Fast Delivery of 3D Web Content: A Case Study

Max Limper[1,2] Stefan Wagner[1] Christian Stein[1,2] Yvonne Jung[1] André Stork[1,2] *

[1] Fraunhofer IGD [2] TU Darmstadt

Abstract

Despite many advances in mesh compression methods within the past two decades, there is still no consensus about a standardized compact mesh encoding format for 3D Web applications. In order to facilitate the design of a future platform-independent solution, this paper investigates the crucial trade-off between compactness of the compressed representation and decompression time. Our case study evaluates different encoding formats, combined with various transmission bandwidths, using different client devices. Results indicate that good compression rates, and at the same time a fast decompression, can be achieved by exploiting existing browser features and by minimizing the complexity of operations that have to be performed inside the JavaScript layer. Our findings are summarized in concrete recommendations for future standards.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality I.3.6 [Methodology and Techniques]: Standards—Languages

Keywords: WebGL, Progressive Meshes, Compression, 3D Formats

1 Introduction

Hardware-accelerated, browser-integrated, plugin-free 3D visualization has gained much attention within the last few years. The availability of low-level graphics APIs like *WebGL* has lead to a wide variety of high-level JavaScript libraries for a rapid, flexible and convenient development of Web-based 3D graphics applications. Popular concepts range from imperative-procedural frameworks (e.g., SpiderGL [Di Benedetto et al. 2010]) to declarative approaches based on scene graphs (e.g., X3DOM [Behr et al. 2009]).

A challenging problem which still remains in today's Web-based graphics APIs is the efficient transmission of 3D scene data from Web servers to client applications. This might seem surprising, as there has been much research dedicated to mesh compression methods within the past two decades [Peng et al. 2005; Jovanova et al. 2008; Maglo et al. 2012]. However, the wide variety of application platforms, including mobile devices with limited CPU horsepower, posts quite different demands on such formats than the powerful CPUs of desktop machines. Since platform independence is achieved through the use of Web technologies, including interpreted languages like JavaScript instead of native code, this constraint becomes even harder. Another very important aspect is that the trade-off between compactness of the compressed representation and decode time has to be reconsidered, given the increasing availability of broadband connections. As a consequence of these open questions, and despite the large amount of different data formats being used among current 3D Web applications, there is still no consensus about a common format for 3D content delivery on the Web, although this is currently a major aim of important actors like the Khronos Group [Trevett 2012].

Within this paper, we thus present a case study on mesh data formats that are in use in current popular 3D graphics APIs for the Web. Besides the very common X3D standard, our study includes compressed and uncompressed binary formats (X3DOM BinaryGeometry [Behr et al. 2012], OpenCTM [Geelnard 2009]) as well as an innovative solution from the *Google Body* project [Blume et al. 2011; Chun 2012]. We also propose and evaluate a new modified version of the OpenCTM format, which produces 25% larger files on average, but also needs only 20-40% of the original decompression time by exploiting the GZIP compression capabilities of HTTP. Our study compares compactness of the compressed representation against time needed for decoding, using two different client devices and evaluating connections with varying bandwidth. Finally, we discuss our experimental results and give directions for further development of mesh formats for the Web.

Our results shed a new light on the trade-off between compactness and decompression time, and are therefore especially valuable for application developers and researches that are interested in an efficient format for compression and transmission of 3D data, relying solely on today's standard browser technology.

2 Mesh Transmission

The following Section 2.1 gives an overview on a comprehensive selection of previous contributions, associated with Web-based streaming of mesh data. Many mesh compression methods are highly specialized on a certain type of input data, e.g. regularly sampled closed manifolds, and are thus not designed to handle arbitrary meshes, including normal and texture information [Gumhold and Strasser 1998; Alliez and Desbrun 2001; Valette et al. 2009]. Furthermore, an exhaustive state-of-the-art report, covering the complete field of mesh compression methods, is clearly out of scope for this paper. The interested reader is referred to survey papers, as provided by Allize and Gotsman or Peng et al. [Alliez and Gotsman 2003; Peng et al. 2005]. We have decided to focus our discussion instead on methods which are directly applicable in today's Web application scenarios, including mobile visualization. However, as progressive mesh compression methods have gained much attention within the past years as candidate technology for Web-based mesh transmission, we have dedicated a few paragraphs to those methods as well.

Section 2.2 presents the formats used in our case study in greater detail, as well as the motivation for selecting them.

2.1 Related Formats

Progressive Meshes. Progressive Meshes (PMs), as originally introduced by Hoppe [1996], provide a continuous, progressive refinement of a polygonal mesh during data transmission over a network. The basic idea is to refine a coarse mesh on the client side, using a stream of vertex split operations, until the original high-resolution mesh has been reconstructed.

