machinery, Article 7, 9 pages. https://doi.org/10.1145/3485444.3487641

Iraj Sodagar. 2011. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE MultiMedia* 18, 4 (April 2011), 62–67. https://doi.org/10.1109/MMUL.2011.71

Dihong Tian and Ghassan AlRegib. 2004. FQM: A Fast Quality Measure for Efficient Transmission of Textured 3D Models. In *Proceedings of the 12th Annual ACM International Conference on Multimedia* (New York, NY, USA) *(MM '04)*. Association for Computing Machinery, New York, NY, USA, 684–691. https://doi.org/10.1145/1027527.1027684

Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612. https://doi.org/10.1109/TIP.2003.819861

Markos Zampoglou, Kostas Kapetanakis, Andreas Stamoulias, Athanasios G. Malamos, and Spyros Panagiotakis. 2018. Adaptive Streaming of Complex Web 3D Scenes based on the MPEG-DASH Standard. *Multimedia Tools and Applications* 77, 1 (01 Jan 2018), 125–148. https://doi.org/10.1007/s11042-016-4255-8

Fan Zhang, Ondrej Stava, Frank Galligan, Kai Ninomiya, and Patrick Cozzi. 2021. KHR draco mesh compression. https://github.com/KhronosGroup/glTF/blob/main/extensions/2.0/Khronos/KHR_draco_mesh_compression/README.md.

Table 5: Impact of parameter N on average loading time for the *Sponza* scene with $M = 1$.

Network	N	First load	Full FOV	Full file
	50	14938	17941	22023
Throughput:	100	18406	18407	21750
20 Mbps	150	18389	18389	21508
Latency:	200	18382	18382	21473
50ms	250	18403	18404	21790
	BM	/	/	21441
	50	3930	3930	6809
Throughput:	100	4801	4801	6793
100 Mbps	150	4798	4798	6221
Latency:	200	4783	4784	6209
10ms	250	4795	4796	6233
	BM	/	/	5542

Table 6: Impact of parameter M on average loading time for the *Sponza* scene with $N = 50$.

Network	M	First load	Full FOV	Full file
Throughput:	1	14943	17969	23090
20 Mbps	2	14876	17935	21275
Latency:	3	14995	17918	21706
50ms	BM	/	/	21441
Throughput:	1	3921	3921	8364
100 Mbps	2	3945	3945	6238
Latency:	3	3924	3925	5825
10ms	BM	/	/	5542

A SPONZA TABLES

This section shows Table 5 and Table 6 accompanying Table 3 and Table 4, respectively. They are shown for completeness sake.