# Streaming Compressed 3D Data on the Web using JavaScript and WebGL

Guillaume Lavoué*
Université de Lyon, CNRS
LIRIS, INSA-Lyon

Laurent Chevalier†
VELVET

Florent Dupont‡
Université de Lyon, CNRS
LIRIS, Université Lyon 1

## Abstract

With the development of Web3D technologies, the delivery and visualization of 3D models on the web is now possible and is bound to increase both in the industry and for the general public. However the interactive remote visualization of 3D graphic data in a web browser remains a challenging issue. Indeed, most of existing systems suffer from latency (due to the data downloading time) and lack of adaptation to heterogeneous networks and client devices (i.e. the lack of levels of details); these drawbacks seriously affect the quality of user experience. This paper presents a technical solution for streaming and visualization of compressed 3D data on the web. Our approach leans upon three strong features: (1) a dedicated progressive compression algorithm for 3D graphic data with colors producing a binary compressed format which allows a progressive decompression with several levels of details; (2) the introduction of a JavaScript halfedge data structure allowing complex geometrical and topological operations on a 3D mesh; (3) the multi-thread JavaScript / WebGL implementation of the decompression scheme allowing 3D data streaming in a web browser. Experiments and comparison with existing solutions show promising results in terms of latency, adaptability and quality of user experience.

**CR Categories:** I.3.2 [Computer Graphics]: Graphics systems—Remote systems I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

**Keywords:** 3D Graphics, Web3D, WebGL, Progressive Compression, Level-of-Details, JavaScript.

## 1 Introduction

Technological advances in the fields of telecommunication, computer graphics, and hardware design during the two last decades have contributed to the development of a new type of multimedia: three-dimensional (3D) graphic data. This growing 3D activity was possible thanks to the development of hardware and software for both professionals (especially 3D modeling tools for creation and manipulation) and for end-users (3D graphic accelerated hardware, new generation of mobile phones able to visualize 3D models). Moreover, the visualization of 3D content through the web is now possible thanks to specific formats like X3D, technologies like the very recent WebGL specification, which makes the GPU controllable by JavaScript, and norms like HTML 5. The Web3D concept (i.e. communicating 3D content on the web) is seen as the future of the Web 2.0, and is supported by many organizations like the W3C and the Web3D consortium.

Numerous application domains are directly concerned by 3D data (some of them are illustrated in figure 1): Mechanical engineering, scientific visualization, digital entertainment (video games, serious games, 3D movies), medical imaging, architecture, cultural heritage (e.g. 3D scanning of ancient statues). In most of these applications, 3D data are represented by polygonal meshes, which modelize the surface of the 3D objects by a set of vertices and facets (see the zoomed part on the right in figure 1).

This type of data is more complex to handle than other media such as audio signals, images or videos, and thus it has brought new challenges to the scientific community. In particular, the interactive remote visualization of 3D graphic data in a web browser remains a challenging issue. As observed by Di Benedetto et al. [2010], the delivery and visualization of 3D content through the web has came with a considerable delay with respect to other digital media such as images and videos, mainly because of the higher requirements of 3D graphics in terms of computational power. First systems used Java Applets or ActiveX controls to expose 3D data on a web browser, however recently the WebGL specification has been introduced [Khronos 2009] and will probably boost the use of 3D data on the web. A lot of industries have interest in providing 3D content through the web, including online video games (to represent virtual worlds), 3D design or e-business companies. Moreover like existing huge repositories of pictures (e.g. Flikr) or videos (e.g. YouTube), community web 3D model repositories are now appearing, such as Google 3D Warehouse. Like stated in the recent study from Mouton et al. [2011], web applications have major benefits compared to desktop applications: firstly, web browsers are available for all mainstream platforms including mobile devices, and secondly the deployment of web applications is straightforward and does not require the user to install or update softwares or libraries other than the browser. All these reasons argue for a high increase of the use of 3D remote graphics on the web in the near future.

An efficient system for interactive remote visualization of large 3D datasets needs to tackle the following technical issues:

1. Removing the latency; in most of existing systems, 3D data are fully loaded in an uncompressed form. Therefore, there is latency before visualization. This latency is particularly critical for web applications.

2. Allowing the adaptation of the levels of details to different transmission networks and client hardwares, in order to allow a good frame-rate even in case of low-power devices such as smartphones.

These issues can be resolved by the use of progressive compression techniques [Peng et al. 2005]. Indeed, progressive compression allows to achieve high compression ratio (and thus fast transmission) and also to produce different levels of details (LoD), allowing to adapt the complexity of the data to the remote device by stopping the transmission when a sufficient LoD is reached. Moreover, users

*e-mail:glavoue@liris.cnrs.fr
†e-mail:laurent.chevalier@velvet.eu.com
‡e-mail:fdupont@liris.cnrs.fr