As many algorithms have been published in this field within the

*E-Mail for all authors:{firstname.lastname}@igd.fraunhofer.de

Approach	Test CPU	Δ/s	b/v
[Hoppe 1998]	Pentium Pro (200 MHz)	**172K**	153
[Pajarola and Rossignac 2000]	R12000 SGI O2 (300 MHz)	46K	20
[Khodakovsky et al. 2000]	Pentium II Xeon (550 MHz)	32K	15
[Alliez and Desbrun 2001]	Pentium III (NA)	5K	**14**
[Valette et al. 2009]	Intel Quad Core (2.66 GHz)	33K	**14**
[Maglo et al. 2010]	NA (2 GHz)	20K	17
[Maglo et al. 2012]	**Intel Core i7 (2.8 GHz)**	122K	16

Table 1: *Reported decode times and compression performance (bits / vertex) for several progressive mesh compression methods.*

past two decades, achieving impressive compression results and high-quality progressive reconstructions, such methods seem ideally suited for a common Web 3D format, for example by implementing them as extensions of the X3D standard [Fogel et al. 2001; Maglo et al. 2010]. Nevertheless, the surprising result illustrated in Table 1 is that the efficient implementation of Hoppe from 1998, using his original algorithm, still provides the fastest decompression. Note that this withstands even though the *reported* times were compared, i.e. the advances in CPU technology are not taken into account at all. Nevertheless, the compression factor between his method and other coders differs in an order of magnitude as well. A main reason for this trend in our opinion lies in the focus on rate-distortion (RD) performance (i.e., the pure compression factor) within the past decade [Alliez and Desbrun 2001; Valette et al. 2009; Maglo et al. 2012; Lee et al. 2012].

Since RD performance measures the efficiency of a compression scheme independently from any specific bandwidth or CPU power, it does completely *ignore* the trade-off between download bandwidth and decompression time, which has already been mentioned in the early work of Hoppe [1998]. This trade-off is still crucial in today's Web-based real-time visualization scenarios, therefore it is the main focus of this paper (see Section 3.2). A typical example, where a complex progressive decoding method would actually *slow down* the application, could be the real-time inspection of CAD data, using a fast company intranet and a tablet PC with only limited CPU power.

PM algorithms are furthermore designed to work well with regularly sampled, closed surfaces, often delivering poor results for meshes composed of multiple objects (e.g., the backyard scene used in this paper). Splitting such meshes into multiple meshes is a possible solution, but introduces additional complexity. Moreover, most PM algorithms post specific requirements on the input data. A general solution, however, needs to be able to easily handle various kinds of models including non-manifold geometry in an efficient way. Even though PM algorithms exist since more than a decade, we were not able to find a stable implementation, capable of handling models of arbitrary topology with all their attributes, like texture coordinates, colors, normals and so on. Besides, only very few authoring tools support exporting general 3D mesh data as PMs, maybe due to the high implementation complexity and specific demands on input data. In summary, we argue that a direct application of existing PM algorithms in current plugin-free 3D Web applications is not a approriate, because of the following reasons:

- The lack of standardized formats and tools.
- The strict requirements of PMs on input data.
- The time-consuming sequential decompression on the CPU, which is impractical in JavaScript-based Web applications.

Web3D and Mobile Approaches. A wide variety of standards and frameworks for Web-based and mobile 3D visualization has emerged within the past few years. The open ISO standard *X3D* provides a flexible file format and run-time architecture, using the text-based classic VRML-style encoding, a text-based XML representation, or a more compact binary encoding (X3DB) [Web3D Consortium 2008]. Since X3D is not natively supported by Web browsers, specific plug-ins have been the common solution to display X3D content. A plugin-free integration of X3D scenes into the HTML document was first realized by Behr et al. [2009] within the *X3DOM* framework, implementing a so-called *polyfill* layer based on JavaScript and WebGL. With *XML3D*, another approach towards declarative 3D was proposed by Sons et al. [2010]. In contrast to X3DOM, their format proposes a new tag set, which is not based on X3D any more. In recent versions of XML3D, mesh data can also be externalized from the HTML document by storing it in separate JSON files.

While text-based declarative 3D content is human-readable and therefore easy to edit, a problem occurs with such frameworks if the size of the models increases, as parsing text files of several hundred megabytes completely breaks the browser performance. To handle this problem, Behr et al. have discussed image-based transmission (*Sequential Image Geometry*) and raw binary encoding (*BinaryGeometry*) of indices and vertex attributes, such as positions and normals, as two possible approaches for data externalization [Behr et al. 2012] (see also Section 2.2). For the ImageGeometry node, 16 bit vertex positions are encoded in two separate 8 bit images, enabling a simple progressive transmission in two steps. Nevertheless, it requires texture access in vertex shaders, which isn't available on every device and also decreases the render performance.

