Multiresolution and fast decompression for optimal web-based rendering

Federico Ponchio[a], Matteo Dellepiane[a]

[a] *Visual Computing Lab, ISTI CNR, Pisa, Italy*

Abstract

Limited bandwidth is a strong constraint when efficient transmission of 3D data to Web clients and mobile applications is needed. In this paper we present a novel multi-resolution WebGL based rendering algorithm which combines progressive loading, view-dependent resolution and mesh compression, providing high frame rates and a decoding speed of million of triangles per second in JavaScript. The method is parallelizable and scalable to very large models.
The algorithm is based on the local multi-resolution approaches provided by the community, but ad-hoc solutions had to be studied and implemented to provide adequate performances. In particular, a compression mechanism that reached very high compression rate without impact on rendering performance was implemented. Moreover, the data partition strategy was modified in order to be able to load different types of data (i.e. point clouds) and better adapt to the potentials and limitations of web-based rendering.

Keywords: multiresolution, WebGL, 3D Web, web based 3D rendering, online 3D content deployment, mesh compression

1. Introduction

Limited bandwidth and increasing model sizes pose a challenge in the transmission of 3D data to Web clients and mobile applications. A possible approach is to compress the 3D model in order to minimize transmission time. Most of the research in this field has been focused on optimizing compression ratio.

Unfortunately, limited bandwidth often pairs with limited computational power, either because of JavaScript environment or low CPU power mobile devices, to the point that for most algorithms decoding time becomes the bottleneck even at moderately low bandwidth. Acceptable rates can be regained reducing compression ratio or using less sophisticate entropy compression algorithms.

A different approach makes use of progressive reconstruction algorithms, which improve the user experience by providing a simplified version of the model that refines while the remaining part of the model is being downloaded. The model converges very quickly at the beginning of the download, and only the details require the full model. However this class of algorithms performs even worse in terms of decoding time (as shown in Limper [**?**]) or in terms of compression ratio.

In the context of multi-resolution methods another desirable feature could be the view-dependent refinement. This allows to prioritize the download, decode a specific part of the model and vary resolution of the rendered geometry to maintain a constant screen resolution. This is obtained by maximizing quality at a given frame rate.

In addition to the above limitations, the 3D models that are now available on the web cover a much broader range of possibilities w.r.t. the past, including point clouds, *triangle soups*, topologically complex geometries, partially textured models. This leads to the necessity to propose a framework which could be robust enough to deal with different cases.

In this paper we present a novel multi-resolution WebGL based rendering algorithm (Figure 1) which combines *progressive loading*, *view-dependent resolution* and *mesh compression*, providing good rates and a decoding speed of million of triangles per second in JavaScript. Additionally, the method is *flexible*, since it is able to handle a variety of 3D formats, including textured models, non-manifold meshes, point clouds. Finally, it is also *scalable*, since it's able to to deal with very large models.

The method is based on a class of multiresolution structures [**? ?**] where the "primitive" of the multiresolution is a patch made of thousands of triangles. The original approach was rewritten to:

- obtain a more efficient data partition, and handle more data formats than triangulated surfaces
- compress the data structure in order to save disk space and bandwidth with no impact on performances
- extend the original algorithm to remote rendering

The paper is organized as follows: Section 2 provides an overview of related works. In Section 3 we describe the multiresolution structure, focusing on the improvements over the existing method, and on how the compression algorithm was designed to optimize decoding time while maintaining an adequate compression ratio. In Section 4 we compare it with existing web solutions for mesh compression and progressive visualization, and we analyze the performances when dealing with different classes of 3D models. The proposed method represents a solid alternative to current solutions, providing a practical mean to handle 3D models on the web.

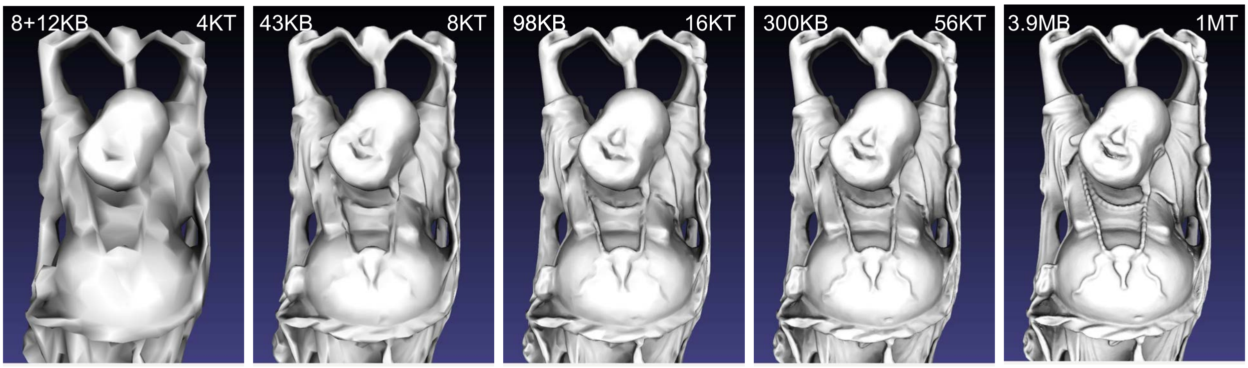

Figure 1: Progressive refinement of the Happy Buddha: on the upper left corner the size downloaded, on the upper right corner the number of triangles in the refined model. The header and index amount to 8KB

2. Related Work

This paper is related to several topics in the field of Computer Graphics. Among them: web-based 3D rendering, progressive and multiresolution rendering approaches, and fast decompression methods for 3D models.
While a complete overview of all these subjects goes well beyond the scope of the paper, in the next subsections we provide a short description of the state of the art, trying to focus on the aspects which are more related to the proposed approach.

2.1. Web-based 3D rendering

Three-dimensional content has always been considered as part of the multimedia family. Nevertheless, especially when talking about web visualization, its role with respect to images and videos has always been a minor one. Visualization of 3D components was initially devoted to external components, such as Java applets or ActiveX controls [**?**].