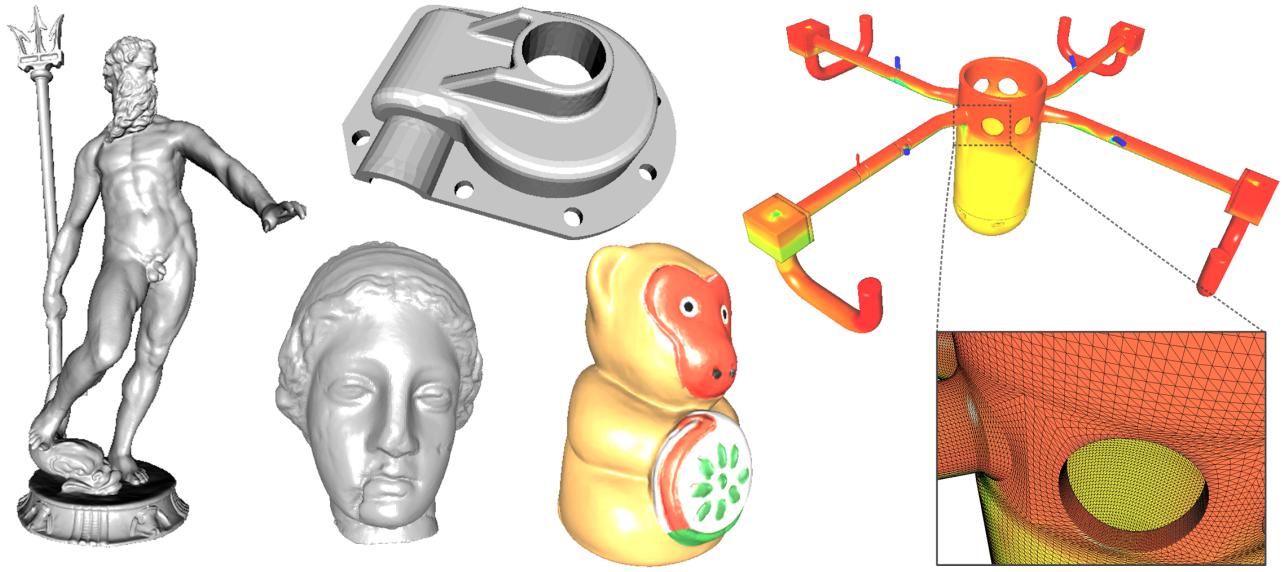

**Figure 1:** *Several 3D graphical models illustrating different application domains. From left to right: Neptune (250k vertices - cultural heritage) , Venus (100k vertices - cultural heritage), Casting (5k vertices - mechanical engineering), Monkey (50k vertices - digital entertainment) and Tank (160k vertices - scientific visualization) which represents a nuclear power plant tank with temperature information (provided by R&D division of EDF).*

are able to quickly visualize a coarse version of the 3D data first, instead of waiting for full objects to be downloaded before they can be displayed. Figure 2 illustrates different levels of details for the power plant tank 3D model. These functionalities are able to neutralize the time latency even for huge data and make possible real-time interactions (i.e. high frame rate) even for mobile devices.

We introduce a technical solution for web-based remote 3D streaming and visualization, which tackles the two issues mentioned above. Our system runs natively in a web browser without any plug-in installation and leans upon three strong features: (1) a dedicated progressive compression algorithm for 3D graphics data with colors, producing a binary compressed .P3DW file which allows a progressive decompression with several levels of details; (2) the introduction of *Polyhedron_HalfEdge.js*, a JavaScript halfedge data structure allowing geometrical and topological operations on a 3D mesh; (3) the multi-thread JavaScript implementation of the associated decompression scheme, using *Polyhedron_HalfEdge.js* and WebGL, allowing 3D data streaming in a web browser. The next section details the state of the art about 3D object compression and web-based 3D data delivery and visualization; then section 3 details our progressive compression algorithm while section 4 presents our JavaScript halfedge data structure and the implementation of the decompression. Finally section 5 illustrates several compression and streaming experiments and comparisons with state of the art, while section 6 concludes the paper.

## 2 State of the art

### 2.1 Progressive compression

The main idea of progressive (or multi-resolution) compression is to represent the 3D data by a simple coarse model (low resolution) followed by a refinement sequence permitting an incremental refinement of the 3D model until the highest resolution (see figure 2, from left to right). This functionality is particularly useful in the case of remote visualization since it allows adapting the level of details to the capacity of the visualization device, the network bandwidth and the user needs.

Most of existing approaches consist of decimating the 3D mesh (vertex/edge suppressions) while storing the information necessary for the process inversion, i.e. the refinement (vertex/edge insertions during the decoding). The existing approaches differ in the way they decimate the mesh and store the refinement information (where and how to refine?).

Since the pioneer work of Hoppe [1996], a lot of methods have been introduced [Taubin and Rossignac 1998; Pajarola and Rossignac 2000; Alliez and Desbrun 2001; Gandoin and Devillers 2002; Peng and Kuo 2005; Valette et al. 2009; Peng et al. 2010; Lee et al. 2012], however most of them are not adapted for remote visualization, indeed some critical issues have been almost totally ignored by the scientific community:

- Most of existing progressive compression techniques have concentrated their efforts on optimizing the compression ratio; however in a remote visualization scenario, improving the quality of the levels of details (see figure 2) is more important than gaining a few bits on the size of the whole compressed stream. Only some very recent methods have tried to focus on the quality of the levels of details [Tian and AlRegib 2008; Peng et al. 2010; Lee et al. 2012].

- Only few existing techniques [Tian and AlRegib 2008; Lee et al. 2012] allow the progressive compression of attributes like color, texture or other information attached to the 3D data. The level of details management of these attributes is particularly important with regards to their influence on the perceptual quality of the visualized object.

- One of the main objectives of progressive compression is to speed up the transmission of the 3D data by decreasing the size of the content to transmit. However if the decompression time is too long, then the user has lost all the benefit of the compression since even if the transmission is fast, a long decompression time will induce a latency for the user. Therefore