Gobbetti et al. proposed a method which also uses image-based mesh description format [Gobbetti et al. 2012]. In contrast to X3DOM's ImageGeometry, their method resamples the model data in order to build a tight atlas parametrization of the mesh geometry. This enables them to use the atlas images also for multi-resolution transmission and rendering via simple mipmap operations.

Similar to X3DOM's BinaryGeometry node, Lee et al. [2010] propose to reduce the size of binary mesh data for efficient storage and transmission, using a straightforward local quantization scheme. They argue that geometry compression for mobile graphics requires a careful choice of the compression method in order to maintain interactive decompression rates.

Comparing text-based (VRML/XML) and binary (X3DB) encodings for X3D files, Stocker and Schickel [2011] demonstrated the advantages of binary formats considering both parsing time and especially transfer time, decreasing the file size to 10% of the original uncompressed textual representation and 50% compared to a gzipped equivalent. Further application of GZIP on top of their binary encoding did not show significant compression results. They used a custom viewer, which was implemented as a native browser plug-in.

While the XML-based COLLADA format provides a generalized way for 3D content description, making it easy to exchange assets between different authoring tools, it has not been originally designed for the use with JavaScript and WebGL. To bridge this gap, the COLLADA2JSON project aims at developing a JSON-based format for efficient decoding and transmission within current Web 3D environments, resulting in the WebGL transmission format *glTF* [Robinet et al. 2013; Trevett 2012]. Similar to X3DOM's BinaryGeometry (but based on JSON instead of XML), glTF is separating lightweight, textual data description from the actual geometry data and connectivity data, which are stored in external binary containers.

The *SpiderGL* framework [Di Benedetto et al. 2010], which is based on JavaScript and WebGL, is able to load 3D model data in the

Figure 1: *Textured test models. Top row: regularly sampled scanned artifacts. Bottom row: irregularly sampled game models.*

open COLLADA format [Arnaud and Barnes 2006]. In addition to that, there are currently ongoing efforts to combine SpiderGL with *Nexus*[1] for out-of-core multi-resolution visualization. Likewise, Rodriguez et al. [Rodriguez et al. 2012] recently presented a mixed client-server architecture, where an intelligent server prepares and delivers large point clouds in such a way that even a mobile client can display the data at interactive frame rates. However, such approaches require a special infrastructure with dedicated servers.

Using the *Three.js* framework [Three.js 2010], it was demonstrated that the *OpenCTM* mesh format (see Sec. 2.2) can be utilized in a Web-based context using a JavaScript implementation [Mellado 2012]. The framework also supports loading JSON and COLLADA files as well as the WebGL-Loader format (see Section 2.2). Nevertheless, a comparative evaluation of decompression performance, as provided in our case-study, was missing so far.

2.2 Evaluated Formats

Standard X3D. Encoding 3D mesh data directly in the text-based X3D format has several advantages. First, X3D is an ISO ratified, open standard, supported by many different plugins, and also by the plugin-free X3DOM framework. Already in use for over 10 years, chances are high that X3D versions of mesh data can be obtained with relatively little effort using existing tools and converters. A drawback of the text-based representation is that the corresponding files tend to become pretty large. This can get very time-consuming, especially in a Web-based context, when browsers have to parse the whole XML-based mesh data representation [Behr et al. 2012]. On the contrary, the textual representation is read by optimized, built-in browser functionality, and the JavaScript-based operations on the client side are kept minimal. In addition to that, the size of the text files can be reduced significantly by applying HTTP's GZIP compression (which utilizes LZ77 along with Huffmann encoding). Finally, in terms of file size and compression performance, an XML-encoded X3D representation is expected to behave pretty similar to a JSON-based format, as the payload of the file is unstructured mesh data, which looks the same in both formats. Therefore, we chose to evaluate the XML encoding of the X3D format within our case study, being a representative *text-based* mesh data format.