After some initial efforts for standardization [**? ?**], the proposal of WebGL standard [**?**], which is a mapping of OpenGL|ES 2.0 [**?**] specifications in JavaScript, brought a major change. Several actions related to the use of advanced 3D graphics has been proposed since then. For a general survey, please refer to the work by Evans [**?**]. Since the use of OpenGL commands needs advanced programming skills, there have been several actions to provide an ”interface” between them and the creation of web pages. We could subdivide the proposed systems between *declarative* approaches [**?**], like X3DOM [**?**] or XML3D [**?**], and *imperative* approaches, like Three.js [**?**], SpiderGL [**?**] and WebGLU [**?**]. The main difference between the groups is that the first ones rely on the concept of *scenegraph*, hence a scene has to be defined in all its elements, while the second ones provide a more direct interface with the basic commands. Other systems provide a sort of hybrid approach [**?**], where a simple scene has to be defined.
Evans [**?**] points out in his survey that declarative approaches had a major impact in the research community, while imperative approaches were mainly used in the programming community.
More in general, given the fact that the amount of data that needs to be sent to the webpage can be quite big, several efforts about a better organization of generic streamable formats [**? ?**] have been proposed. Nevertheless, when complex 3D data have to be streamed, these structures are not flexible enough to handle them.
In order to face this problem, in the last three years some progressive compression methods ad hoc for 3D streaming have been developed. Gobbetti et al. [**?**] proposed a quad-based multi-resolution format. Behr et al. [**?**] transmit different quantization levels of the geometry using a set of nested GPU-friendly buffers. Lavouè et al. [**?**] proposed an adaptation for the Web (reduced decompression time at the cost of a low compression ratio) of a previous progressive algorithm [**?**]. Other research has been also conducted to handle other types of data, like point clouds [**?**], which may present different types of issues to face with.
The rendering of textures or textured 3D models has been taken into account even before the standardization actions. In these cases the main issue is the amount of image data: standard techniques like mip-mapping can be adapted and improved both on the software and hardware side [**?**]. The issue of handling geometric data has been usually considered a minor one , due to the usual low complexity of 3D textured models [**?**]. Nevertheless, recently complex 3D models with texture coordinates are available from acquisition devices and technologies. Next subsections will provide further details.

2.2. Progressive and Multi-resolution methods

An important feature for user experience when rendering over slow connections or compressed models is progressiveness: the possibility to temporarily display an approximated version of the model and to refine it while downloading or processing the rest of the data.

The simplest (and widely used) strategy is to use a a discrete set of increasing resolution models (usually known as Level Of Detail, LOD). The main drawback with this approach is the abrupt change in detail each time a model is replaced.

A change of paradigm was brought by progressive meshes, introduced by Hoppe [**?**]. These meshes encode the sequence of operations of a edge collapse simplification algorithm. This sequence is traversed in reverse, so that each collapse becomes a split, and the mesh is refined until the original resolution. An advantage of progressive techniques is the much more smooth transition resolution changes, and the possibility to combine it with selective refining or view-dependent multiresolution, but this high granularity was achieved at the cost of low compression rates: about 37 bpv with 10 bit vertex quantization.

A large number of progressive techniques were later developed, but as noted in [**?**], Table 1, the research focus, however, was on rate-distortion performances and speed was mostly neglected. Latest algorithms still run below 200KTs in CPU.

Mobile and web application would be really too slow using these methods. As a compromise, pop buffers [**?**] propose a method to progressively transmit geometry and connectivity, while completely avoiding compression.

Another desirable feature, especially for large models, is view-dependent loading and visualization. Most multiresolution algorithms were made obsolete by the increased relative performances of GPU over CPU around the first years of 2000. It simply became inefficient to operate on the mesh at the level of the single triangle. Several works [**? ? ? ?**] achieved much better performances by increasing the granularity of the multiresolution to a few thousand triangles.

The main problem when increasing the granularity is ensuring boundary consistency between patches at different resolution: Yoon [**?**] and Sander [**?**] both employ a hierarchical spatial subdivision, but while the first simply disables simplification of most boundary edges, which results in scalability problems, the second relies on global, spatial GPU geomorphing to ensure that progressive meshes patch simplification is consistent between adjacent blocks. The works by Cignoni [**? ?**] rely instead on a non hierarchical volumetric subdivision and a boundary preserving patch simplification strategies that guarantee coherence between different resolutions, while at the same time ensuring that no boundary persists for more than one level. While not progressive in a strict sense, given current rendering speed, the density of triangles on screen is so high that popping effects are not noticeable.

An additional issue when dealing with view-dependent multiresolution techniques is the handling of textured models. While the encoding of texture coordinates can be easily taken into account when creating the patches of different resolution, the boundary consistency among them needs to take into account the texture images. Previous multi-resolution methods [**? ?**] proposed solutions for this, but they could fail when dealing with complex geometries.

Compression comes as a natural extension to this family of multiresolution algorithms: each patch can be compressed independently from the others as long as the boundary still matches with neighboring patches. A wavelet based compression was developed in [**?**] for terrains, a 1D Haar wavelet version in [**?**] for generic meshes on a mobile application. A comprehensive account of compression algorithms and the convergence with view-dependent rendering of large datasets can be found on a recent survey from Maglo et al.[**?**].

2.3. Fast Decompression of 3D models

Given that decompression speed is a key factor in order to be able to use compressed mesh, there's been surprisingly little effort by the community to provide solutions.

Gumhold and Straßer [**?**] developed a connectivity only compression algorithm that was able to decompress at 800KTs in 1998. Pajarola and Rossignac in [**?**], in 2000, reported 26KTs for a progressive compression algorithm, and developed a high-performance Huffman decoding identifying entropy compression as a possible bottleneck.

Finally, Isenburg and Gumhold in 2003 [**?**] developed a streaming approach to compression of gigantic meshes reaching an impressive decompression speed of 2MTs. The method accounts also for texture coordinates. A further work on this was proposed in 2005 [**?**]. We don't know of any following work specifically geared toward fast decompression. Even the recent survey by Maglo [**?**] shows that most of the effort is devoted to compression quality, but the performances of the works by Isenburg haven't been matched yet.

3. Method

Our multiresolution algorithm builds upon the methods described on [**? ?**], which are recapped in Section 3.1 for completeness. This Section provides a description of the novel method, by presenting an improved partition strategy, and a novel compression scheme (Section 3.2) tailored around the need for decompression speed, which is obtained using entropy encoding 3.3. Finally, the implementation for remote rendering (section 3.4) is presented.

3.1. Batched Multiresolution

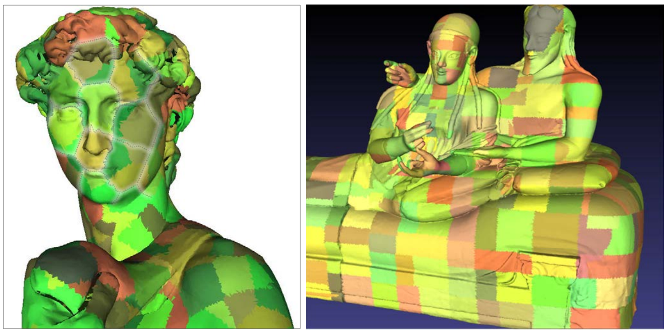

Figure 2: Left: volume partition by Cignoni [**?**]. Right: the volume partition obatined with our method.