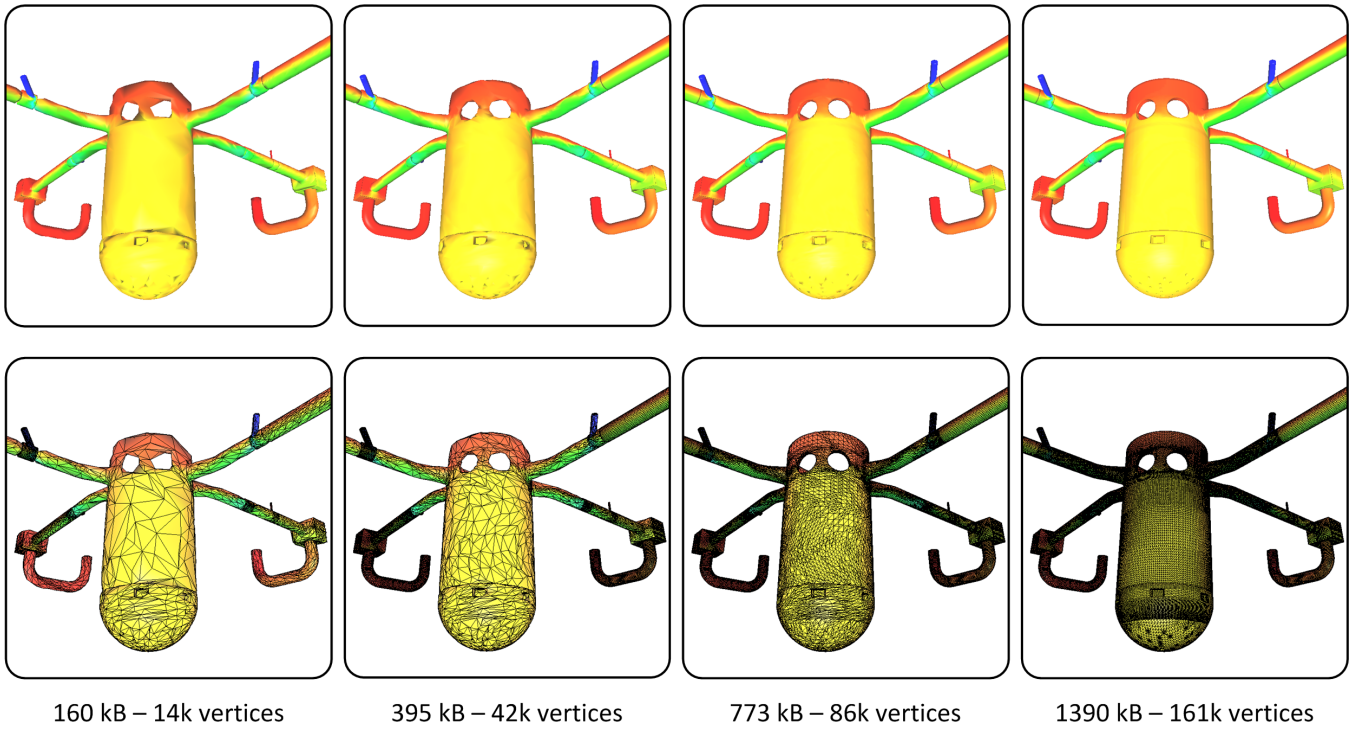


**Figure 2:** *Progressive decoding of the compressed P3DW file corresponding to the Tank model (with and without wireframe). From left to right : 12%, 28%, 56% and 100% of the stream are respectively decoded. Such progressive decoding allows to stream the data in order to obtain very quickly a good approximation of the model. Moreover, it allows an adaptation to the client device hardware (for a high performance workstation the full resolution model can be loaded and visualized interactively but in case of a smartphone, a low resolution version has to be preferred). The original ASCII OFF file size is 12762 kB.*

a critical issue for a compression scheme is to optimize the decompression time by relying on simple yet efficient schemes. However at present very few progressive compression algorithms have focused on optimizing and simplifying this step. Our objective is to focus on that point to take full advantage of the gain in transmission time provided by the small size of the compressed stream. Such a simplified decoding algorithm would also make possible its transcription in JavaScript for a full web integration using WebGL. This time issue is of major importance for a realistic industrial use of progressive compression.

In our system, we propose an adaptation of the recent progressive compression algorithm from Lee et al. [2012] which fulfills these requirements (quality of the LoD, color handling, decompression simplicity). Our algorithm produces a binary compressed file (.P3DW) that allows a fast and simple progressive decompression. We have selected the algorithm from Lee et al. [2012] since it produces among the best state of the art results regarding rate-distortion performance and it is publicly available in the MEsh Processing Platform (MEPP) [Lavoué et al. 2012] (`http://liris.cnrs.fr/mepp/`).

### 2.2 Remote 3D visualization on the web

Like stated in the introduction, the delivery and visualization of 3D content through the web has came with a huge delay with respect to other digital media, mainly because of the higher requirements of 3D graphics in terms of computational power. Some web solutions exist, like ParaViewWeb [Jomier et al. 2011], that compute the 3D rendering on the server side and then transmit only 2D images to the client. However we focus this state-of-the-art on client side rendering solutions where the full 3D data are transmitted.

Many works on web-based remote visualization of 3D data have been conducted for remote collaborative scientific visualization for which an overview has recently been conducted by Mouton et al. [2011]. ShareX3D [Jourdain et al. 2008] provides a web-based remote visualization of 3D data; the web rendering is based on Java. The COVISE (COllaborative VIsualization and Simulation Environment) platform, in its last version [Niebling and Kopecki 2010], offers a web client implemented using JavaScript and WebGL. However these systems, like most of existing ones, make use of XML-based ASCII format such as VRML or X3D [Jourdain et al. 2008] or JavaScript vertex arrays [Niebling and Kopecki 2010] to exchange the 3D data, which involves a significant latency due to the large file size.

To resolve this latency issue, some compression methods have been proposed. A simple binary encoder (X3Db) has been introduced for the X3D format, however it produces poor compression rates (around 1:5 regarding the original ASCII X3D size). Isenburg and Snoeyink [2003] propose a compression method integrated to the VRML/X3D format however the decompression needs a dedicated Java client. Recently, several interesting compression methods, allowing decompression in the web browser, have been proposed: Google introduces *webgl-loader* [Chun 2012] (http://code.google.com/p/webgl-loader/), a WebGL-based compression algorithm for 3D meshes in the context of the Google Body project [Blume et al. 2011]; it is based on UTF-8 coding, delta prediction and GZIP and produces compression ratio around 5 bytes/triangle (for encoding coordinates, connectivity and normals). Behr et al. [2012] propose to use images as binary containers for mesh geometry within the X3DOM framework;

they obtain compression ratio around 6 bytes/triangle in the best configuration. These two latter approaches produce interesting compression ratio and a fast decoding mostly on the GPU. However they are single-resolution hence they do not allow streaming or level of details selection.
Some authors have proposed solutions for 3D data streaming; Di Benedetto et al. [2010], in their JavaScript library *SpiderGL* (which leans upon WebGL), propose a structure for managing levels of details however it is only suited for adaptive rendering since no streaming or compressed format is associated. In the context of remote scientific visualization, Maglo et al. [2010] propose a remote 3D data streaming framework based on a progressive compression algorithm; however they need a dedicated desktop client (Java and C++) to decompress the binary stream. A similar early version of this latter work was proposed by Chen and Nishita [2002] who consider an older and less efficient progressive compression scheme. Some works on 3D streaming have also been conducted in the context of online gaming and virtual world, Marvie et al. [2011] present a streaming scheme for 3D data based on a X3D extension. Their work is based on the progressive mesh representation introduced by Hoppe [1996] and thus allows streaming and level of details selection. However this progressive representation is not really compressed, moreover once again a dedicated desktop client is needed to visualize the data (no web integration). Finally, very recently, Gobbetti et al. [2012] propose a nice multi-resolution structure dedicated to the rapid visualization of large 3D objects on the web, however it requires complex preprocessing steps (parameterization and remeshing) and cannot handle arbitrary meshes.

