glTF Streaming from 3D Repo to X3DOM

Timothy Scully[1], Sebastian Friston[2], Carmen Fan[1], Jozef Doboš[1] and Anthony Steed[2]

[1]3D Repo Ltd, London, UK
[2]University College London, UK

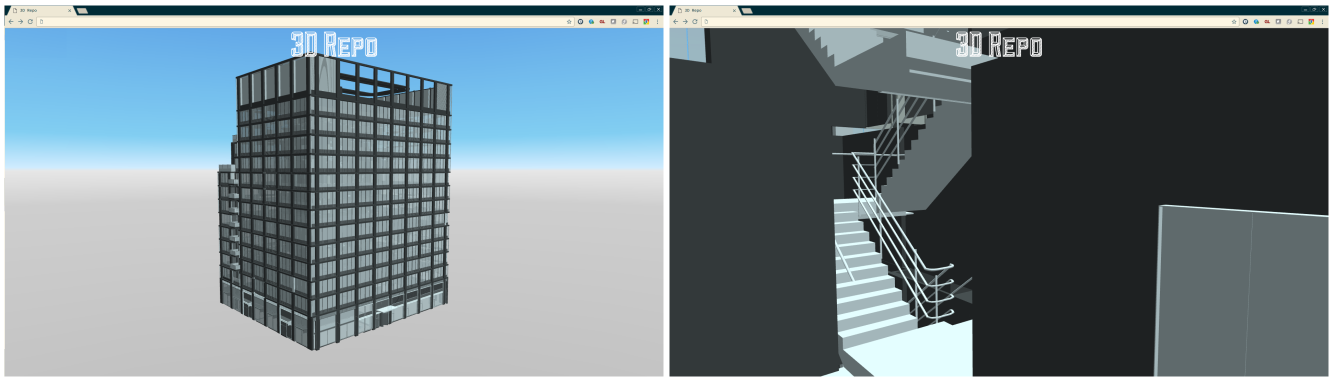


Figure 1: *An example of a 3D model shown with visibility culling in a web browser reducing the immediate GPU memory consumption. The view from outside (left) consumes 693 MB, whereas the view from the inside (right) with visibility culling consumes only 161 MB of the GPU memory. This is a part of a much larger scene shown in Fig. 8. Model courtesy of Canary Wharf Contractors Limited.*

Abstract

As Web3D technology advances, the need for delivering real-time 3D content online has gained traction in the academic as well as commercial world. Various efforts have been made in creating a suitable transmission format for streaming of 3D assets over the Internet. Despite being accustomed to waiting for long periods of time for massive scenes to load in CAD editors, end users often expect an instant rendering on a web browser. An effective streaming transmission format, coupled with progressive encoding methods, is able to create a better interactive experience for the users. Most of the existing techniques are either domain specific, tying the users in on a particular rendering engine, or they are too general; resulting in extra processing at the application level. In this paper, we demonstrate a novel method of transmitting 3D assets in glTF format for high interoperability and scalability in 3D Repo. Firstly, we extend glTF with the ability to stream binary data buffers with a progressive encoding technique to increase performance and overall client interactivity. Next, we extend X3DOM for glTF support and introduce multipart optimization into glTF as a way of grouping multiple meshes together which significantly reduces the number of network requests as well as draw calls. Finally, we investigate memory management protocols and devise a novel GraphicsMemoryManager suitable for streaming on top of X3DOM in order to render models that otherwise would not fit available video memory.

Keywords: streaming, multipart, glTF, 3D Repo, REST

Concepts: •**Information systems** → *RESTful web services;* •**Computing methodologies** → *Graphics file formats;*

Web3D '16,, July 22 - 24, 2016, Anaheim , CA, USA
ISBN: 978-1-4503-4428-9/16/07
DOI: http://dx.doi.org/10.1145/2945292.2945297

1 Introduction

There is an increasing demand for rendering of massive 3D models online. This is especially true in the architecture, engineering and construction (AEC) sector where demand increased by the introduction of the government mandated Building Information Modelling (BIM). The mandate describes a shared data model which collates all information for a construction project with the aim of increased collaboration and efficiency at both design and construction time. Due to the ubiquitous nature of cloud based platforms, it forms the perfect candidate to base such a system on. However, a working solution needs to be developed to deal with large models like those represented in Figs. 2 and 8. An exemplar such as this can be federations of several detailed sub-models and consist of millions of components, and with a large number of projects of comparable size there is a need to develop the technology further.

There are several challenges associated with rendering large complex models on the web, some of which are shared with desktop based systems. The first is overcoming memory limitations of small mobile devices and those imposed by browsers. The second is rendering a large number of disparate meshes resulting in a large number of network requests and draw calls. However, these two challenges are juxtaposed with each other; the former seeks to render only partial models on a granular level, whereas the latter looks to render everything at once. In this paper, we develop a streaming system that attempts to address both challenges. We conclude by looking at the performance of the system under different conditions.

Several transmission formats have been created which are suitable for web-based streaming, e.g. [Limper et al. 2014; Sutter et al. 2014]. However, many of these formats are tied to a specific renderer reducing the opportunity for interoperability, see §2 for details. Progressive encoding techniques such as [Limper et al. 2013a] have endeavoured to reduce bandwidth requirements as well as allow for lower level of detail representations to be rendered before all data is transmitted. This results in higher interactivity, but does not necessarily reduce the memory footprint.

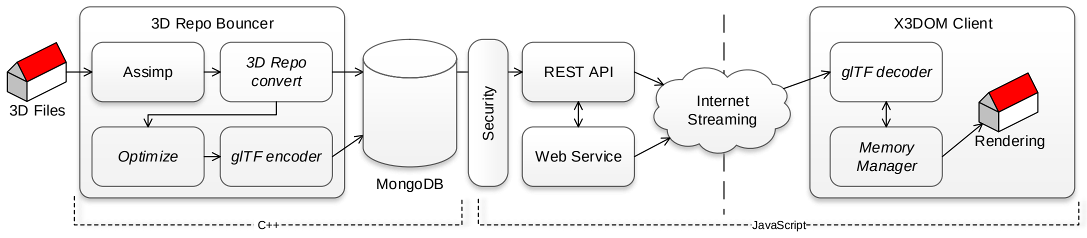


Figure 2: *High-level processing pipeline overview. 3D files are firstly processed by Open Asset Import Library (Assimp), then converted to 3D Repo internal VCS format and stored in the DB. Next, they are optimized and converted into glTF for DB "stashing". The data is then streamed through REST API to the X3DOM client for rendering. All processing-intensive parsing and optimization is done in C++ while data serving and client-side visualization is implemented in Javascript using a combination of NodeJS, AngularJS and X3DOM.*

In this paper, we present a solution based on the open source revision control system 3drepo.io [Scully et al. 2015]. We adapt it to face the aforementioned challenges by integrating a streaming architecture. This works by reconstituting smaller disparate meshes into a large supermesh that can be rendered in a single draw call. Such a reconstitution process happens in parallel to the rendering thread in order to avoid degrading overall system interactivity. The fixed nature of the revisions means that we do not deal with dynamic or otherwise animated models. In addition, the particular use case of inspecting construction models means that reconstitution needs only to happen in reasonably interactive but not necessarily real time. We drive the selection for reconstitution based on the current viewing frustum and a mesh prioritization function. Finally, we integrate support for POP Buffer [2013a] encoding to allow progressive visualization in the future. The culmination of this is a system that can provide the user with interactive model inspection facilitating the rendering of much larger models than before.