In a multiresolution method, the model is split into a set of small meshes at different resolutions, obtained through a simplification process, that can be assembled to create a seamless mesh by simply traversing a tree which encodes the dependencies between each patch, using the estimated screen error to select the resolution needed in each part of the model. The screen space error is computed starting from the geometric error due to the simplification process, and computing the corresponding size in pixel when projected on screen.

For the simplification of the mesh we used the Quadric Edge Collapse method [?]. This simplification proved to be the most accurate and reliable, even though the simplification speed may be lower than other algorithms. Since partitions have to be created in a pre-processing stage, it was decided to use the slower alternative to obtain more accurate results. Moreover, quadric edge collapse can be easily extended to handle textured models (see later).

To build this collection of patches we need a sequence of non-hierarchical volume partitions (V-partition) of the the model; non hierarchical means essentially that no boundary is preserved between partitions at different levels of the hierarchy.

Cignoni et al [?] showed that any non-hierarchical sequence of volume partitions can be the base of a patch based multiresolution structure. Good partition strategies minimize boundaries, thus generating compact cells. In addition, they allow streaming construction and generate well balanced trees even when the distribution of the model triangles is very irregular. They used the Voronoi structure (see Figure 2, left), which is optimal for boundary minimization and balance. However, this partition is not suitable for streaming, leading to long processing times. On the other hand, the regular spatial subdivision used the previous version of the method [?] might generate unbalanced trees for very irregular models. This may impact on adaptivity.

We propose a different volume partition, defined by the leaves of a KD-tree built on the triangles of the model; to ensure the non hierarchical condition, the split ratio in the KD-tree alternates between 0.4 and 0.6 instead of the usual 0.5.
Figure 2 shows a typical partition provided by our method. The choice of this partition helps to better stream the model and provides better adaptivity. Additionally, the very regular shape of the patches may be useful when adding texture support (see later).

3.1.1. *Extension to textured models and point clouds*

The main challenge with multiresolution textured models lies in the simplification algorithm: it needs to take into account texture seams, and minimize deformation of the texture on the surface. We used the algorithm employed in Meshlab [?], which is an extension of Garland work [?] based on quadrics in 5 dimensions which include texture coordinates in selection of the collapse and the vertex optimization computation.

Enabling support for point clouds requires a few changes, merging into the same data structure, the functionality of Batched Multitriangulation [?] and those of Layered Point Clouds [?]: the simplification algorithm needs to be replaced with a point filtering approach, where half of the points are removed at each level. The octree structure of this data sampling perfectly fits in the compression and rendering paradigms explained in the next sections.

3.2. *Mesh Compression*

When porting multiresolution methods to the web, bandwidth becomes a further limitation. Hence, mesh compression becomes valuable not only to improve rendering performances, but also to reduce disk space occupancy.
However, in addition to the issue of fast decompression, our multiresolution algorithm imposes a set of constrains to mesh compression:

- each patch needs to be encoded independently from the others, so the method must be efficient and fast even on small meshes
- boundary vertices, replicated on neighboring patches, need to remain consistent through compression
- non manifold models must be supported

It would be possible to exploit the redundancy of the data due to the fact that the same surface is present in patches at different levels of resolution. We choose not to do so in order to keep the compression stage independent of the simplification algorithm used and to simplify parallel decompression of the patches. Otherwise, we would have to keep track of and enforce dependencies. In the next subsections, the strategies for the compression of the elements of the 3D models are shown.

3.2.1. *Connectivity compression*

The compression of connectivity can be difficult to handle, especially in the case of non manifold meshes. We modified the algorithm presented in [?], that supports non manifold meshes and surfaces with handles or holes.

The face-face topology for compression is computed as follows: we create an array containing three edges for each triangle, and sort it so that edges sharing the same vertices will be consecutive (independently of the order of the edges). The edges are then paired taking orientation into account, and all non paired edges are marked as boundary. Non manifold meshes will simply force the creation of some artificial boundaries.

The encoding process starts with a triangle and expands iteratively adding triangles. The processed region is always homeomorphic to a disk and if the region meets already considered triangles, we consider the common vertices as duplicated. The boundary of the already processed (encoded or decoded) region is stored as a doubly linked list of oriented edges (*active edges*), The list is actually implemented as an array for performances reasons. A queue keeps track and prioritize the *active edges*.

The first triangle adds three active edges to the list; iteratively an edge is extracted from the queue and, if not marked as processed, the following codes are emitted (see Figure 3):

SKIP if the edge is a boundary edge, or the adjacent triangle has already been encoded; the edge is marked as processed.

LEFT or **RIGHT** if the adjacent triangle shares two edges with the boundary; The two edges are marked as processed, a new edge added to the queue and its boundary adjacencies adjusted.

VERTEX if the adjacent triangle shares only one edge with the boundary, in this case the edges is marked as processed and two new edges added to the queue. If vertex of the new triangle opposing the edge was never encountered before its position is estimated using parallelogram prediction and the difference encoded, otherwise its index is encoded (in literature this case

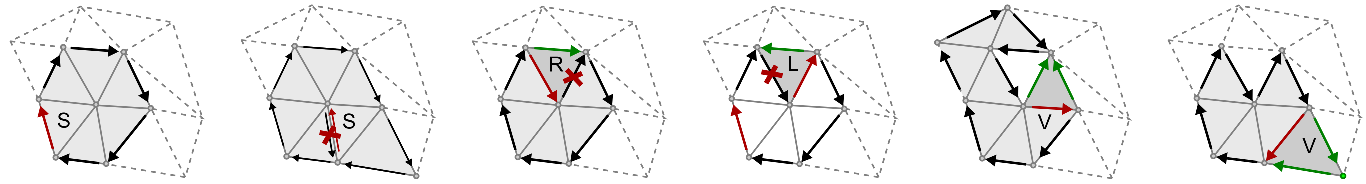

Figure 3: The four decompression codes: black arrows represent the front, the red arrow the current edge, in green the new edges added to the front.

is often referred as a “split”). This is a key difference with [?], where in the second case a SKIP code would be emitted, to keep the encoded region simple.

If the mesh is composed of several connected components, the process is restarted for each component.

The order in which the active edges are processed is important as we would like to minimize the number of VERTEX split operations, and generate a vertex-cache-friendly triangle order. To do so, we simply prioritize the right edges in the VERTEX operation, so that the encoding proceeds in ’spirals’. If the mesh is not homeomorphic to a disk, some split operations are required. This strategy reduces the number of splits to less than 1% in our examples, incurring in an average of 0.2 bpv cost.

This algorithm is certainly not optimal in term of bitrate, but it is extremely simple, linear in the number of triangles and robust to non-manifold meshes; as we will see in the results, speed is more important than bitrate.