In our system, we consider a compressed representation of the 3D data (in the form of a P3DW file) which provides good compression ratio: around 1:10 equivalent to roughly 3 bytes/triangle, when encoding coordinates and connectivity. This compressed format allows a progressive decompression and streaming. The whole decompression process is implemented in JavaScript and WebGL hence the 3D streaming works directly on a web browser without need of plug-in.

# 3 Web-based progressive compression

Like stated above, we need a simple decompression scheme to make possible a JavaScript/WebGL implementation providing reasonable processing time. For this purpose we have made a web-based adaptation of the progressive algorithm from Lee et al. [2012] which is based on the valence-driven progressive connectivity encoding proposed by Alliez and Desbrun [2001]. During the encoding, the mesh is iteratively simplified into several levels of details until reaching a base mesh (around a hundred vertices). At each simplification iteration, the information necessary for the refinement is recorded; it contains connectivity, geometry and color data. The encoding process and data are presented below.

## 3.1 Iterative simplification

The encoding process is based on the iterative simplification algorithm introduced by Alliez and Desbrun [2001]. At each iteration, the algorithm decimates a set of vertices by combining *decimation* and *cleansing* conquests to get different levels of details (these two steps are illustrated in figure 3). The decimation conquest consists in removing a set of independent vertices using a patch-based traversal, and then retriangulating the holes left (see fig.3.b). Then, the cleansing conquest removes vertices of valence 3 in order to regularize the simplified mesh (see fig.3.c). For regular meshes, the combination of these two conquests performs the inverse $\sqrt{3}$ subdivision. For non-regular meshes, the retriangulation follows a deterministic rule so that the mesh connectivity is kept as regular as possible during the simplification process. These iterative simplification steps are applied until reaching a coarse base mesh.

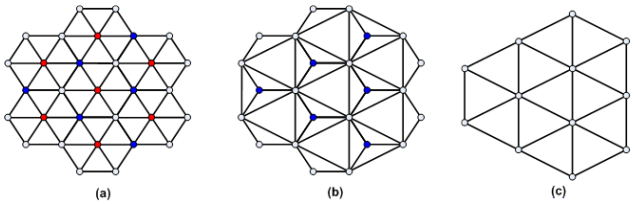


**Figure 3:** *One iteration of the progressive encoding algorithm. (a) Original mesh, (b) intermediate result after decimation (red vertices are removed) and (c) final result after cleansing (blue vertices are removed).*

## 3.2 Encoded information

As presented above, during the encoding the mesh is iteratively simplified (decimation + cleansing). At each simplification step, the connectivity, geometry and color information of each removed vertex are written in the compressed stream, to allow the refinement at the decoding.

For the connectivity, like proposed in [Alliez and Desbrun 2001], our algorithm only encodes the valences of the removed vertices, plus some $null\_patch$ codes when vertices were not able to be simplified for irregular connectivity reasons. This valence information is sufficient for the connectivity refinement at the decoding. In [Alliez and Desbrun 2001] the valence values are fed to an arithmetic coder. In our algorithm, we wish to avoid a complex arithmetic decoding in JavaScript, hence we consider a straightforward binary encoding. We use 3 bits per valence value and $null\_patch$ code, this gives an average of 10 bits/vertex for the connectivity.

For the geometry, Alliez and Desbrun [2001] first apply a global and uniform quantization to the mesh vertex coordinates. When a vertex is removed, its position is predicted from the average position of its 1-ring neighboring vertices and only the prediction residue is encoded (once again using arithmetic coding). Lee et al. [2012] improve this geometry encoding by introducing an optimal adaptation of the quantization precision for each level of details. In our web-based algorithm, we consider a global uniform quantization (on $Q$ bits) of the (x,y,z) coordinates and then we simply binary encode them (without any prediction nor entropy coding).

For the color, Lee et al. [2012] first transform the RGB color components into the CIE L*a*b* representation which is more decorrelated than the RGB space; thus it is more appropriate for data compression. In [Lee et al. 2012], the L*a*b* components are then quantized adaptively according to the level of details, and predicted using a color-specific rule. The authors also introduce a color metric to prevent the removal of visually important vertices during the simplification. In our algorithm we also use this metric. However, like for connectivity and geometry, the color encoding is simplified: we apply a 8 bits quantization and a simple binary encoding of the L*a*b* components (no prediction, nor entropy

coding).

In practice, for $Q = 12$ bits of geometric quantization, without color information, a 3D model is compressed using 46 bits/vertex ($\approx$ 2.9 bytes/triangle). Basically we have made the choice of losing a part of the compression performance to decrease the complexity and the decompression time.

Figure 4 presents the compressed stream. This stream is naturally decomposed into several parts, each standing for a certain level of detail and each containing connectivity (C), geometry (G) and color (Cl) information. The first part is the base mesh (usually around a hundred vertices) which is encoded in a simple binary form. Then, each part of the stream, together with the already decompressed level, allows building the next level of details. At the end of the stream decoding, the original object is retrieved (lossless compression).

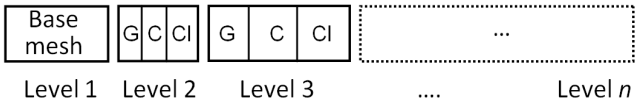


**Figure 4:** *Format of the encoded stream. Each part of the stream contains geometry (G), connectivity (C) and color (Cl) data needed for mesh refinement.*

# 4 Web integration using WebGL and JavaScript

## 4.1 An halfedge data structure in JavaScript

The decompression mechanism is basically the following: first the base mesh is decoded, and then each layer of the compressed stream provides connectivity, geometry and color information to construct the next level of details. This mesh refinement corresponds to steps $(c) \rightarrow (b)$ and $(b) \rightarrow (a)$ from figure 3 and thus needs quite complex topological operations on the current 3D mesh like vertex insertion, face merging, etc. These topological operations need an efficient data structure for representing the mesh and allowing fast adjacency queries. In classical compiled C++ Computer Graphics applications, the most widespread structure is the *halfedge* data structure (see figure 5), like used in the CGAL library (`http://www.cgal.org`). It is an edge-centered data structure capable of maintaining incidence information of vertices, edges and faces. As illustrated in figure 5, each edge is decomposed into two halfedges with opposite orientations. The halfedges that border a face form a circular linked list around it. Each halfedge stores pointers to its incident face, its incident vertex and it's previous, next and opposite halfedges. Every faces and vertices store a pointer to their incident halfedge. This data structure is able to answer local adjacency queries in constant time.