Contributions. Our main contributions are as follows:

1. glTF extension of X3DOM
2. Multipart extension of glTF
3. Geometry streaming using glTF
4. POP Buffers extension of multipart extension of glTF
5. GraphicsMemoryManager for out-of-order rendering
6. Open source implementation of 1. to 5. available on GitHub

2 Related Work

Web3D rendering. Recent advances such as 3drepo.io [Scully et al. 2015] building on top of 3D Repo [Doboš et al. 2013] and webVis/Instant3DHub [Behr et al. 2015] on top of [Jung et al. 2012] began to provide support for commercial web-based 3D rendering. An extensive review of other Web3D trends can be found in Mouton et al. [2011]. Historically, the two main streams of Web3D represent both imperative [Dirksen 2013] as well as declarative [Behr et al. 2009; Sons et al. 2010] approach to defining 3D on the web. This paper is a direct extension of 3drepo.io [Scully et al. 2015] based on declarative X3DOM [2009] adding support for new data format, streaming and memory management.

Data formats. Due to the popularization of WebGL [Khronos Group 2014], many efforts have been made on designing a new file format suitable for transmission of 3D geometry over the Internet. Apart from the early days of embedding geometric data directly into HTML—which proved unsuitable for larger models—one of the attempts was using images and binary geometry in [Behr et al. 2012]. These, together with other formats based on JSON, BSON and OpenCTM [Geelnard 2009] were contrasted in [Doboš et al. 2013], with a similar study being concurrently performed by Limper et al. in [2013b]. Such formats were later superseded by the *Shape Resource Container (SRC)* [Limper et al. 2014] where rendering buffer is specified as multiple buffer chunks allowing data to be progressively transmitted over a network. To cope with large models online, they further introduced bounding interval hierarchies (BIH) which are spatial data structures calculated on the client to accelerate visibility culling [Stein et al. 2014]. Similarly, [Sutter et al. 2014] defined a format called Blast; a generic container dedicated to efficient streaming of binary data in self-contained chunks. However, Blast is a typeless container that can support any kind of data on a key-value pair basis. Hence, the client has to decode and interpret the data at the application level, making it less interoperable despite arguably being more flexible.

What was missing, however, is a single ratified format for the web. Although COLLADA [Arnaud and Barnes 2006] has long been the de-facto standard for asset exchange between authoring tools, it is unsuitable for web-based delivery and fast rendering. Therefore, the *GL Transmission Format (glTF)* [Khronos Group 2016] was devised taking inspiration from previous work such as SRC. A scene within glTF is described in a single JSON header file, with buffer data, shaders and textures divided into separate binary files. However, the nature of the specification does not support any form of progressive transmission of binary data. Very recently, work has been done by [Cozzi et al. 2016] creating a variation of glTF, known as KHR binary glTF. Similar to SRC, the binary data is concatenated with a JSON header describing the format, length of the JSON scene structure and length of the entire file. The extension `KHR_binary_glTF` is introduced to allow for creation of buffer views into the appended binary data, embedding binary data within the glTF to save network requests without the need to convert into Base64-encoding. Nonetheless, given the current glTF specification, our implementation is far less invasive, see §6.

Progressive visualization. A bottleneck for any web-based rendering is the transmission of large binary data blocks in order to visualize the scene. Such bandwidth requirements can be reduced by compressing and decompressing meshes. [Lavoué et al. 2013] suggested the use of a progressive compression algorithm described in [Lee et al. 2012] to visualise the meshes in a lower-level of detail during a progressive decompression operation, increasing the quality of the visualization as the operation executes. [Limper et al. 2013a] took it one step further by using a simple quantization scheme to avoid the need for decoding, thus diminishing the amount of processing required on the client. In addition, primitive data was reordered into nested levels of detail, allowing the buffer to be transferred progressively and perform rendering during data reception, minimizing the impact of network latency on the user.

3 System Overview

The basic 3drepo.io architecture consists of two subsystems shown in Fig. 3. The first is the C++ backend that performs heavy processing and optimization to prepare the models for web rendering. The second is the Javascript web server and frontend that serves and renders data for the end user. Both require changes so that streaming can occur. In this section, we provide an overview of the overall architecture leading to a discussion of the adaptations in later sections of this paper.

The C++ backend is based on 3D Repo [Doboš and Steed 2012] which is an open source version control system (VCS). There, 3D models and associated metadata are stored in decomposed Binary JSON (BSON) format, see [Doboš 2015]. To facilitate the storage of BSONs the system uses a NoSQL database MongoDB. *3D Repo Bouncer* is a C++ library which was originally designed to protect the database by processing and rejecting potentially malicious requests as well as by verifying the identity of the users. Since then, it evolved into a *Compute Node*-style processing library which takes as input 3D models and converts them into an internal 3D Repo format for version control purposes. In addition, it performs several layers of optimizations in order to reduce the complexity of the models for web-based rendering while preserving their fidelity. In contrast, the web client consists of an AngularJS user interface combined with a Javascript-driven X3DOM renderer which is served information by a NodeJS REST API that sends compressed data from the database to the client.

A core change over the original 3drepo.io system is the introduction of multipart nodes [Behr and Sturm 2015]. These allow the number of web requests and draw calls to be reduced by batching meshes together into several supermeshes. Individual attributes, such as color properties and visibility, are stored in textures and accessed on a submesh basis via a shader. This process is suitable for static geometry only and requires pre-transforming of all vertex data into a common coordinate system. Due to the limitation of WebGL which can only handle 16-bit index buffers, the supermesh can contain maximally $2^{16} \doteq 65$k indices. At this point, any submeshes greater than this limit are split. The finalized scene graph then consists of a single root transformation with the multipart meshes as its children. Each supermesh further retains a mapping of the original submeshes to the merged vertex data. This allows the submeshes to be individually selected using picking buffers for connection to engineering metadata. This processing happens in the C++ backend before being stored in DB ready for the web server to stream it.