X3DOM BinaryGeometry (BG). To overcome the main drawbacks of using a text-based X3D representation in a Web-based context, Behr et al. have proposed a binary encoding format for the X3DOM framework, entitled *BinaryGeometry* [Behr et al. 2012]. The general idea of externalizing unstructured mesh data using binary containers, along with a lightweight, structured description in a human-readable format, was enabled by the recent *TypedArray* specification. This allows to download and manipulate binary data directly in a Web page using JavaScript. The idea was also adopted by the recent **glTF** proposal, with the small difference that it uses JSON for the structured information instead of XML. Once the binary data chunks have been downloaded from the server, they can be transferred *directly* to GPU memory. This is a huge advantage in contrast to compressed binary formats, where some decoding operations need to be performed inside the JavaScript layer before the upload to the GPU can be performed. Nevertheless, this direct GPU upload comes at the cost of massively limiting the compression capabilities. The X3DOM BinaryGeometry allows data reduction by supporting indexed triangle strips, which have to be converted from the triangle data during preprocessing. The format also allows to reduce the size of the binary containers by using a 16 bit integer quantization. This introduces an additional translation and scale operation to obtain correctly transformed floating-point positions during rendering, which can, however, be realized efficiently by adapting the corresponding Model-View Matrix [Lee et al. 2010]. Within our experiments, we have made use of both optimizations, stripification and quantization. Because of the interesting property that it does not involve any client-side decode operations, we chose to evaluate the X3DOM BinaryGeometry format as a representative format for *uncompressed binary* mesh data transmission.

OpenCTM. OpenCTM is an open binary format for 3D mesh compression [Geelnard 2009]. It has the great benefit of offering good compression rates, while still providing a relatively fast decompression for native desktop applications. In contrast to formats like VRML/X3D, OpenCTM is solely concerned with encoding mesh data and not encoding any scene description information, like materials, transformations or interactive aspects. From the three available modes of OpenCTM (RAW, MG1 and MG2), we have used the most compact MG2 encoding throughout our experiments. The compact binary encoding mainly builts on entropy reduction and LZMA entropy coding, which combines LZ77 with Markov chains. To reduce the size of the compressed connectivity data, the indices representing the triangles are sorted by the smallest index of each triangle. For efficient LZMA compression, the resulting list is then delta-coded with a very simple scheme which, however, includes a case differentiation during coding and decoding. The model is furthermore subdivided into several uniform cells, and the position of each vertex relative to the corresponding cell origin is computed. The resulting cell-space coordinates are then sorted by their x-coordinate and then delta-encoded. Normals and texture coordinates are delta-encoded as well. As a result, entropy is reduced significantly and LZMA coding can efficiently compress the data. Moreover, vertex data can be stored in a quantized integer format, resulting in good compression rates which are expected to be superior to the simple quantized binary storage formats, like glTF or X3DOM's BinaryGeometry. However, the OpenCTM format is not supported by any browser natively, nor are there any plug-ins available. Therefore, platform-independent Web applications using OpenCTM will first have to decode the compressed data in-

[1] Compare `http://vcg.isti.cnr.it/nexus/`

Model	# Triangles	# Vertices	X3D		BG		CTM-G		Chun		CTM	
			RAW	GZIP	RAW	GZIP	RAW	GZIP	RAW	GZIP	RAW	GZIP
Backyard	4,615	2,625	240	71	79	46	147	43	62	43	31	31
Pharao	16,866	8,437	618	227	185	151	495	116	149	111	81	81
Tractor	49,480	27,251	2,296	617	646	431	1,539	361	506	301	259	259
Bird	184,472	69,948	7,330	2,454	1,958	1,647	5,465	1,197	1,453	1,020	947	948

Table 2: *Size of test models, given in KB. Some formats are build on additional GZIP compression during transmission, as specified in HTTP. Texture images are sent separately for all formats, hence their size has been neglected.*

side the JavaScript layer before being able to upload it to the GPU for rendering. It is therefore an interesting format to investigate the trade-off between compactness of the compressed representation and decode time. Within our case study, OpenCTM is the only *compressed binary* mesh data format.

WebGL-Loader (Chun). The *Google Body* project, which was aiming at a browser-based inspection of human anatomy, resulted in *WebGL-Loader*, a minimalistic JavaScript library for compact 3D mesh transmission [Blume et al. 2011; Chun 2011; Chun 2012]. The latest version performs a vertex cache optimization on the index list [Forsyth 2006]. After an additional optimization for the pre-transform vertex cache, indices are then delta-coded with respect to the current high watermark. Instead of a simple delta encoding, a more advanced parallelogram prediction is used for the attributes. It predicts the next vertex position by constructing a parallelogram with the last three vertices of the triangle strip. The normals are predicted using the cross product of the edges of every triangle. Finally, all the attributes are quantized to less than 16 bit and stored in a UTF-8 file. The UTF-8 file format is a good alternative to binary formats because it can be parsed very quickly with JavaScript, while also providing variable-length encoding. The sorting and delta encoding of the algorithm achieves a comparatively good compression and, combined with the native GZIP implementation of the browser, realizes fast decompression without the need for additional plug-ins. Because of the interesting property of building on *browser features* like UTF-8 and GZIP, the WebGL-Loader format has been included in our experimental comparison.