We have implemented the *Polyhedron_HalfEdge.js* library which describes a complete halfedge data structure in JavaScript. This library allows to represent vertices, edges, faces and color attributes and it provides access to all incidence relations of these primitives (e.g. all incident faces from a given vertex). Moreover some complex topological operations have been implemented like `create_center_vertex` which adds a vertex to the barycenter of a face and connect it with its neighbors (see step $(c) \rightarrow (b)$ in figure 3), `join_facets` which merges two facets into a single one of higher degree, `split_face` , `split_vertex`, `fill_hole`, etc.

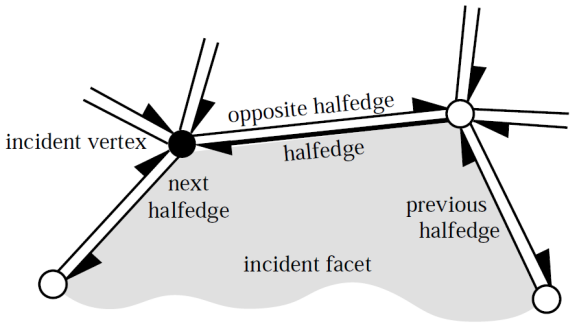


**Figure 5:** *The halfedge data structure. Reprinted from [Kettner 1999].*

Our library relies on the *Three.js* library (`https://github.com/mrdoob/three.js`) for the representation of basic geometrical primitives (vertex, faces, 3D positions); *Three.js* is actually one of the most popular JavaScript libraries for WebGL.

## 4.2 Implementation of the decompression

Basically five main components have been implemented in JavaScript to allow the 3D data streaming:

- A *binary reader* which decodes the connectivity, geometry and color information from the binary P3DW file, in a streamed way (i.e. level by level). For the streaming, nothing is implemented in the server side; we just make one single XMLHttpRequest. We use the *responseText* property of the XHR to read the stream progressively. The maximum size of a level (see figure 4) is estimated using the number of already decompressed vertices. The corresponding refinement is launched as soon as enough data are available. In future implementations, we plan to add a header at the beginning of the compressed stream with the sizes of all levels, to read exactly the necessary numbers of bytes. A minimal implementation on the server side could also bring some interesting features; for instance we could imagine authorizing the transmission of the first levels of details in a free basis, and authorizing the next ones after payment (like in online image libraries).
- The *base mesh initializer* which constructs the base mesh (usually several dozens of vertices).
- The *LoD decompressor* which constructs the next level of details, starting from an already decompressed level of details and using the decoded connectivity, geometry and color information. This component computes the necessary geometrical and topological operations (steps $(c) \rightarrow (b)$ and $(b) \rightarrow (a)$ from figure 3). It is mostly based on our *Polyhedron_HalfEdge.js* library presented above.
- The rendering and user interaction management, which are mostly based on functions from the *Three.js* library.
- An efficient multi-thread implementation which enables user interactions while decompressing the levels of details, hence yielding an improved quality of user experience. JavaScript owns the important limitation of being executable only in one single thread. HTML5 has very recently provided a solution, the *Web Workers*, which allow to run scripts in background threads. The problem is that these *Workers* do not have access to the DOM (Document Object Model) hence they have to constantly communicate their data to the main

thread. Fortunately the *Array Buffers* have been very recently introduced (September 2011) and allow a zero-copy transfer between threads (like a pass-by-reference). In our implementation the decompression runs as background thread and we use this brand new Array Buffer technology to quickly send the decoded information to the main thread.

Note that an important effort of implementation was dedicated to the minimization of the garbage collection. Indeed the garbage collector is particularly harmful in JavaScript applications. It may induce very visible pauses (half second or more). This fact is particularly true for our refinement algorithms which allocate and destroy of lot of elements (vertices, faces, halfedges) during the topological operations with a naive implementation. Therefore we have optimized the object recycling; no object is destroyed in our current implementation.

All the features presented above are integrated into a web platform illustrated in figure 6. Once the user has chosen a remote P3DW file, the levels of details are streamed and visualized interactively, as illustrated in the accompanying video that shows a live recording of the streaming.

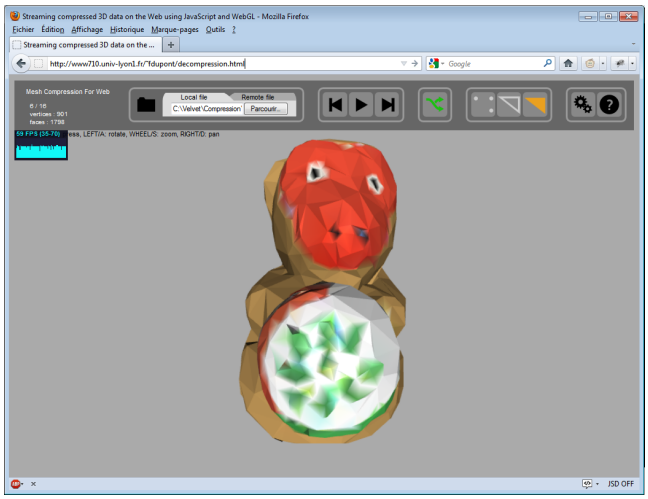

**Figure 6:** *Illustration of our web page for streaming compressed 3D data.*

# 5 Experiments and comparisons

## 5.1 Compression ratio

We have conducted experiments using the objects presented in figure 1. Table 1 presents respectively the original sizes of the 3D models (OFF ASCII file format) and the sizes of the compressed streams corresponding to a lossless compression (all levels of details). The geometry quantization precision $Q$ was fixed between 11 bits and 13 bits according to the model, so as to obtain no perceptible difference with the uncompressed version. We can observe that the compression ratios are very good (around 1:10), which is between 2 and 3 times better than a ZIP compression.