In order to then proceed to a streaming architecture, further changes have to be made. Firstly, to increase interoperability and provide a standardized container for the streaming functionality we implement glTF in both the backend and frontend. We base the streaming on the user's current visibility so that visible submeshes are streamed, and submeshes outside the user's vision are culled, all determined client-side in real time. This is in conflict with the introduction of multipart where submeshes are combined into supermeshes. To integrate both approaches we introduce a new `GraphicsMemoryManager` on the client. This manager takes the streamed data from the server and reconstructs dynamically multipart meshes on the client. The reconstruction takes place in parallel to the rendering pipeline to avoid reducing interactivity for the user.

To facilitate streaming from the database we utilize MongoDB's GridFS technology. Each glTF document generated by the C++ backend is stored in the database as a JSON header and a set of binary blobs. These blobs are stored in the GridFS system as files. The document metadata such as MIME type, file size, file name, etc. are stored in a `.files` postfixed collection. The document data is stored in a corresponding `.chunks` collection in a series of 255KB documents. By storing in this manner, the collection becomes a random access storage medium for use in web streaming. This allows the client to request byte streams of data from the database with information given to it in the glTF header.

4 glTF Implementation

We improve upon the original 3drepo.io system [Scully et al. 2015] by exporting models from the system in glTF format [Khronos Group 2016]. In order to provide support for this encoding, we introduce modifications and overall improvements to both the server in §4.1 as well as the client in §4.2.

4.1 Server

The glTF specification has recently been finalized and is developed by the Khronos Group to become the de-facto web transmission standard. This file format consists of two parts; i) a binary blob which contains the mesh data, and ii) a JSON header which describes its format. The move from SRC to glTF marks a departure from an X3DOM-specific format to a more interoperable representation for 3D Repo. Fortunately, there are strong similarities between the original SRC implementation [Scully et al. 2015] and the new glTF specification [Khronos Group 2016] making the transition relatively straightforward.

In SRC, meshes are defined with `accessors` to `position`, `normal` and `index` offsets into a particular `bufferView`. This in turn describes `bufferChunks`, i.e. slices of the specific vertex attribute or index buffers. glTF operates in a very similar way, the most significant difference being a `bufferView` that declares an offset and byte size window frame into a `buffer`. However, unlike in SRC, the glTF header is a separate file to its binary data. The `buffer` thus specifies a `uri` pointing to the location containing the data. Such a design supports multiple binary files in a single glTF. In addition, it encapsulates the scene graph structure which therefore does not need to be described within X3D any more. Thus, a single scene can be represented in one glTF header what increases the interoperability significantly.

Once the model is exported in glTF, it is "stashed", i.e. stored in the database ready to be served over the Internet. This takes advantage of the fact that 3D Repo is a VCS where once a 3D model revision has been committed, we can be sure it will not be changed any more. New changes would trigger a revision commit, at which point the system would generate a new "stash". To take advantage of this situation, an X3D representation of each revision is automatically generated at the point of importing the model by the C++ backend. This relieves the extra operations required on the REST API service, allowing it to concentrate on serving requests rather than transcoding data dynamically as done in previous iterations of the system, c.f. [Doboš et al. 2013; Scully et al. 2015].

4.2 Client

To add glTF support to the client we create a custom fork of the X3DOM library on which it is based. The library already contains simple geometry containers such as SRC or BinaryGeometry [Behr et al. 2012] but unlike these, glTF contains a full scene description, as well as geometry and material definitions. Consequently, we made two significant modifications to X3DOM, adding support for two new elements: `glTFNode` and `glTFGeometryNode`.

glTFNode. The new `glTFNode` element is an API for specifying a glTF file. When a glTF node is imported, the scene graph is parsed and used to generate a compliant DOM. This is then passed to X3DOM where it is processed and added to the current scene graph using the same mechanism as the existing `<inline>` element. Such an approach allows glTF retrofitting to be minimally invasive, reducing the difficulty of merging future upstream changes. Furthermore, it maximises the reuse of existing X3DOM functionality. In addition, because the DOM is built using the published X3DOM API, it is trivial to maintain. Where any deviations from the original API occur, they are for simplicity and performance reasons. For instance, the DOM is created in memory rather than actually rendering XML, which allows the passing of objects from the `glTFNode` instance to child elements directly within the graph.

glTFGeometryNode. After parsing the glTFHeader, the `glTFNode` creates a number of `glTFGeometryNode` elements representing supermeshes in the scene. Each supermesh consists of multiple submeshes which share a common set of shaders, shader parameters and attribute binding parameters—as required for all submeshes to be drawn in a single call. The submesh attribute data is contained in `bufferViews`, and depending on the interleaving used may be spread across one or more views. Although it is primarily used by the `glTFNode`, it can be invoked via user supplied XML just as well. Within the `bufferViews`, each submesh attribute data is represented by segments. These correspond to a contiguous block of data within the buffer and are defined by glTF accessors. Such segments can be added or removed from the GPU memory as single units. This is the highest resolution view of the attribute data, although manipulating data on the individual accessor/attribute level would involve byte-level operations and, thus, defeat the point of using glTF. Since attribute data can be interleaved in a number of different ways, a constraint is required to make it contiguous, e.g. the index data, because that way it can be moved in a single block. If it were not, it would require byte-level operations to arrange the attribute data which might be prohibitively expensive in terms of CPU processing.

4.2.1 Shader Function Support

One complication, however, is that glTF uses semantics to identify shader parameter bindings, whereas X3DOM uses shader variable names by convention. On a typical platform, it would be trivial to convert the semantics when binding constants, but for X3DOM, it would require modifying the shader source—possible but suboptimal in Javascript. Instead, we make the following conditions:

1. All shader variable names shall be X3DOM compliant, i.e. they will match the names provided by the X3DOM documentation. All X3DOM specified variables will be supported, not just the subset defined by glTF.
2. glTF semantics shall be ignored by X3DOM. Because there is no overlap in the mechanisms of names and semantics, by convention all glTF semantics shall be ignored by X3DOM, and shaders must be X3DOM compliant. The glTF will still specify parameter semantics for third-party viewers, but only the parameter names will be of any consequence in X3DOM.