3 Case Study

Within this section, we describe our case study on compact delivery of 3D Web Content. We have evaluated several encoding methods in terms of compression performance and decompression time, using a desktop machine (i7 CPU, 3.4 GHz) as well as an iPad 3 tablet.

Figure 1 shows the test models used in our experiments. The bird model and the pharao model (left side) are real-world artifacts which have been regularly sampled by two different scanning devices. In contrast, the backyard scene and the tractor model are carefully optimized game models, where the polygon count has been reduced to a minimum yet preserving all important features. All models are textured, and per-vertex normals have been stored along with the mesh data.

3.1 Compression Rate

Table 2 shows a comparison of the file sizes of our test models, using the different encoding formats. As today's browsers support the HTTP option to compress files for transmission using GZIP, we have included GZIP-compressed variants for each encoding method. GZIP uses LZ77 to eliminate repeated character sequences, and it utilizes Huffman encoding for the remaining sequence.

Model	X3D	BG	CTM-G	Chun *	CTM
Backyard	25	0	14	2	49
Pharao	77	0	24	4	127
Tractor	248	0	60	8	353
Bird	880	0	190	25	1139

Model	X3D	BG	CTM-G	Chun *	CTM
Backyard	288	0	452	57	1,455
Pharao	835	0	1,563	144	4,541
Tractor	3,008	0	4,760	470	14,006
Bird	11,055	0	16,863	1,464	47,786

Table 3: *Decompression / parsing times (ms), measured on a desktop PC (top) and on an iPad 3 tablet (bottom).*
** The WebGL-Loader format also offers progressive decoding during the download, which was not exploited here as we show pure decode time without any assumptions about download bandwidth.*

We notice that file sizes differ quite drastically among the various file formats. As expected, the text-based X3D format produces the largest files. However, compressed with GZIP they are roughly a third of their original size. The smallest files are generated using the OpenCTM format, which already uses LZMA compression and therefore does not benefit at all from additional GZIP compression. The X3DOM BinaryGeometry (BG) and the WebGL-Loader encoding are ranked between those two extremes and provide files of approximately the same sizes. However, the WebGL-Loader strongly benefits from GZIP compression and is therefore able to offer superior compression rates. Still, all evaluated formats are significantly less compact than the advanced PM formats referenced in Table 1, as can be seen in Table 4.

As expected, the JavaScript-based decoding of the LZMA-compressed OpenCTM format in most cases took a lot of time (see Section 3.2), so we decided to replace the final LZMA part of the OpenCTM encoder with server-side GZIP compression over HTTP. Results are included in Table 2, labeled *CTM-G*. In line with expectations, this leads to less impressive compression rates. Resulting files are, in the GZIP-compressed form, still more compact than those using the X3DOM BinaryGeometry encoding, but less compact than the ones using the GZIP-compressed WebGL-Loader format. However the result is expected to decode much faster after the JavaScript-based LZMA decompression step has been removed (see Section 3.2).

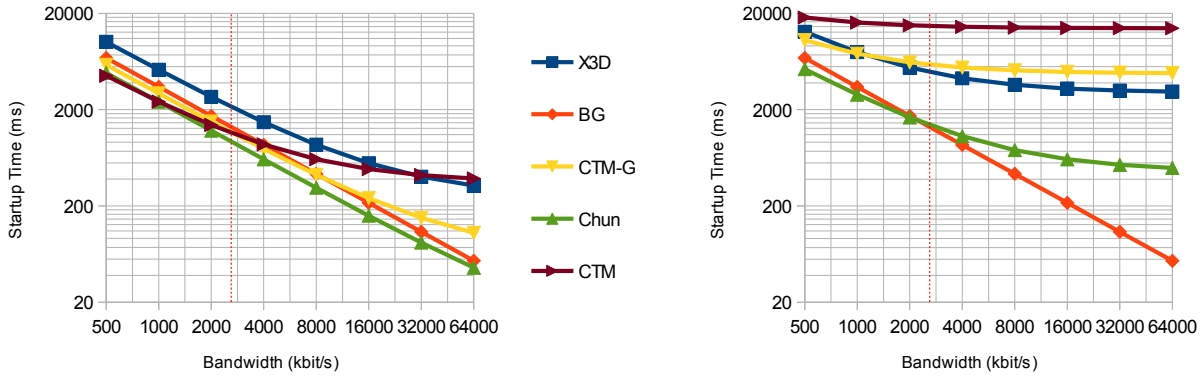

Figure 2: *Combined download and decode time (tractor model). Left: using a desktop PC. Right: using an iPad 3.*

	X3D	BG	CTM-G	Chun	CTM
Δ/s	203K	NA	707K	**5,022K**	132K
b/v	223	149	120	110	**83**

Table 4: *Average compression and decompression performances, using a JavaScript-based implementation on a desktop machine.*

3.2 Transmission and Decompression Speed

Table 3 summarizes the times needed for decompressing the test data on the desktop machine and on the iPad 3.