Table 2 illustrates a comparison with concurrent state of the art methods: Google webgl-loader [Chun 2012] (UTF8 codec) and the X3DOM's image geometry approach [Behr et al. 2012] (SIG codec). For a fair comparison, we have taken two models

| Name (#vertices) | OFF | ZIP | P3DW |
|---|---|---|---|
| Neptune (250k) | 19,184 | 6,788 (2.8) | 1,509 (12.7) |
| Tank (160k) | 12,762 | 2,980 (4,3) | 1,390 (9.2) |
| Venus (100k) | 7,182 | 2,701 (2.7) | 609 (11.8) |
| Monkey (50k) | 5,105 | 1,856 (2.8) | 430 (11.9) |
| Casting (5k) | 332 | 112 (3) | 28 (11.9) |

**Table 1:** *Baseline evaluation of the compression rates: file size (kB) and associated reduction factor (in parenthesis) of our compressed format (P3DW) against standard ASCII format (OFF) and ZIP compression (ZIP), for the test models from figure 1.*

considered by these methods and we have reproduced exactly the parameters: 16 bits quantization (for position and normals) for the Bunny (as the X3DOM's approach), and 11 bits for position and 8 bits for normal for the Happy Buddha (as the webgl-loader approach). We usually not consider the compression of normals in our approach, but we have included them for these comparisons. For the X3DOM SIG approach we have selected the best setting (SIG with PNG compression). We also include results from the X3Db codec provided in [Behr et al. 2012]. We can observe that our compression rate is quite similar to these two recent concurrent approaches. Such compression factor will obviously fasten the transmission time and thus reduce the latency for remote visualization. However the main feature of our compressed P3DW format is that it allows a progressive decoding and therefore yields to very quickly visualize a coarse version of the 3D data (then progressively refined) instead of waiting for the full object to be downloaded before it can be displayed.

## 5.2 Quality of the levels of details

Figure 7 illustrates the geometric error associated with the different levels of details according to the percentage of decoded vertices/facets, for the Venus model. We can easily see than the visual appearance of the decoded model becomes very quickly correct. Indeed, after decoding only 10% of the elements (this corresponds basically to decoding 10% of the P3DW file, which represents around 1% of the original ASCII file size) we already have a nice approximation of the object.

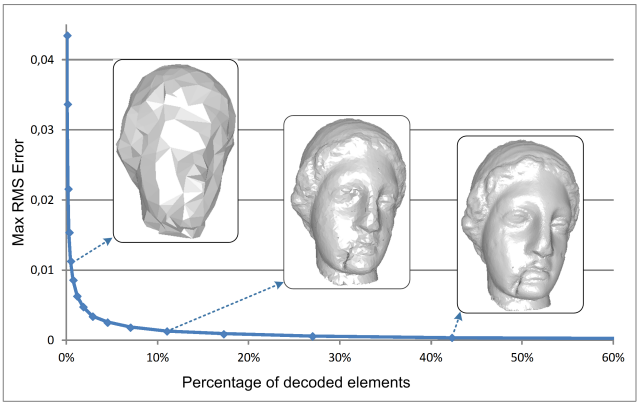


**Figure 7:** *Maximum Root Mean Squared error vs percentage of decoded elements for the Venus model.*

| Name (#vert.) | OFF | ZIP | P3DW | X3Db | X3DOM | UTF8 | GZIP UTF8 |
|---|---|---|---|---|---|---|---|
| Bunny (35k) | 2,448 | 860 | 462 | 937 | 406 | NA | NA |
| Happy Buddha (540k) | 41,623 | 10,132 | 4,523 | NA | NA | 6.789 | 2.849 |

**Table 2:** *Comparison with X3Db, Google webgl-loader [Chun 2012] (UTF8 codec) and the X3DOM's image geometry approach [Behr et al. 2012] (SIG PNG codec). File sizes are given in kB.*

### 5.3 Decompression time

Like stated in the introduction, the decompression time is of great importance for the usability of the method. Table 3 illustrates the decompression times for our web platform using the Mozilla Firefox browser on a 2GHz laptop. The data are downloaded/streamed from a remote server using a high speed connection in order to make the downloading time negligible. The table shows that the decoding time has a linear behavior regarding the number of decoded elements. On average our platform is able to decode between 20k and 30k vertices per second. This timing is very good for a JavaScript implementation and allows bringing a very significant gain in a remote visualization scenario, in term of quality of experience. Even if the whole decompression may take several seconds for large objects, it is interesting to see that whatever the original size of the data, we obtain very quickly several thousands of vertices, hence a very nice approximation. Note that these results correspond to the multi-thread implementation, the mono-thread version is around 25% faster (but the user cannot interact with the object until it is fully decompressed).

| Name | 10% | 20% | 50% | 100% |
|---|---|---|---|---|
| Neptune | 1.0 (25k) | 1.8 (50k) | 4.6 (125k)) | 11.2 (250k) |
| Tank | 1.1 (16k) | 1.6 (32k) | 4.0 (80k) | 8.8 (160k) |
| Venus | 0.4 (10k) | 0.8 (20k) | 1.5 (50k)) | 3.2 (100k) |
| Monkey | 0.3 (5k) | 0.4 (10k) | 0.9 (25k)) | 1.8 (50k) |
| Casting | 0.1 (0.5k) | 0.1 (1k) | 0.1 (2.5K)) | 0.2 (5k) |

**Table 3:** *Decompression time (in seconds, for a 2GHz laptop) and associated numbers of vertices (in parenthesis) according to the percentage of decoded elements, for the test models from figure 1.*

### 5.4 Remote 3D streaming results

We have conducted some experiments and comparisons in order to evaluate the gain, in term of quality of experience, of our web 3D streaming technical solution compared with concurrent approaches.

#### 5.4.1 Baseline results

In this first experiment, we have considered the remote visualization of the Neptune and Venus 3D models through a good ADSL Internet access (10 Mbit/s), and have compared our results with the transmission in uncompressed form and the transmission after ZIP compression.
The latency (i.e. the time the user will wait before seeing anything) in the case of the transmission/visualization in uncompressed form is respectively 6.7 seconds for Venus (5.9s for transmission and 0.8s for loading the .OBJ file) and 18.5 seconds for Neptune (15.7s for transmission and 2.8s for loading the .OBJ file). If we consider the transmission of ZIP files, the latency is then 3s and 8.3s for Venus and Neptune respectively. In comparison, with our system, the user immediately (0.3s) starts to see coarse versions of the models. After 0.5s he visualizes the Neptune and Venus models with respectively 5% and 10% of elements (around 10K vertices), which constitute already very good approximations. Figure 8 and 9 illustrate the percentage of decoded elements in function of the time the user wait. The levels of details corresponding to 10% elements are illustrated. The curves corresponding to uncompressed representation and ZIP compression are also shown.
The accompanying video shows the live results for these two models.