Comparing the glTF specification to the X3DOM documentation, we can see that the only time there is a conflict is for the inverse transpose matrices, for which X3DOM expect 4x4 while glTF expects a 3x3 representation. To support this case we extend the X3DOM matrix4x4 prototype. When a shader parameter is set, the X3DOM type is converted to a native type with the `toGL()` method. We add an equivalent method, but that takes a target type to convert to, and set X3DOM to use this whenever the normal matrix is set. Since the target type is ascertained from the X3DOM shader object,

```
"mesh_0": {
  "primitives": [{
    "material": "material_0",
      "mode" : 4,
      "indices" : "acc_mesh_0_m0_f",
      "attributes" : {
      "NORMAL": "acc_mesh_0_m0_n",
      "POSITION" : "acc_mesh_0_m0_p",
      "IDMAP" : "acc_mesh_0_m0_id",
      "TEXCOORD_0" : "acc_mesh_0_m0_uv_0"
    },
    "extras": {"refID": "smesh_0"}
  },
  { /* another submesh entry */ }
}
```

Listing 1: *A sample declaration of a multipart mesh. This model has a single multipart mesh with two submeshes.*

which itself determines the type from the shader source, this approach does not conflict with condition 2. above. The result is that a single shader can now work with both X3DOM and any third-party viewer with no further manipulation on the client and with no additional data having to be provided by the exporter as desired.

4.2.2 Multipart Support

To utilise the existing multipart functionality of X3DOM, the glTF importer places the Shape element containing the `<glTFGeometry>` element inside a `<multipart>` element. The importer generates an *idMap* object from the data within glTF and sends this object, along with a reference to the Shape element, inside the DOM. For simplicity, the `<multipart>` node has been modified to detect this case and use these objects directly, rather than trying to retrieve them from the url attributes. An example declaration is shown in Lst. 1. The `<multipart>` implementation material model assumes the usage of the X3DOM Material API, and supports only these parameters. For convenience and performance, the server includes along with each material an equivalent X3DOM material expressed as a JSON object, which is incorporated directly into the procedurally generated *idMap*. This is implemented in X3DOM by using a shader with a specific set of capabilities, akin to a fixed function pipeline. The only material parameters supported are those defined by an X3D material. We discuss the limitations of this in §5 below.

5 Streaming

The overall requirements of the streaming architecture depicted in Fig. 4 are two-fold. The first is to reduce the amount of memory that the client uses to allow loading of large models. The second is to allow models to be progressively loaded to give the user an increase in response time. Both these requirements can be satisfied by the same mechanism based on the current viewing frustum. To reduce the memory usage we offload submeshes that are not visible to the client. We then add back in submeshes that are visible based on their visual importance to the user using metrics such as size and distance. The introduction of multipart into the system precludes this from being implemented using the standard X3DOM culling framework. The large multipart supermeshes have no concept of culling the submeshes which constitute them and so if only a small part is visible then the whole supermesh is still held in GPU.

We therefore extend the culling framework of X3DOM such that as the camera moves through the model, the set of visible submeshes is computed and a new set of rendering buffers is dynamically calculated. To support this we make modifications to both the server and the client. On the server, we compute spatial partitioning of

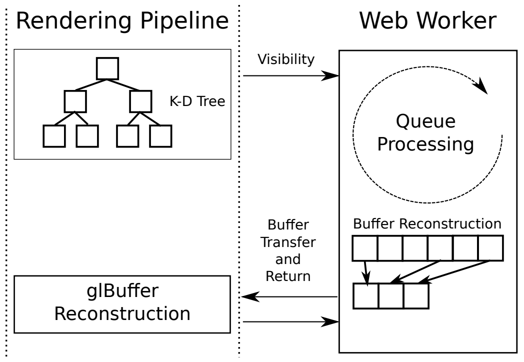


Figure 3: *An overview of the streaming architecture illustrating its multi-threaded nature and the transfer of memory between the Web Worker and the rendering pipeline.*

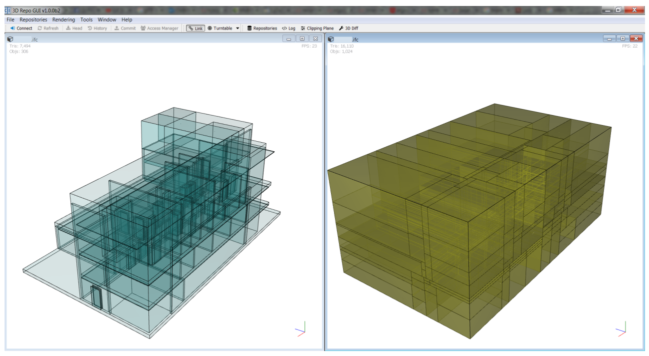


Figure 4: *Visualisation of bounding boxes (left) vs kd-tree spatial partitioning with depth of eight (right) in 3D Repo GUI.*

the model using kd-trees. On the client, we then use this kd-tree to dynamically stream data to the GPU. We traverse the kd-tree to compute the visibility of submeshes as part of the standard rendering pipeline. A list of visible meshes is then used to construct a new set of rendering buffers on-the-fly. To avoid decreasing the interactivity of the renderer these buffers are constructed in a separate set of threads as shown in Fig. 4.

5.1 Server Implementation

The server implementation consists of pre-computing spatial partitioning which is then passed on to the client. Once there, it is used to control when and what parts of the model are requested and streamed from the server to the client.