3.2.2. Geometry and vertex attribute compression

To ensure consistency between boundary vertices of adjacent patches, we adopt a global quantization grid for coordinates, normals and colors. The global grid step for vertex position quantization is chosen automatically, based on the quadric errors during the simplification step in construction.

Geometry and vertex attributes are encoded as differences to a predicted value. The distribution of these values exhibit a bias which we can exploit to minimize the number of bits necessary to encode them. Our strategy is based on the assumption that most of the bias is concentrated on the position of highest bit (the log_2 of the value) of these value while the subsequent bits are mostly random. We simply store in an array, which is later entropy coded, the number of bits necessary to encode the value; the subsequent bits are stored in an uncompressed bitstream. In this way we need to decode a single symbol, from a limited alphabet, and read a few bits from a bitstream to decode a difference.

Each new vertex position, result of a VERTEX code, is estimated using a simple parallelogram predictor, and the differences with the actual position encoded as above. Color information is first converted into YCbCr color space and quantized, we encode the difference with one of the corner of the edge processed when emitting the VERTEX code. Normals vector are estimated using the decode mesh position and connectivity, and differences encoded as usual.

3.2.3. Texture coordinates and textures

Texture coordinates in the dataset are stored per vertex, replicating vertices on texture seams. They are compressed using the same parallelogram prediction algorithm employed for the vertex coordinates. Employing more sophisticated methods would drastically increase decoding time (see timings in Table 2 in [?]) or require additional linear algebra JavaScript libraries, for a limited decrease in bitrate.
Texture images are first mip-mapped to create different levels of detail, and then stored into the dataset as JPEG binaries. Textures are loaded on demand like the mesh patches, hence the mipmap level for rendering follows the same rules described in Section 3.4.

3.2.4. Point clouds

In the case of point clouds, the compression strategy for the vertex coordinates cannot rely on parallelogram prediction, in this case, after coordinate quantization we sort the points in z-order and store the differences between consecutive points using the same approach used for the meshes.

3.3. Entropy coding

Once that connectivity, geometry and attributes have been encoded into a stream of symbols and bits, the symbol stream is compressed following the biased probability distribution of the symbols.

Entropy decoding is the speed bottleneck in many mesh decompression methods, often due to the main goal of minimizing bit per vertex. Pajarola and Rossignac [?] developed a high-performance Huffman decoding algorithm in order to overcome this problem. The main advantage of this method is that it reduces the decoding phase to a couple of table lookups. Arithmetic coding, for example, outperforms Huffman in term of compression rate, but exhibits lower speed. A problem with this approach is the initialization time required to create the, possibly very large, decoding tables. It is then not suitable for decoding small meshes where the construction time would dominate over the decoding time.

Unlike Huffman and other variable-length codes, Tunstall code [?] maps a variable number of source symbols to a fixed number of bits. Since in decompression the input blocks consists of a fixed number of bits and the output is a variable number of symbols, Tunstall is slightly less efficient than Huffman, especially where the bit size of the input block is small. The decoding step is very similar to the high-performance Huffman algorithm, as it consists in a lookup table and a sequence of symbols for each entry, but the table size is only determined by the word size, and a fast method to generate it described in [**?**].

Given an entropic source of M symbols, to generate an optimal encoding table for a word size of N bits, we need to generate 2^N symbol sequences that have a frequency as close as possible to 2^{-N}, allows to encode every possible input (it is complete) and no sequence is a prefix of any other sequence (it is proper).

Tunstall optimal strategy starts with the M symbols as initial sequences, removes the most frequent sequence A and replaces it with M sequences concatenating A with every symbol until we reach 2^N sequences. The most time consuming step of the algorithm is to find the most probable sequence.

If we use a matrix where the first column contains the sorted symbol in order of probability, and at each step we replace the sequence with highest probability with M sequences adding a new column, we can observe that this table is sorted both in columns and rows (see Figure 4). This allows to select the next sequence by keeping each row in a queue and using a priority queue to keep track of which queue has the highest front element.

A 0.50	AA 0.25	BA 0.15	AAA 0.125	BAA 0.075
B 0.30	AB 0.15	BB 0.09	AAB 0.075	BAB 0.045
C 0.10	AC 0.05	BC 0.03	AAC 0.025	BAC 0.015
D 0.10	AD 0.05	BD 0.03	AAD 0.025	BAD 0.015

Figure 4: First four steps in construction of a Tunstall code with four symbols, the sequences A, B, AA, BA are replaced with a new column, beside each sequence, its probability is shown. In green the candidates for the next expansion.

To initialize the decoding table the symbol frequencies needs to be transmitted in advance.

Finally, an important advantage of variable-to-fixed coding is that the compressed stream is random accessible: decoding can start at any block. This makes it especially suited for parallel decompression in GPU. Unfortunately, current limitations in the capabilities of WebGL do not allow for such an implementation.

3.4. Remote view-dependent rendering

In the context of batched multiresolution approaches, the rendering requires the traversal of the patch tree, which is usually quite small since each patch is in the range of 8-32K vertices. An approximated screen space error in pixel is calculated by taking into account the view matrix, the bounding sphere of the patch, and the quadric error (or any other error metric) calculated during simplification.

At each step of the traversal, the selected nodes of the tree correspond to a complete representation of the model; each additional node included in the traversal increases locally the resolution of the model. The traversal is stopped whenever one of these four conditions is reached: a required patch is not available, **triangle budget** is reached, the **error target** is met, or there's no available **memory left**. The latter three parameters can be defined in advance, in order to deal with hardware resources. Once the traversal is terminated, the collection of selected patches is rendered as simple geometry. New patches are downloaded in order of screen error, to maximize the improvement in rendering quality of the model.

Since the rendering can start when the first patch is downloaded and the model is refined as soon as some patch is available, this is effectively a progressive visualization albeit with higher granularity. On the other hand, this structure is view dependent and thus able to cope with very large models, on the order of hundreds of millions of triangles.