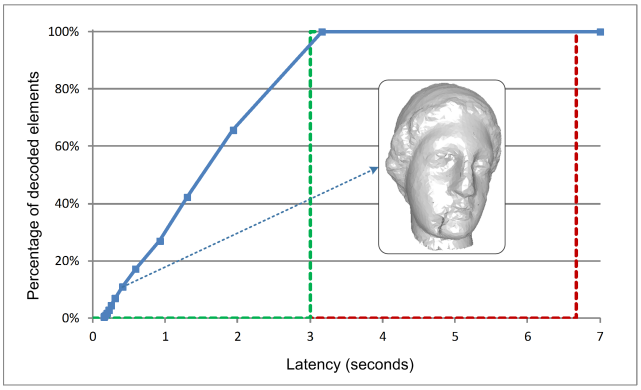


**Figure 8:** *Percentage of decoded elements according to the time latency starting from the selection of the Venus model on the web page. The dotted lines represent the scenarios of transmission/visualization in uncompressed ASCII form (red) and ZIP format (green).*

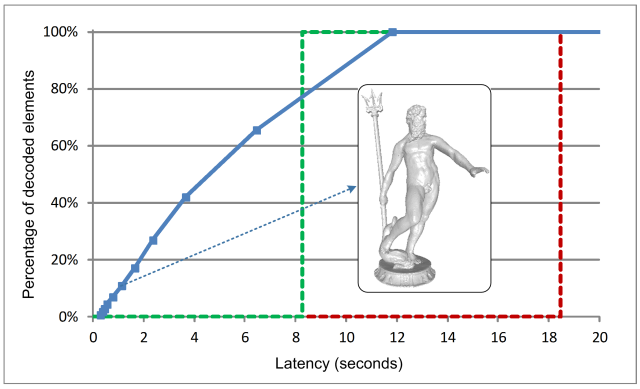


**Figure 9:** *Percentage of decoded elements according to the time latency starting from the selection of the Neptune model on the web page. The dotted lines represent the scenarios of transmission/visualization in uncompressed ASCII form (red) and ZIP format (green).*

#### 5.4.2 Comparison with concurrent state of the art

We have tested our web platform for the remote visualization of the Happy Buddha model (500K vertices) using a 5 Mbit/s Internet access (a typical 3G+ bandwidth). We have compared the results

against the Google webgl-loader (Happy Buddha available here[1]) and the X3DOM's image geometry approach (Happy Buddha available here[2]), under the same conditions (same PC, same bandwidth, same browser). Of course the servers are not the same, since we used the servers from the owners of the solutions. However the server owns a tiny influence on the results which mostly depend on the PC and bandwidth. Figure 10 illustrates some screenshots of the visualization for these three approaches, after respectively 800ms, 1.5s, 3s and 6s after launching the loading of the web pages. The live results are available in the accompanying video.
We can easily see the benefit of our streaming approach in this low bandwidth case. Indeed, after 800ms, we already have a coarse but illustrative model (1.5K vertices) which is then refined progressively: 16k vertices after 1.5s, 38K vertices after 3s and 92K vertices after 6s. On the other hand, for the single rate approaches, the user has to wait around 10s to see the whole 3D model. Our approach is particularly suited for medium and low bandwidth channels; indeed, in case of very high speed connections (50 Mbit/s), Google webgl-loader and X3DOM's approach are very efficient.

We have not tested our platform on a mobile device. However, the management of the levels of details is a very interesting feature for this kind of lightweight device. For instance, we could decide to interrupt the stream when a certain number of vertices are reached or when the frame-rate decreases under a threshold.

## 6 Conclusion

We have presented a technical solution for the remote streaming and visualization of compressed 3D content on the web. Our approach relies on a dedicated progressive compression algorithm, a new halfedge JavaScript structure and a fast and multi-thread JavaScript implementation of the streaming and decompression into levels of details. Our approach brings a clear gain in term of quality of user experience by removing the latency and providing very quickly a good approximation of the 3D model even for huge data.
Our approach is one of the first attempts to implement complex geometry processing operations directly in JavaScript; hence it provides useful insights on the benefits and limitations of this scripting language. JavaScript has shown unexpected impressive performances in our case. Of course, the main weakness of our approach is to make an intensive use of the CPU. We plan to investigate parallel decoding algorithms in order to overcome this limitation.

One remaining critical issue for the practical industrial use of web 3D data streaming is the question of the intellectual property protection. Indeed, during its transmission or visualization the 3D content can be duplicated and redistributed by a pirate. This issue can be resolved with the use of watermarking techniques. Such technique hides secret information in the functional part of the cover content (usually the geometry in case of 3D data). We plan to integrate such watermarking algorithm in the next version of our web platform however this algorithm has to be embedded within the compression and this constitutes a quite complex issue.

## Acknowledgment

We thank the anonymous reviewers for helping us to improve this paper. This work is supported by Lyon Science Transfert through the project *Web 3D Streaming*.

[1] http://webgl-loader.googlecode.com/svn/trunk/samples/happy/happy.html
[2] http://x3dom.org/x3dom/example/x3dom_imageGeometry.html

## References

ALLIEZ, P., AND DESBRUN, M. 2001. Progressive encoding for lossless transmission of 3D meshes. In *ACM Siggraph*, 198–205.

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 *ACM Web3D*, 17–26.

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 *ACM Siggraph Talks*.

CHEN, B., AND NISHITA, T. 2002. Multiresolution streaming mesh with shape preserving and QoS-like controlling. In *ACM Web3D*, 35–42.

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

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

GANDOIN, P.-M., AND DEVILLERS, O. 2002. Progressive lossless compression of arbitrary simplicial complexes. In *ACM Siggraph*, 372–379.

GOBBETTI, E., MARTON, F., RODRIGUEZ, M. B., GANOVELLI, F., AND DI BENEDETTO, M. 2012. Adaptive quad patches. In *ACM Web3D*, 9.

HOPPE, H. 1996. Progressive meshes. *ACM Siggraph*.

ISENBURG, M., AND SNOEYINK, J. 2003. Binary compression rates for ASCII formats. In *ACM Web3D*, 6–11.

JOMIER, J., JOURDAIN, S., AND MARION, C. 2011. Remote Visualization of Large Datasets with MIDAS and ParaViewWeb. In *ACM Web3D*.