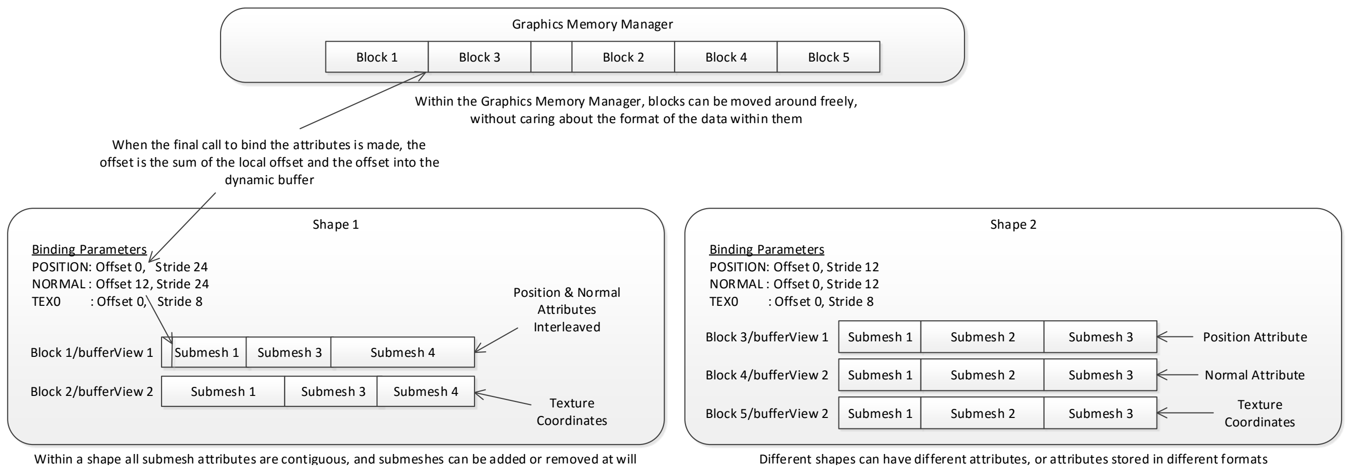
Spatial partitioning. The use of multipart models means that the traditional bounding box visibility tests of X3DOM are no longer applicable. The multipart meshes represent large parts of the scene, with possibly little or no spatial coherence. This means that even looking at small part of the scene negates culling of the other parts. In order to restore this feature on a submesh level, the information about bounding boxes needs to be added to X3DOM separately. The algorithm used here, unlike in [Stein et al. 2014], is based on kd-tree [Bentley 1975] and adapted for bounding boxes. It starts with the bounding box of the entire scene, and for each iteration, the bounding box is partitioned at the median value of the midpoint of all submeshes that currently reside within the section. The inner nodes of the kd-tree represent the partitioning (axis, median value), with the leaf nodes storing the unique ID (UID) of the submeshes that reside within. We use the median so that the partitioning is balanced between the number of meshes on the right and left subbranches. Fig. 5 demonstrates a visual comparison between the bounding boxes of submeshes against the kd-tree partitioned space. Although the kd-tree is costly to construct in comparison to other algorithms such as Bounding Volume Hierarchy and Bounding Interval Hierarchy [Zachmann 2002; Nam and Sussman 2004], they are fast to traverse for the client [Wald et al. 2007]. Although certain spatial partitioning structures can be computed client-side, c.f. [Stein et al. 2014], the static nature of the version controlled models means we need to prepare the kd-tree only once.

5.2 Client Implementation

The client implementation must be adapted to perform several operations. The first is to compute visibility of submeshes, and the second is to use the list of visible submeshes to construct the dynamic buffers for rendering.

Submesh visibility. To compute the submesh visibility, we integrate with the standard X3DOM visibility culling algorithm. When the algorithm reaches a glTFNode in the scene graph, we perform an additional set of steps to compute the visibility of its submeshes. In the first step, the kd-tree is traversed to identify partitions of space that are not visible. As it passes down the tree, the corners of the viewing frustum are tested against the median value for the associated axis. For an axis-aligned frustum, it simply needs to test on which side the maximum and minimum corner values for that axis lie. If they lie either side of the median line, both sides are visible, and if both are on one side, then only that side is visible. To ensure that the viewing frustum is always axis-aligned it is transformed by the inverse world transformation of the glTFNode. After traversing the tree, we end at a leaf node containing a set of *possibly* visible submeshes. For each one of these, its bounding box is tested against the viewing frustum. If the bounding box is fully or partially inside the frustum, it is visible, otherwise not. By doing so, the small feature culling functionality of X3DOM that removes submeshes smaller than a given projected pixel size is preserved.

Job queue. Once there is a list of visible submeshes, they need to be processed in order to modify the dynamic buffers for rendering. To achieve this without degrading the interactivity of the rendering, we create a multi-threaded system. Typically, browser Javascript interpreters are single threaded and do not offer parallel processing capabilities. We therefore use HTML5 Web Workers [WHATWG 2016] to create a thread pool for processing outside the rendering thread. For 3D engineering applications, unlike fast moving games, the dynamic buffers do not need to be computed real-time. We therefore only pass the constructed buffers back to the rendering thread when the buffer is valid and after a regular time interval. This avoids the problem of object tearing and constant interruptions to the rendering thread. For each Web Worker, there are two queues for processing associated with it. The first queue contains a list of submeshes to add to the dynamic buffer, and the second queue contains a list of submeshes to remove from it. These queues are updated based on the list of visible meshes. One aim of implementing streaming is to reduce the memory consumption of the system. We therefore prioritize the removal of submeshes from the buffer first, so that the memory footprint is reduced, and then process the submeshes to be added to the buffers. To ensure that the most visually relevant submeshes are displayed, they are sorted by distance and projected pixel size. If the defined memory limit is being approached we then stop adding any more until some memory is freed. This potentially leaves an incomplete model, but still allows the user to zoom in to see more detail based on spatial proximity.



Graphics Memory Manager
Block 1
Block 3
Block 2
Block 4
Block 5
Within the Graphics Memory Manager, blocks can be moved around freely, without caring about the format of the data within them
When the final call to bind the attributes is made, the offset is the sum of the local offset and the offset into the dynamic buffer
Shape 1
Binding Parameters
POSITION: Offset 0, Stride 24
NORMAL : Offset 12, Stride 24
TEX0 : Offset 0, Stride 8
Position & Normal Attributes Interleaved
Block 1/bufferView 1
Submesh 1
Submesh 3
Submesh 4
Block 2/bufferView 2
Submesh 1
Submesh 3
Submesh 4
Texture Coordinates
Within a shape all submesh attributes are contiguous, and submeshes can be added or removed at will
Shape 2
Binding Parameters
POSITION: Offset 0, Stride 12
NORMAL : Offset 0, Stride 12
TEX0 : Offset 0, Stride 8
Block 3/bufferView 1
Submesh 1
Submesh 2
Submesh 3
Position Attribute
Block 4/bufferView 2
Submesh 1
Submesh 2
Submesh 3
Normal Attribute
Block 5/bufferView 2
Submesh 1
Submesh 2
Submesh 3
Texture Coordinates
Different shapes can have different attributes, or attributes stored in different formats

Figure 5: *Implementation overview diagram. In order to share a buffer, either each* `glTFGeometryNode` *needs to be aware of all the other nodes, and the format of their datasets within the buffer, or we remove awareness of the shared buffer so they can each work on their own part of it in isolation. The latter requires an external object to manage allocations within the buffer, hence the* `GraphicsMemoryManager`.