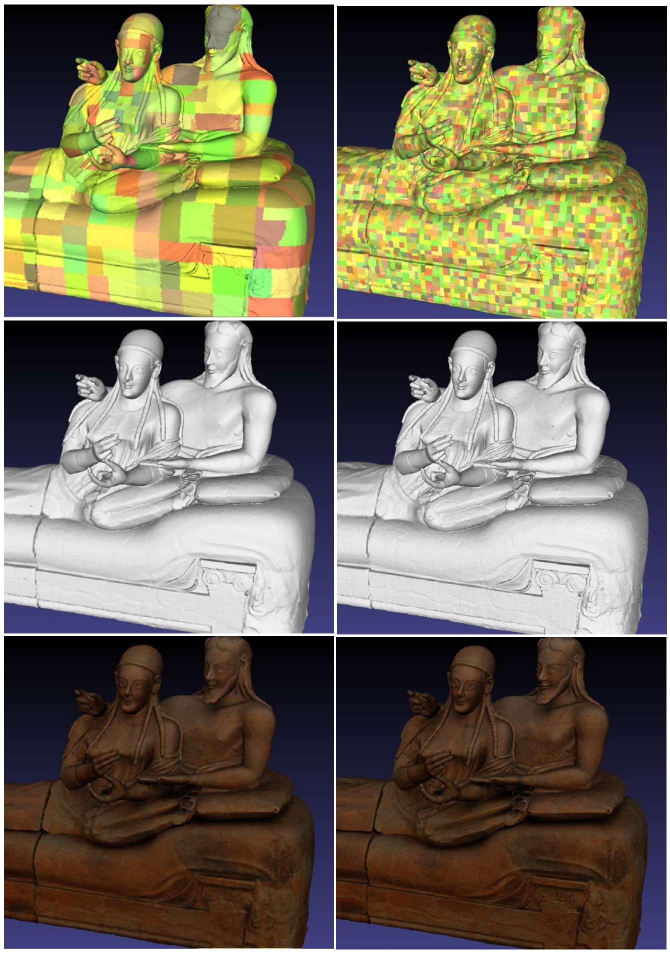

Figure 5: First column: before refinement. Second column: after refinement. From top to bottom: a visual representation of the geometric patches representing the model, the model with pure geometry, the model with color information.

The extension of the local multiresolution solution to remote rendering is quite straightforward. The data structure is composed of a fixed size header describing the attributes of the models, an small index which contains the tree structure of the patches and the position of each patch in the file, and the patches themselves. We use HTTP Range requests to download header and index, ArrayBuffers to parse this structures into JavaScript; the patches are then download prioritizing highest screen error.

Remote rendering is possible due to the WebGL framework: in this case, the implementation was integrated in 3DHop [?], a set of tools for web publication based on the SpiderGL [?] graphics library. The patches are encoded as binary files. Figure 5 shows an example of a model before and after view-dependent refinement.

4. Results

The proposed method has been tested on several cases, including very complex geometries. Additionally, whenever possible a comparison with existing systems was performed. A demo page, that shows the comparison and a few examples, is available at http://fastdec.duckdns.org (for reviewers only).

Our implementation has been successfully tested on major browsers on a variety of platform, from desktop machines to low end cell phones. The results we report here were measured on an iCore5 3.1Gh, using Chrome 41. Timings on other browsers (e.g.Firefox) where comparable. Regarding the multiresolution model construction, this is a preprocessing operation. Compression time is negligible, since it can be performed at about 1M triangles per second. The most cumbersome part is the quadric simplification algorithm, that runs at about 60K triangles per second per core. Nevertheless, the model construction must be performed only once.

The results section is organized as follows: the first part is dedicated to a qualitative comparison with available commercial or free solutions for remote rendering; the second part shows the results of the tests made on our compression scheme; the third part comments the performances of the system on a variety of examples.

4.1. Comparisons with existing systems

Several solutions for the visualization of 3D data with WebGL have been proposed in the last years. Most of them, unfortunately, do not deal with any strategy about compression and progressive visualization. Hence, they are able to deal only with smaller 3D entities.

Table 1 shows a comparison of the available systems. Also the issue of scalability (the capacity to handle very big models) is taken into account. The most similar and used systems are Potree, which is limited to point clouds, and Sketchfab, a widely used commercial tool, which uses a progressive visualization which is not view-dependent. Additionally, the scalability of the implementation of Sketchfab is unclear, and anyway the support for more complex models is currently limited not only for the "Basic" (50 Mb uncompressed size), but also for the "Pro" (200 Mb) and "Business" (500 Mb) accounts. Our method is able to handle models of any size (see later).

Hence, our method proves to be the most flexible one, since efficient solutions were developed to ensure not only performances, but also wide usability.

4.2. Entropy Compression: Comparison

Compression is an important issue when dealing with more complex models, and the need for remote rendering raises additional issues. Hence, we tested, both in C++ and JavaScript, compression rates and decompression speed of:

- our implementation of Tunstall coding (T)
- Huffman coding (H), in the high-performance version of Pajarola [?] (our implementation, C++ only)
- available implementations of LZMA
  in C++: http://www.7-zip.org/sdk.html
  and JavaScript: https://code.google.com/p/js-lzma/
- lz-string, a LZW based JavaScript implementation
  http://pieroxy.net/blog/pages/lz-string/index.html

The results are presented in Table 2, the lenght of 32K has been chosen since it is typical in our application.

Huffman and Tunstall are very similar in term of decompression speed, the difference is mainly in the time required to generate the decoding tables which are much larger for Huffman, especially when increasing the number of symbols. We tested also other probability distributions and found little difference in terms of speed. LZMA and LZW avoid this startup cost, however their more complex and adaptive dictionary management allows them to outperform Huffman and Tunstall in term of decompression speed only for very small runs (and very small dictionaries). In terms of compression ratio, Huffman and LZMA performed quite close to the theoretical minimum, while Tunstall was about 10% worse.

We did not implement Huffman in JavaScript, as we are confident the result would be very similar. On the other hand the numbers for LZMA change dramatically. Lz-string serves as a comparison, as a better library, optimized for JavaScript. The poor LZMA performances in JavaScript help explain the relatively slow performances of CTM in Limper [?].

4.3. Mesh Compression: Comparison

We used the Happy Budda model (in Figure 1), to compare compression ratio and decompression speed with OpenCTM (CTM) [?] Pop buffers (POP)[?], P3DW [?], WebGL-loader (CHUN) [?]. We compare our multiresolution (OUR) and, to test single resolution performances of our compression approach, a version (FLAT) which loads only the highest resolution level of the model. In each case the model has been quantized at 11 bit for coordinates and 8 bit for normals, and includes colors. The apparently better performances by WebGL-loader (CHUN) are explained by the fast that it is not a multi-resolution method, hence the whole model has to be downloaded before being able to decompress and visualize the model. This means that the compression rate of the highest resolution in our method (FLAT) is higher, and in any case WebGL-loader becomes unusable when more complex models are used.

Our decompression JavaScript implementation can decode about 1-3 million triangles per second with normals and colors in a single thread, on a desktop machine and 0.5 MT/s on a iPhone Five. Performances are somewhat degraded when the code is run during streaming visualization.

Table 1: Comparison of the main systems for web-based rendering.