JOURDAIN, S., FOREST, J., MOUTON, C., NOUAILHAS, B., MONIOT, G., KOLB, F., CHABRIDON, S., SIMATIC, M., ABID, Z., AND MALLET, L. 2008. ShareX3D, a scientific collaborative 3D viewer over HTTP. In *ACM Web3D*.

KETTNER, L. 1999. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry 13*, 21957, 65–90.

KHRONOS, 2009. WebGL - OpenGL ES 2.0 for the Web.

LAVOUÉ, G., TOLA, M., AND DUPONT, F. 2012. MEPP - 3D Mesh Processing Platform. In *International Conference on Computer Graphics Theory and Applications*.

LEE, H., LAVOUÉ, G., AND DUPONT, F. 2012. Rate-distortion optimization for progressive compression of 3D mesh with color attributes. *The Visual Computer 28*, 2 (May), 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 *ACM Web3D*.

MARVIE, J.-E., GAUTRON, P., LECOCQ, P., MOCQUARD, O., AND GÉRARD, F. 2011. Streaming and Synchronization of Multi-User Worlds Through HTTP/1.1. In *ACM Web3D*.

MOUTON, C., SONS, K., AND IAN GRIMSTEAD. 2011. Collaborative visualization: current systems and future trends. In *ACM Web3D*.

NIEBLING, F., AND KOPECKI, A. 2010. Collaborative steering and post-processing of simulations on HPC resources: Everyone, anytime, anywhere. In *ACM Web3D*.

PAJAROLA, R., AND ROSSIGNAC, J. 2000. Compressed progressive meshes. *IEEE Visualization and Computer Graphics 6*, 1, 79–93.

PENG, J., AND KUO, C.-C. J. 2005. Geometry-guided progressive lossless 3D mesh coding with octree (OT) decomposition. *ACM Transactions on Graphics (TOG) 24*, 3.

PENG, J., KIM, C.-S., AND KUO, C.-C. J. 2005. Technologies for 3D mesh compression: A survey. *Journal of Visual Communication and Image Representation 16*, 6, 688–733.

PENG, J., KUO, Y., ECKSTEIN, I., AND GOPI, M. 2010. Feature Oriented Progressive Lossless Mesh Coding. *Computer Graphics Forum 29*, 7, 2029–2038.

TAUBIN, G., AND ROSSIGNAC, J. 1998. Geometric compression through topological surgery. *ACM Transactions on Graphics 17*, 2, 84–115.

TIAN, D., AND ALREGIB, G. 2008. Batex3: Bit allocation for progressive transmission of textured 3-d models. *IEEE Transactions on Circuits and Systems for Video Technology 18*, 1, 23–35.

VALETTE, S., CHAINE, R., AND PROST, R. 2009. Progressive lossless mesh compression via incremental parametric refinement. *Computer Graphics Forum 28*, 5 (July), 1301–1310.

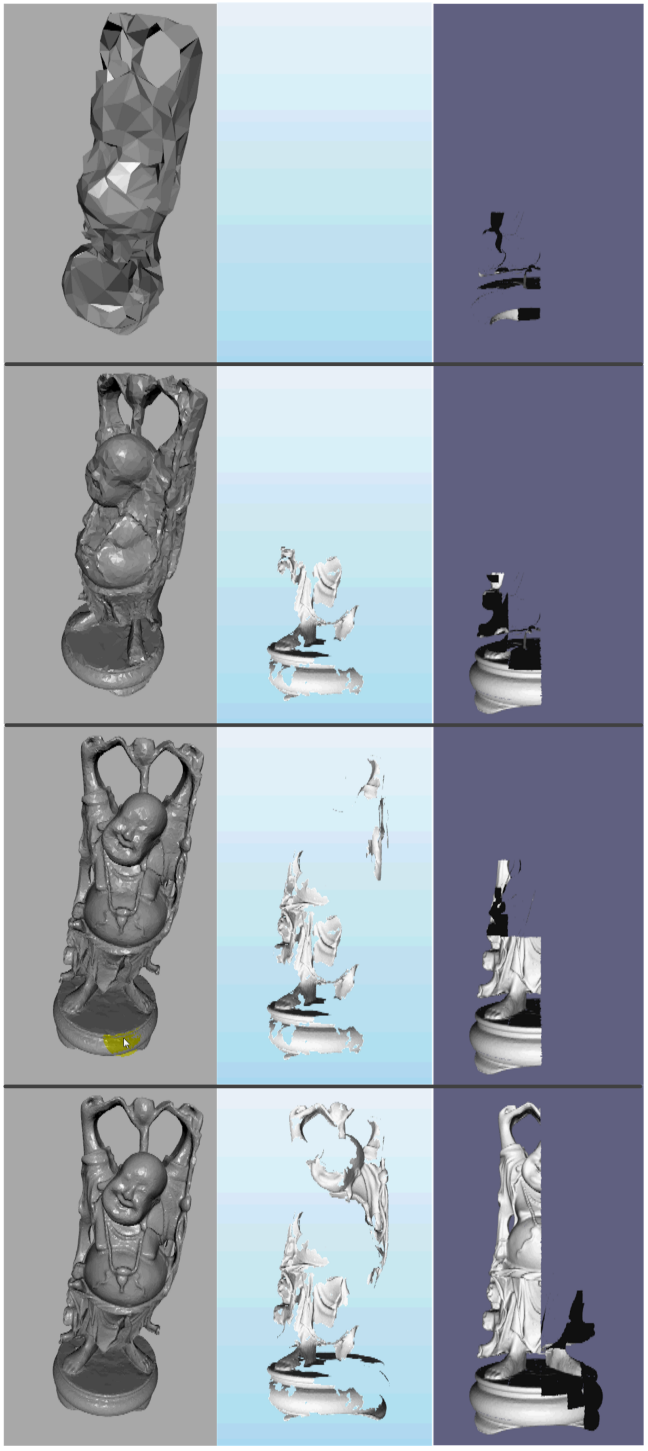

**Figure 10:** *Some screenshots illustrating the remote visualization of the Happy Buddha (1 million triangles) on a 5 Mbit/s Internet access, using our approach (left column), webgl-loader [Chun 2012] (middle column) and X3DOM's image geometry [Behr et al. 2012] (right column). From top to bottom, screenshots are taken respectively at 800ms, 1.5s, 3s and 6s after loading the web page.*