5.3 Memory Manager

In order to manage offloading of data from memory, we introduce a `GraphicsMemoryManger` object per thread which is shared between one or more `glTFGeometryNodes`. This memory manager contains the queues described in §5.2 which prioritize the adding and removal of submehes based on their visibility. When the rendering thread has processed the visibility of the `glTFNode`, it sends a message to the Web Worker containing a list of visible nodes. Upon receiving the message, the worker updates the queues which are then picked up by a processing function ready to change the GPU buffers. In the thread, we construct a copy of the buffer in CPU memory ready to be uploaded to the GPU. At this point, any re-indexing of the index buffer due to changes in the order and a number of meshes is performed. To avoid constant copying of the memory between the memory manager thread and the rendering thread, we use the ability of Web Worker messages to transfer memory object akin to a copy by reference in C++.

Once we have the buffers built in CPU memory, we transfer them to the GPU buffers. Whilst it is possible for the `glTFGeometryNodes` to rebuild their mesh data with new GPU buffers each time, it could lead to significant GPU memory fragmentation, suboptimal memory usage and possibly unpredictable performance characteristics over time. Instead, the nodes request blocks of memory represented by `GraphicsMemoryBlock` objects which facilitate the communication between `GraphicsMemoryManager` and the owner `glTFGeometryNodes`, see Fig. 6. The `GraphisMemoryManager` object then maintains a small number of large attribute and index arrays, which contain the data for a number of `glTFGeometryNode` meshes across multiple threads. When it is time to render one of these nodes, offsets into the large arrays are provided by the `GraphicsMemoryBlock` and set in the binding calls. It should be noted that this does not overcome the 16-bit index array limitation. Each draw call can only address up to 65k vertices at a time. Different sections of the same buffer are used across different draw calls with different binding parameters. When a glTFGeometryNode changes size, it indicates this via the `GraphicsMemoryBlock` object to the `GraphicsMemoryManager`. When instructed, the memory manager will rebuild one or more large dynamic arrays, and update the memory block objects with the new location of this block. The actual gl buffer and the offset into this are passed by `GraphicsMemoryBlock` to the `glTFGeometryNode` which uses them to set the attribute binding parameters. The motivation is that although the arrangement of the attributes within blocks of memory is multipart specific, the GPU memory must be handed globally.

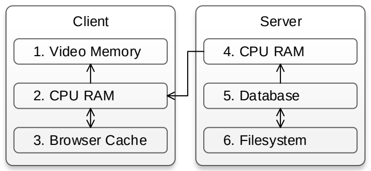


Figure 6: *Different levels of "caching" ordered by relative time to the client seeing the results on screen. It is assumed that local IO is faster than network latency which is likely the case in most setups.*

CPU vs GPU memory. Essentially, the blocks are populated by the `glTFGeometryNode`, and then copied into the larger dynamic buffers for rendering by the `GraphicsMemoryManager`. We use a strategy of double buffering the visible meshes in CPU memory for performance purposes, to avoid rebuilding the meshes constantly. Nonetheless, we do not need to use double the memory, i.e. by storing the same geometry in CPU as well as GPU simultaneously, as there is a `write()` method that sends them directly to the GPU without going through an intermediate array, at least not a persistent one. The geometry node writes data directly from the source described by the `bufferView`, which may be on the server provided by GridFS or on disk in the browser's cache. Thus, the geometry could be removed from the CPU memory entirely if necessary. See Fig. 7 for details. While the `GraphicsMemoryBlocks` are updating, the larger dynamic array is still being rendered as valid even if incomplete model. The same large array is used for multiple draw calls, with the binding parameters changing to address different parts of it. This means that unless for double buffering, one node may bind to an area of the buffer which is being modified by another. To prevent this, an entire multipart mesh is turned off once its data is flagged as being moved, until it has been rewritten, rather than render a broken model.

6 Evaluation and Discussion

The contributions of this paper are two-fold; first, a glTF extension to X3DOM with multipart capability aimed to both increase interoperability and leveraging the functionality of multipart, and

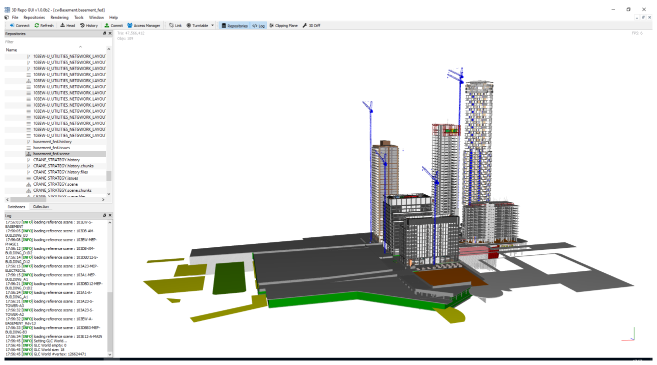
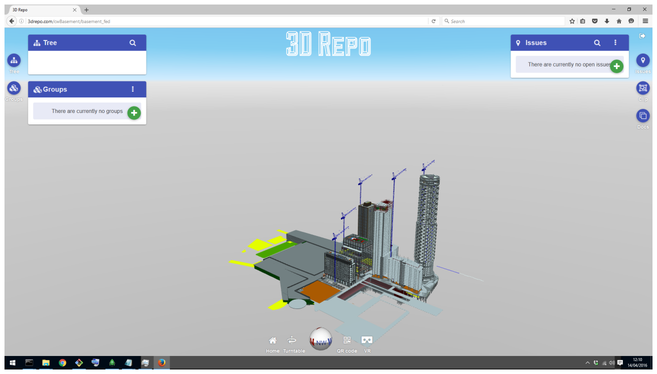



Figure 7: *A large 3D scene federated from 19 individual sub-projects consisting of over 1m components and 50m triangles. Each of the buildings and the basement are delivered by different teams of companies using various types of software such as Autodesk Revit, Graphisoft ArchiCad, Trible Tekla, etc. Left is the 3D Repo GUI desktop application and right the 3drepo.io web app in Firefox. Without support for multipart, this scene would not run in either of the systems. Model courtesy of Canary Wharf Contractors Limited.*