	3D Meshes	3D Textured	Point Clouds	Streamable	Compressed	View-Dependent	Scalable
OpenCTM [**?**]	Yes	Yes	**No**	**No**	Yes	**No**	**No**
WebGL-loader [**?**]	Yes	Yes	**No**	Yes	Yes	**No**	**No**
Pop Buffers [**?**]	Yes	Yes	**No**	Yes	Yes	**No**	**No**
Potree [**?**]	**No**	**No**	Yes	Yes	Yes	Yes	Yes
SketchFab [**?**]	Yes	Yes	Yes	Yes	Yes	**No**	**(Yes)**
Our Method	Yes	Yes	Yes	Yes	Yes	Yes	Yes

	C++			JavaScript		
symbols	T	H	LZMA	T	LZMA	LZW
4	1058	520	1066	201	19	55
9	369	212	170	145	10	23
13	423	168	95	150	6	20
17	359	136	77	163	6	19
22	332	98	67	180	6	17

Table 2: Decompression speed in million of output symbols per second for Poisson distribution of 32K sequences

	FLAT	OUR	CTM	CHUN	POP	P3DW
MB	1.9	3.9	3.5	2.8	15	4.5
bpv	28	57	51	41	220	66
full	0.4	0.9	5.3	0.06	0.5	10

Table 3: Statistics for the Happy Buddha: model size in megabytes, bit per vertex and time in seconds required to fully decompress the model.

An important comparison is with the work by Rodriguez [**?**], which employs the same multiresolution batched strategy. For their mobile multiresolution application they report compression rates of 45-50 bpv on large colored meshes (which should be compared to our 28bpv). The difference is probably mostly due to the different connectivity encoding which, in their case, requires 20bpv against our 4 or 5bpv. It is difficult to compare the speed of the two decompression approaches since they run natively in C# on an iPhone4 while we run in JavaScript on the same platform. Our implementation speed is still, if a bit faster than their 50KTS [1], at about 60KTs. The difference is probably due their more sophisticate (and slow) arithmetic encoding.

C++ decompression speed is of course faster, reaching 9MTs, including colors and normals, and 16MTs for just position and connectivity. The speed reported in [**?**] of 35KTs for just the connectivity, as they mention, is due to the dynamic memory allocation in their implementation.

4.4. Compression of Textured models

In the case of textured models, the compression and quantization has to be applied not only on the geometric attributes, but also on the texture image.
Regarding the latest, using 13 bits quantization for a 4096x4096 pixel texture (which amount to half pixel precision) results in an hardly noticeable distortion (see Figure 6). With parallelogram prediction, texture coordinates are encoded in about 8 bpv (with a reduction of 48 to 1 respect to the standard 6 floats per face). We had to include the replicated vertices coordinates, that in our samples amounted to at most 15% of the total amount in models with many seams. Overall, adding texture coordinates increases the data size of about 25%, and decompression speed decreases accordingly.

We uploaded a few textured model to Sketchfab, for a comparison with a state of the art industrial solution: our multiresolution structure results on average 10% smaller, although it includes all the resolutions.

4.5. Streaming and Rendering

Loading the geometry through the Range HTTP request requires an increased number of HTTP calls: one for each patch, or 30-60 calls every million triangles. This does not really impact over performances: the overhead is quite small (about 400 bytes per call) and pipelining (the process of enqueueing requests and responses between browser and server) ensures full utilization of the available bandwidth. Random access is really necessary only to fully exploit the view-dependent characteristics of the multiresolution structure: the code could be easily modified to load the model with a single call if a higher number of HTTP calls is problematic on certain web hosting architectures.

In the demo page (http://fastdec.duckdns.org) it is possible to compare the performances of our method w.r.t. existing solutions in the case of a slow connection. Moreover, very complex geometries are also available for further testing. In the following, we show some example of the performances of the compression method in several types of models, in order to test its flexibility.

4.5.1. Point clouds

Point clouds are a quite common type of models, especially when large environments are taken into account. Terrestrial laser canners, but also UAV may provide dense point clouds. Ad-hoc solutions for encoding and rendering have been devised [**?**], but their implementation is usually very hard to be extended to triangulated surfaces.
On the contrary, the method proposed in this paper can be seamlessly applied also in point clouds: the data that are compressed and streamed are only the vertices attributes.

[1] The number is extrapolated from the decoding time of a large mesh given in their paper

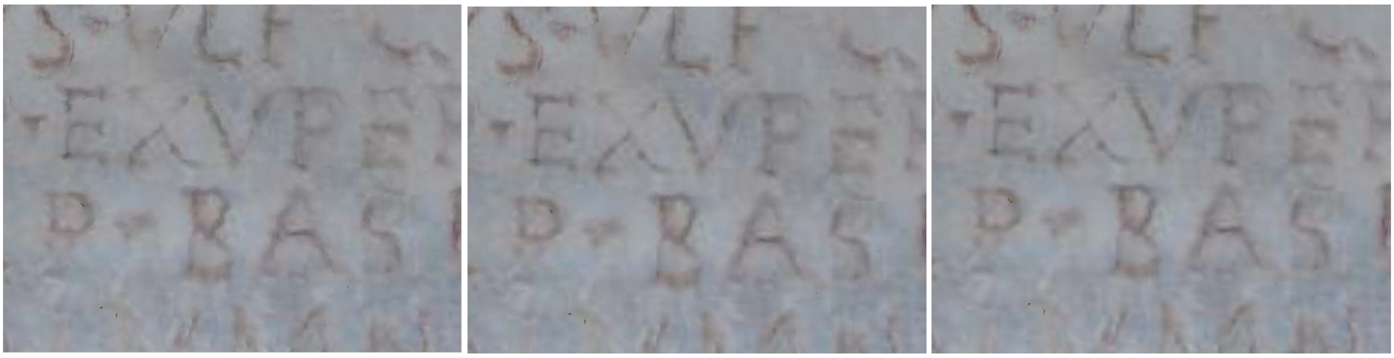

Figure 6: Example of textured model from left to right: texture coordinates quantized at 12 bits, at 13 bits and uncompressed. Notice the slight distorsion when precision is lower than half a pixel.