The X3DOM BinaryGeometry (BG) format does not employ any client-side parsing or decompression of the actual mesh data. As the downloaded buffers are directly pushed to the GPU without any further client-based processing time, decompression time is indicated as zero for all cases. In comparison with the other formats, the fast decompression of the WebGL-Loader format offers the shortest decode time in all cases. Even on the iPad, decode times stay relatively moderate. Additionally, it has to be mentioned that the WebGL-Loader offers to decode the data progressively during download. As the overall decode time is depending on the transmission bandwidth, we decided to show plain decode time in Table 3, and we note that WebGL-Loader is able to potentially provide even slightly better results in practice.

In contrast to the fast BinaryGeometry and WebGL-Loader formats, all other methods perform poorly when decoding larger models on the iPad. Especially the JavaScript implementation of the OpenCTM format using LZMA compression is not feasible in practice, even for moderately-sized meshes, due to decode times of multiple seconds. Our OpenCTM variant relying on GZIP (CTM-G) performs significantly better, especially on the desktop machine. Nevertheless, it is outperformed by the text-based X3D format on the iPad 3.

Figure 2 illustrates combined download and decode times for various transmission bandwidths. Since the decode times of the different test formats are varying in more than an order of magnitude, we decided to use a logarithmic scale for visualization purposes.

As expected, the choice of an "ideal" format also depends on the available bandwidth. Nevertheless, we found some formats being more suited for common use than others. The WebGL-Loader format provides very good results on both devices and throughout all bandwidths, as it provides a compact encoding and at the same time fast decompression. The BinaryGeometry approach works very well on the iPad 3, as the data does not need to be decoded on the client's CPU. On the desktop machine it still performs well, although being outperformed by the WebGL-Loader format due to its limited compression capabilities.

The excellent compression rate of the OpenCTM format only pays off at small download bandwidths, using a relatively powerful desktop machine. Our GZIP-compressed variant, CTM-G, provides better results. On the desktop machine, it is superior to the text-based X3D encoding, whereas this relation is inverted on the iPad, as soon as a transmission bandwidth of more than 2 Mbit/s is available.

4 Design Discussion

Basically we can identify two stages necessary to transfer and present a 3D model to a user's client. In sequential orders these are *Download Stage* and *Decode Stage*. Generally both of them have to be taken into account, when measuring the performance of a compression algorithm[2]. The average global connection speed is currently about 2.8 Mbps [Akamai Technologies 2012], also highlighted with red lines in Fig. 2. This means that, for most users, the connection speed is still the limiting factor. However, for devices with limited processing power like mobile devices, the decoding time of complex compressed formats like OpenCTM can often exceed the download time. This is shown in Figure 3.

The results imply that, for users with very fast connections, one should try to minimize the decode time. This can be achieved by avoiding additional compression and transferring the mesh data directly as binary data, like X3DOM's Binary Geometry or glTF. Otherwise the trade-off between a fast download time achieved by a compact compression and the necessary decoding time has to be evaluated. When using a desktop PC, the OpenCTM format performs best for small bandwidths below 1 Mbit/s due to its high compression rates. However, using an iPad3 it is slower than all the other measured formats. Depending on the bandwidth, X3DOM's BinaryGeometry or the WebGL-Loader format performs best. Overall, the latter seems to offer a good compromise between compactness and decompression performance.

[2]In some application scenarios, even the encoding phase may be time-critical

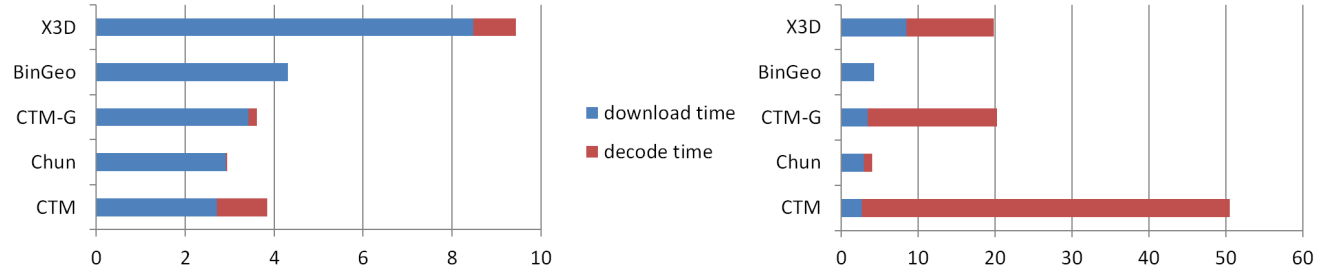

Figure 3: *Download and decode time (ms) for Bird model at fixed connection speed of 2.8 Mbps on a desktop PC (left) and an iPad 3 (right). Note that those measurements don't include the network connection overhead.*