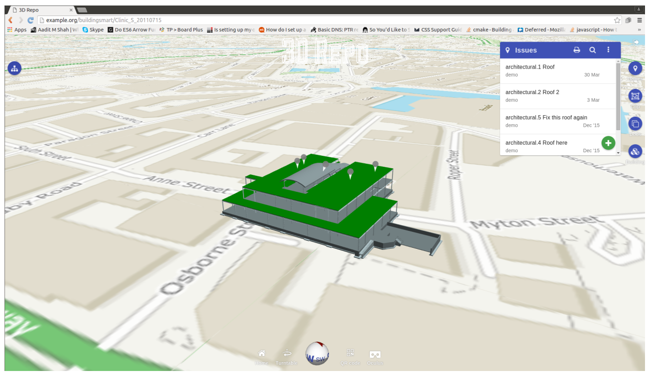
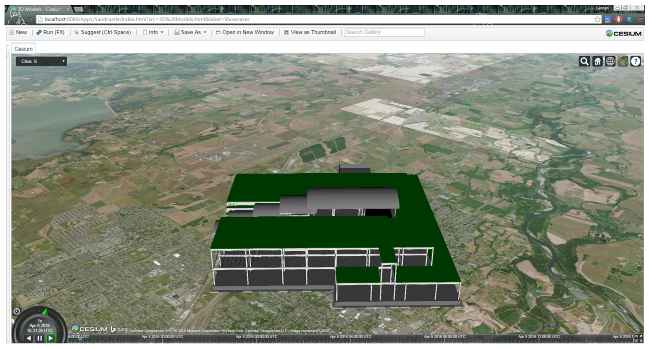



Figure 8: *The same 3D model exported in glTF from 3D Repo to 3drepo.io viewer (top) vs Cesium viewer (bottom). Note that Cesium skips streaming and federation from our extension of glTF but renders multipart correctly. Model courtesy of buildingSMART.*

second, a streaming architecture designed to reduce the memory footprint for rendering large models on memory limited devices. In this section, we evaluate the efficacy of these contributions.

To increase the interoperability of the 3drepo.io system, we introduced the glTF format. This format, although only recently finalized, has already been integrated in some rendering engines. In Fig. 9, we show the same 3D model visualised in 3drepo.io as well as in CesiumJS thanks to glTF. Even though Cesium is not currently able to take advantage of our optimization for multipart and streaming, it correctly visualizes the same model nonetheless. Its only limitation is presenting the model as large co-joint meshes, hence lacking part-based highlighting, plus lacking federation support, see §6.1.

As part of our glTF integration, we introduce multipart as an extension to it. In Fig. 8, we show the extent to which multipart can help render truly large scenes in desktop as well as browser-based applications. In this case, with over one million 3D components, the sheer amount of requests would crash the browser. In addition, the number of required draw calls would make it impossible to render in a dedicated desktop application, let alone a web browser.

Finally, Fig. 2 shows an example of the progressive streaming. The architecture successfully performs its function and culls objects outside the viewing frustum, and restores them when they become visible again. The memory manager also successfully reduces the memory footprint when submeshes go outside the viewing frustum. The multi-threaded nature allows the user to still interactively manipulate the model while the buffers are being re-built. In the figure, we show a viewpoint outside and inside the building. This corresponds to a drop in GPU memory usage from 693 MB to 161 MB. However, in tests on very large models, such as those in Fig. 8, the amount of CPU RAM usage grows to orders of magnitude greater than required by the double buffering. This is part due to the bloat of the glTF format as discussed in the limitations section below. In addition, the rapidly changing memory requirements mean that the Javascript engine is unable to garbage collect quickly enough in order to release the memory. It should be possible to fix this by implementing a more sophisticated memory manager that tailors to specific idiosyncrasies of Javascript memory allocation.

6.1 Limitations

In this section, we describe a number of limitations many of which will be addressed as part of future work.

glTF format. Despite the close resemblance of glTF to SRC, one problem was representing the notion of federated 3D models using the new format. Previously, the federation of several 3D scenes into one was defined in X3DOM as a single X3D file linking to further X3D files for each submodel. Under the specification of glTF, it is not possible to link to another glTF header recursively. Whilst glTF does support multiple scenes within a single file, our preference is to keep the geometry of each subscene separated so that a submodel in a VCS can be updated without the need of updating the federation itself. Due to this restriction, the semantics of the federated models were left within X3DOM, meaning a user who wishes to view a federated model in a different viewer would have to export multiple glTFs and the renderer of choice would have to support visualization of multiple glTF files within the same scene. In addition, the verbosity of the glTF causes memory difficulties within modern browsers. For some of the models shown in the paper, the glTF header can reach upwards of 250 MB.

glTF shader semantic support. Currently, X3DOM creates a shader object procedurally when a shader is imported. It dynamically generates setter methods which are invoked when one of the documented variables is assigned. The methods take the native GL representation and write it to the shader parameter slot using the WebGL API. In theory, it would be possible to add a layer of indirection at this stage, allocating the setter name based on a semantic, rather than the actual variable name in the shader, but keeping the assignment going to the correct slot. This would allow support for glTF semantics/arbitrary shader variable names and remove the requirement that shaders must be X3DOM compliant. Again, because the native matrix type that is encoded is set by the shader, this method should support 4x4 or 3x3 glTF specific normal matrices with no modifications. Nevertheless, this is left for future work.

Arbitrary multipart shaders. Similarly, the principle behind multipart makes the constraint that all submeshes must have the same shader, but it is only the current X3DOM implementation that limits this shader to the X3DOM Material API. In the future, it should be possible to generate multipart shaders, supporting any arbitrary material parameters.

This is somewhat more complicated than supporting them for one shader, because compositing a multipart shader requires that the fragment shader source be modified. Parsing and modifying arbitrary shader sources in Javascript in the browser is unlikely to be a suitable solution. Instead, using the extras facility of glTF, it should be possible to send 'Shader Functions' along with the shaders, much like X3DOM material equivalents are sent currently.

The shader function objects would consist of a source code that can be added directly to a shader with a simple string append. Along with this would be a JSON object describing how to call this function, i.e. the semantics of the parameters and their indices. Using this, multipart shaders could be produced on-the-fly. A shader would be built around this function, with code generated to read the material properties from dynamically generated textures, and the remaining parameters being passed in from the CPU.

Memory management. In terms of memory management, when submeshes are added or removed, no data is changed upfront, only the block size set by the multipart meshes is changed. If a block increased in size, for example, a smart memory manager would take advantage of another shrinking to avoid moving large blocks which have not changed. A very smart one could even avoid signalling a multipart mesh that is about to have its data moved until the very last minute, when another one actually overwrites it. The performance of a dumb memory manager, however, is no worse than if geometry nodes were to directly manipulate large shared arrays themselves. An important thing to note is that the `GraphicsMemoryManager` only rebuilds the arrays on command. It can take the same approach to garbage collection as any other comparable system, and can be as complex or as simple as required. For example, it could rebuild every time a block is resized. Or it could link with the job queue and rebuild after N jobs have been dispatched. An advanced algorithm could intelligently minimise the movement of data, and therefore the amount of time spent by each geometry node reinitialising its blocks.