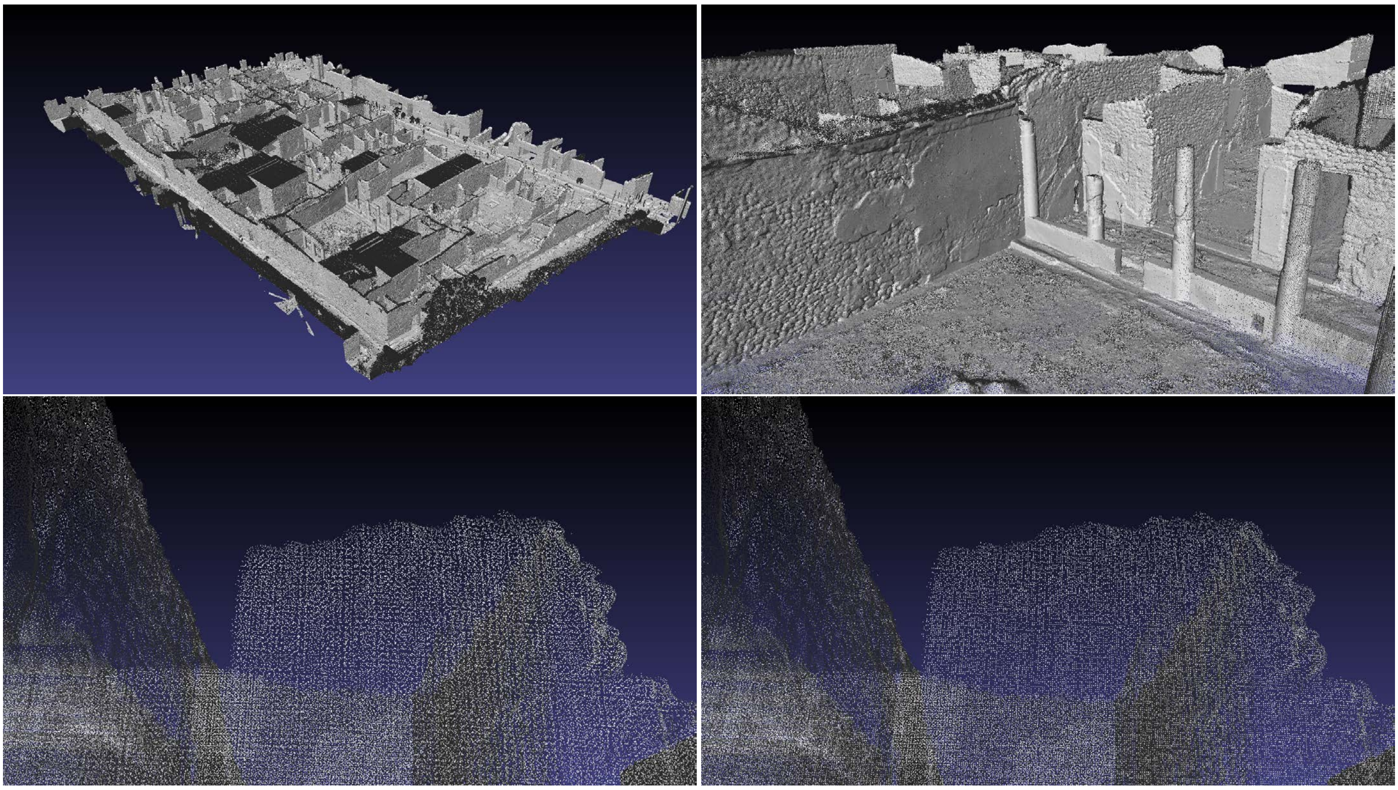

Figure 7: Pompei point cloud: original PLY file, 95M points, 2.26 Gb; uncompressed multires cloud, 1.68 Gb; compressed cloud, 326 Mb. Top left: the full compressed model, top right: a detail. Bottom left: a detail of the uncompressed point cloud . Bottom right: the same detail of the compressed point cloud

Figure 8: Big statue rendered in a browser: original PLY file, 84M triangles, 1.6GB; uncompressed model, 2.54 Gb; compressed model, 158 Mb. Top left: the full compressed model, top right: a detail. Bottom left: a detail of the uncompressed model . Bottom right: the same detail of the compressed model

Figure 7 shows an example of a point cloud of Insula V of Pompei, obtained with terrestrial laser scanning. The top part shows the compressed model and a point of view where all the details are visible. The bottom part of the Figure shows the difference between the compressed and uncompressed point clouds: the quantization of the original data brings to a reduction of points, since the quantization tends to "regularize" the points grid (in this case the quantization step was 0.5 mm). This effect is not noticeable in triangulated surfaces, but it may be an issue for point clouds. In this case, it's possible to change the quantization step in order to find the best tradeoff between compression and data quality.

4.5.2. Dense triangulated models

View-dependent progressive method have been especially devised to handle dense, triangulated 3D models. For this reason, the proposed method is able to provide optimal performances even when hundred millions triangles have to be taken into account. In the following, two examples of complex geometries are shown.

Figure 8 shows the 3D model of a 3-meter tall statue which was acquired with triangulation structured light scanner. The compressed model, which is nearly 10 % of the original PLY file, exhibits a detail that is undistinguishable from the uncompressed version.

In Figure 9 we show our system rendering the Portalada, a 180M triangles model at 30fps. The triangle budget has been fixed at 1M triangles and the streaming requires 2-3 seconds to reach full resolution on a good connection. The original model is 3.6GB, while the compressed multiresolution model is 838MB. The Figure also shows how the view-dependent paradigm is able to handle different resolutions of different parts of the model when peculiar points of view are shown.

4.5.3. Non-optimal, topologically complicated models

Some of the solutions proposed for progressive view dependent rendering proved to be limited since their basic assumptions on data processing didn't take into account that most of the more complex 3D models come from acquisition devices or techniques. This leads very often to the presence of geometric artifacts or unbalanced density.

Figure 10 shows two examples where the method deals with non-optimal geometries. On the left side, a model exhibiting strong topological artifacts. On the right side, a model with very unbalanced data density. In both cases, the method is able to deal with the issues and provide an accurate and reliable rendering.

5. Conclusions and possible improvements

The method proposed in this paper is a multi-resolution solution that provides remote view-dependent rendering of compressed 3D data. The main improvements w.r.t. current solutions are: the effectiveness in a wide range of bandwidth availability, computing power and rendering capabilities; the possibility to handle a wide variety of 3D models types, including very complex geometries; a mesh compression strategy that provides the best tradeoff between data compression and rendering performances.

Nevertheless, improvements in both compression and rendering performances can be obtained by further exploitation of the characteristics of some types of models.

For example, in the case of Point clouds, the rendering paradigm plays a key role to obtain a satisfying visualization. The current rendering method could be improved by implementing and extending existing approaches [**?**]. The attributes (i.e. radius) that could be used for efficient rendering can be easily inserted in the compression framework.

The compression and rendering of textured models can be further improved, by working on ways to better compress and handle textures, or moving to other texturing paradigms. An example could be the projective textures (a similar approach on point clouds was recently proposed by Arikan [**?**]), that could remove the need for parametrization, and open to even more complex datasets.