Finally, another aspect not yet taken into account is the *rendering speed*. Some approaches like the X3DOM *Image Geometry* rely on additional decoding steps performed during rendering [Behr et al. 2012]. However, taking into account the potentially very limited GPU power of mobile devices, we think that a general 3D mesh format for the Web should not directly affect the rendering pipeline.

All things considered, our recommendations for a possible future standard format for 3D mesh data delivery on the web, e.g. for the next generation of X3D (version 4.0) or for further development of glTF, are as follows:

- Mesh data which can directly be mapped to GPU structures, like vertex positions and indices, should be stored in binary chunks. Those chunks should be separated from the structured mesh information (e.g. materials, transformations) and - in an ideal case - directly be uploaded to the GPU.
- Depending on the available bandwidth budget, as well as on the expected processing power of the client, a corresponding profile should be available to minimize the overall transmission time. Concretely, a *mobile* profile could e.g. optimizes for decoding speed instead of file size.
- For most use cases, mesh data can be stored in a quantized form, e.g. by using 16 bit attributes, without a significant loss of quality.
- Compression algorithm should be carefully designed with respect to the additional decode time, especially for mobile platforms. A good strategy is to design a format in such a way that it exploits browser's existing compression capabilities, for example by using delta encoding along with GZIP.

5 Conclusion and Future Work

Within this paper, we have compared several mesh encoding formats in terms of their compactness and decompression time needed by a Web browser. We have performed our tests using two different client devices, a desktop PC and an iPad 3 tablet.

Our experiments have revealed the trade-off between compactness of the encoded data and decode time to be crucial for the design of a platform-independent 3D mesh encoding format for the Web. In order to meet the various requirements arising from this trade-off, we propose to provide different encoding profiles, as the choice of the best-suited format strongly depends on the bandwidth budget, as well as on the target device. Those could be a *mobile* profile, which tries to avoid additional decoding, or a *desktop* profile, exploiting browser's built-in compression capabilities. Considering the potential speedup of future devices and JavaScript (e.g., through *asm.js* [Herman et al. 2013]), another profile could also provide more sophisticated encoders to achieve far better compression rates.

For the future, we would like to investigate simple methods for progressive transmission, which still perform good in terms of compactness and decompression performance. We also would like to include encoding times in our evaluation, as this is another important aspect, especially in the context of cloud-based applications, where each second of CPU usage has to be paid.

6 Acknowledgements

Some results of this work have been previously published in the diploma thesis of Stefan Wagner, supervised by Stefan Gumhold, Andreas Stahl (both TU Dresden) and Michael Kopietz (Crytek GmbH) [Wagner 2012]. The tractor and backyard scene models have been kindly provided by Crytek as part of the CrySDK. The bird model is courtesy of the MIT CSAIL database. The Pharao model is courtesy of the EU project 3D-COFORM. This work was also partly funded by the EU project *v-must* (FP7 2007/2013, grant agreement 270404).

References

AKAMAI TECHNOLOGIES. 2012. The state of the internet. Tech. rep. 3rd quarter 2012, Executive Summary.

ALLIEZ, P., AND DESBRUN, M. 2001. Progressive compression for lossless transmission of triangle meshes. In *Proc. SIGGRAPH*, 195–202.

ALLIEZ, P., AND GOTSMAN, C. 2003. Recent advances in compression of 3D meshes. In *In Advances in Multiresolution for Geometric Modelling*, 3–26.

ARNAUD, R., AND BARNES, M. 2006. *Collada*. AK Peters.

BEHR, J., ESCHLER, P., JUNG, Y., AND ZÖLLNER, M. 2009. X3DOM: a DOM-based HTML5/X3D integration model. In *Proc. Web3D*, 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 *Proc. Web3D*, 17–25.

BLUME, A., CHUN, W., KOGAN, D., KOKKEVIS, V., WEBER, N., PETTERSON, R. W., AND ZEIGER, R. 2011. Google body: 3D human anatomy in the browser. In *SIGGRAPH Talks*, 1–1.

CHUN, W., 2011. WebGL-Loader. simple, fast, and compact mesh compression for WebGL. `http://code.google.com/p/webgl-loader/`.