7 Conclusions

In this paper, we presented a series of novel enhancements to the 3drepo.io Web3D data management system in order to provide higher interoperability as well as better overall ability of rendering large models with fast visual feedback. This required the implementation of adapting the SRC encoder to produce glTF server-side, novel support for MonogDB GridFS-based streaming and the addition of two new elements in X3DOM, namely `glTFNode` and `glTFGeometryNode`. Even though streaming provided faster feedback for the end-users, on its own it would not address the eventual crashing of WebGL when 3D assets outsize the amount of available video memory on the client.

To solve this, we devised a novel `GraphicsMemoryManager` object which is common to all `glTFGeometryNodes` adding and removing their buffer segments dynamically. Despite extending the base glTF specification with multipart, streaming and federations using the extras field, the files still render correctly in third-party viewers, too, fulfilling our desire of increased interoperability of data served by 3drepo.io.

7.1 Future Work

There is a number of extensions and improvements that we would like to add in the near future. These include support for terrain model streaming using a variation of the `BVHRefiner` component of X3DOM driven by MongoDB rather than a filesystem as well as further reduction of geometry into triangle strips in order to decrease the size of geometry by a third and add support for compression such as Open Compressed Triangle Mesh (OpenCTM) [Geelnard 2009] like done previously in [Doboš et al. 2013].

Acknowledgements

Aspects of this work have been part-funded by Innovate UK Digitizing the Construction Sector grant No. 102 054 and the European Institute of Innovation & Technology for which we are thankful.

References

ARNAUD, R., AND BARNES, M. C. 2006. *COLLADA: Sailing the Gulf of 3D Digital Content Creation*. A K Peters/CRC Press, August. ISBN-10: 1568812876.

BEHR, J., AND STURM, T. 2015. Multipart - offline creation and online api. Documentation, Fraunhofer IGD.

BEHR, J., ESCHLER, P., JUNG, Y., AND ZÖLLNER, M. 2009. X3dom: A dom-based html5/x3d integration model. In *Proceedings of the 14th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '09, 127–135.

BEHR, J., JUNG, Y., FRANKE, T., AND STURM, T. 2012. Using images and explicit binary container for efficient and incremental delivery of declarative 3d scenes on the web. In *Proceedings of the 17th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '12, 17–25.

BEHR, J., MOUTON, C., PARFOURU, S., CHAMPEAU, J., JEULIN, C., THÖNER, M., STEIN, C., SCHMITT, M., LIMPER, M., DE SOUSA, M., FRANKE, T. A., AND VOSS, G. 2015. webvis/instant3dhub: Visual computing as a service infrastructure to deliver adaptive, secure and scalable user centric data visualisation. In *Proceedings of the 20th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '15, 39–47.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18* (09), 509–517.

COZZI, P., FILI, T., NIMOIYA, K., LIMPER, M., AND THONER, M., 2016. Khr_binary_gltf. [Online; accessed 7-April-2016].

DIRKSEN, J. 2013. *Learning Three.js: The JavaScript 3D Library for WebGL*. Packt Publishing, October. ISBN-10: 1782166289.

DOBOŠ, J., AND STEED, A. 2012. 3d revision control framework. In *Proceedings of the 17th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '12, 121–129.

DOBOŠ, J., SONS, K., RUBINSTEIN, D., SLUSALLEK, P., AND STEED, A. 2013. Xml3drepo: A rest api for version controlled 3d assets on the web. In *Proceedings of the 18th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '13, 47–55.

DOBOŠ, J. 2015. *Management and Visualisation of Non-linear History of Polygonal 3D Models*. EngD thesis, UCL.

GEELNARD, M. 2009. Open compressed triangle mesh. Software specification v1.0.3.

JUNG, Y., BEHR, J., DREVENSEK, T., AND WAGNER, S. 2012. Declarative 3d approaches for distributed web-based scientific visualization services. In *Dec3D*.

KHRONOS GROUP. 2014. Webgl - opengl es 2.0 for the web. Tech. rep., October.

KHRONOS GROUP, 2016. gltf - efficient, interoperable transmission of 3d scenes and models. [Online; accessed 7-April-2016].

LAVOUÉ, G., CHEVALIER, L., AND DUPONT, F. 2013. Streaming compressed 3d data on the web using javascript and webgl. In *Proceedings of the 18th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '13, 19–27.

LEE, H., LAVOU, G., AND DUPONT, F. 2012. Rate-distortion optimization for progressive compression of 3d mesh with color attributes. *The Visual Computer 28* (02), 137–153.

LIMPER, M., JUNG, Y., BEHR, J., AND ALEXA, M. 2013. The pop buffer: Rapid progressive clustering by geometry quantization. *Computer Graphics Forum 32* (10), 197–206.

LIMPER, M., WAGNER, S., STEIN, C., JUNG, Y., AND STORK, A. 2013. Fast delivery of 3d web content: A case study. In *Proceedings of the 18th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '13, 11–17.

LIMPER, M., THÖNER, M., BEHR, J., AND FELLNER, D. W. 2014. Src - a streamable format for generalized web-based 3d data transmission. In *Proceedings of the 19th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '14, 35–43.

MOUTON, C., SONS, K., AND GRIMSTEAD, I. 2011. Collaborative visualization: Current systems and future trends. In *Proceedings of the 16th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '11, 101–110.

NAM, B., AND SUSSMAN, A. 2004. A comparative study of spatial indexing techniques for multidimensional scientific datasets. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, IEEE Computer Society, Washington, DC, USA, SSDBM '04, 171–.

SCULLY, T., DOBOŠ, J., STURM, T., AND JUNG, Y. 2015. 3drepo.io: Building the next generation web3d repository with angularjs and x3dom. In *Proceedings of the 20th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '15, 235–243.

SONS, K., KLEIN, F., RUBINSTEIN, D., BYELOZYOROV, S., AND SLUSALLEK, P. 2010. Xml3d: Interactive 3d graphics for the web. In *Proceedings of the 15th International Conference on Web 3D Technology*, ACM, New York, NY, USA, Web3D '10, 175–184.

STEIN, C., LIMPER, M., AND KUIJPER, A. 2014. Spatial data structures for accelerated 3d visibility computation to enable large model visualization on the web. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies*, ACM, New York, NY, USA, Web3D '14, 53–61.

SUTTER, J., SONS, K., AND SLUSALLEK, P. 2014. Blast: a binary large structured transmission format for the web. In *Proceedings of the 19th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '14, 45–52.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (Jan.).

WHATWG, 2016. Html living standard. [Online; accessed 7-June-2016].

ZACHMANN, G. 2002. Minimal hierarchical collision detection. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, ACM, New York, NY, USA, VRST '02, 121–128.