Acknowledgements

The research leading to these results was funded by EU FP7 project ICT Harvest4D (http://www.harvest4d.org/ , GA

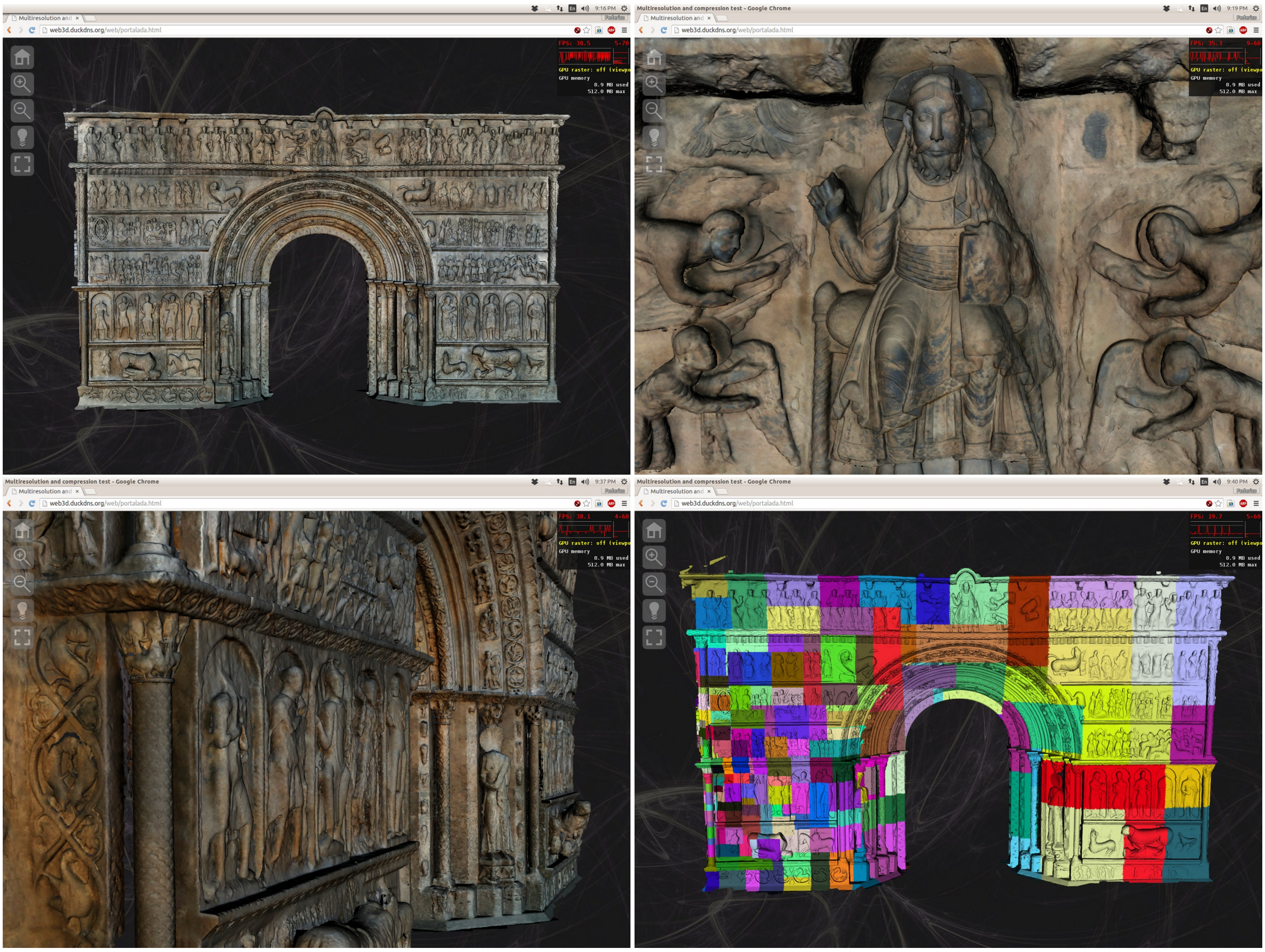

Figure 9: Portalada rendered in a browser: original PLY file, 180M triangles, GB; compressed model, 621 Mb. Top left: the full model, top right: a detail of the figure above the arch, bottom right: the resolution of the model as seen from the bottom left view point (without frustum culling)

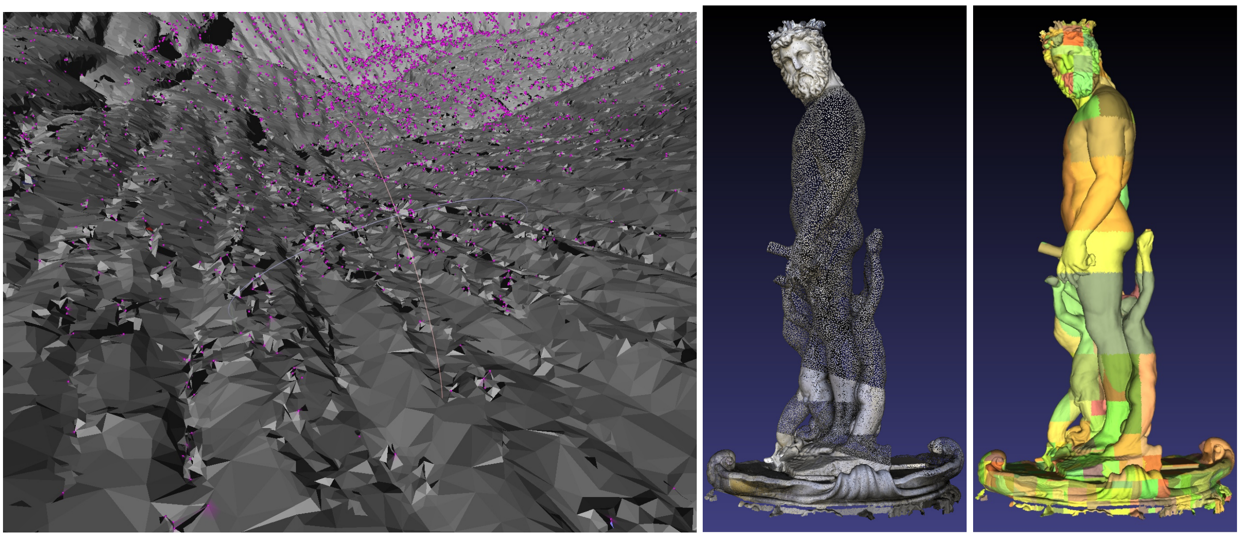

Figure 10: Left: a model with severe topological issues. Right: a model with very imbalanced vertex distribution

n. 323567) and EU INFRA Project Ariadne (GA n. 313193, http://www.ariadne-infrastructure.eu/).