CHUN, W. 2012. WebGL models: End-to-end. In *OpenGL Insights*, P. Cozzi and C. Riccio, Eds. CRC Press, July, 431–454.

DI BENEDETTO, M., PONCHIO, F., GANOVELLI, F., AND SCOPIGNO, R. 2010. SpiderGL: a javascript 3D graphics library for next-generation WWW. In *Proc. Web3D*, 165–174.

FOGEL, E., COHEN-OR, D., IRONI, R., AND ZVI, T. 2001. A web architecture for progressive delivery of 3D content. In *Proc. Web3D*, 35–41.

FORSYTH, T., 2006. Linear-speed vertex cache optimisation. http://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html/.

GEELNARD, M., 2009. Open ctm mesh compression format. http://openctm.sourceforge.net/.

GOBBETTI, E., MARTON, F., RODRIGUEZ, M. B., GANOVELLI, F., AND DI BENEDETTO, M. 2012. Adaptive quad patches: an adaptive regular structure for web distribution and adaptive rendering of 3D models. In *Proc. Web3D*, 9–16.

GUMHOLD, S., AND STRASSER, W. 1998. Real time compression of triangle mesh connectivity. In *Proc. SIGGRAPH*, 133–140.

HERMAN, D., WAGNER, L., AND ZAKAI, A., 2013. asm.js. http://asmjs.org/spec/latest/.

HOPPE, H. 1996. Progressive meshes. In *Proc. SIGGRAPH*, 99–108.

HOPPE, H. 1998. Efficient implementation of progressive meshes. *Computers & Graphics*, 27–36.

JOVANOVA, B., PREDA, M., AND PRETEUX, F. 2008. MPEG4 Part 25: A Generic Model for 3D Graphics Compression. In *Proc. 3DTV-CON*.

KHODAKOVSKY, A., SCHRÖDER, P., AND SWELDENS, W. 2000. Progressive geometry compression. In *Proc. SIGGRAPH*, 271–278.

LEE, J., CHOE, S., AND LEE, S. 2010. Mesh geometry compression for mobile graphics. In *Proc. CCNC*, 301–305.

LEE, H., LAVOUÉ, G., AND DUPONT, F. 2012. Rate-distortion optimization for progressive compression of 3D mesh with color attributes. *Vis. Comput.*, 137–153.

MAGLO, A., LEE, H., LAVOUÉ, G., MOUTON, C., HUDELOT, C., AND DUPONT, F. 2010. Remote scientific visualization of progressive 3D meshes with X3D. In *Proc. Web3D*, 109–116.

MAGLO, A., COURBET, C., ALLIEZ, P., AND HUDELOT, C. 2012. Progressive compression of manifold polygon meshes. *Comput. Graph.*, 349–359.

MELLADO, J., 2012. Js-openctm. port of openctm to javascript. http://code.google.com/p/js-openctm/.

PAJAROLA, R. B., AND ROSSIGNAC, J. 2000. Squeeze: Fast and progressive decompression of triangle meshes. In *Proc. CGI*, 173–182.

PENG, J., KIM, C.-S., AND JAY KUO, C. C. 2005. Technologies for 3D mesh compression: A survey. *J. Vis. Comun. Image Represent.*, 688–733.

ROBINET, F., PARISI, T., AND COZZI, P., 2013. WebGL transmission format (gltf). https://github.com/KhronosGroup/collada2json/wiki/glTF.

RODRIGUEZ, M. B., GOBBETTI, E., MARTON, F., PINTUS, R., PINTORE, G., AND TINTI, A. 2012. Interactive exploration of gigantic point clouds on mobile devices. In *Proc. VAST*.

SONS, K., KLEIN, F., RUBINSTEIN, D., BYELOZYOROV, S., AND SLUSALLEK, P. 2010. XML3D: interactive 3D graphics for the web. In *Proc. Web3D*, 175–184.

STOCKER, H., AND SCHICKEL, P. 2011. X3D binary encoding results for free viewpoint networked distribution and synchronization. In *Proc. Web3D*, ACM, New York, NY, USA, 67–70.

THREE.JS, 2010. Three.js javascript 3D framework. http://threejs.org/.

TREVETT, N. 2012. 3D transmission format. In *Seventh AR Standards Community Meeting Talks*.

VALETTE, S., CHAINE, R., AND PROST, R. 2009. Progressive lossless mesh compression via incremental parametric refinement. In *Proc. SGP*, 1301–1310.

WAGNER, S. 2012. *Effiziente Datenübertragung von Modellen und Texturen für die Verwendung in WebGL*. Diploma thesis, TU Dresden, Germany.

WEB3D CONSORTIUM, 2008. X3D specification. http://www.web3d.org/realtime-3d/specification/current.