Adaptive Quad Patches: an Adaptive Regular Structure for Web Distribution and Adaptive Rendering of 3D Models

Enrico Gobbetti[1] Fabio Marton[1] Marcos Balsa Rodriguez[1] Fabio Ganovelli[2] Marco Di Benedetto[2]

[1]CRS4, Italy [2]ISTI-CNR, Italy

Abstract

We introduce an approach for efficient distribution and adaptive rendering of 3D mesh models supporting a simple quad parameterization. Our method extends and combines recent results in geometric processing, real-time rendering, and web programming. In particular: we exploit recent results on surface reconstruction and isometric parametrization to transform point clouds into two-manifold meshes whose parametrization domain is a small collection of 2D square regions; we encode the resulting parameterized meshes into a very compact multiresolution structures composed of variable resolution quad patches whose geometry and texture is stored in a tightly packed texture atlas; we adaptively stream and render variable resolution shape representations using a GPU-accelerated adaptive tessellation algorithm with negligible CPU overhead. Real-time performance is achieved on portable GPU platforms using OpenGL, as well as on exploiting emerging web-based environments based on WebGL. Promising applications of the technology range from the automatic creation of rapidly renderable objects for games to the set-up of browsable 3D models repositories in the web that will be accessible by upcoming generation of WebGL-enabled web browers.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/Network Graphics I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Display Algorithms

Keywords: distributed rendering, quad meshing, subdivision

1 Introduction

As it already happened with photography and audio/video, the creation of 3D content is becoming more and more affordable in terms of time, user skills and, consequently, economic investment. In the last few years, 3D scanning systems have become commodity components. At the same time, the rapid evolution and proliferation of low-cost graphics hardware has made advanced 3D modeling available to a variety of user. As content becomes easier to create and cheaper to host, more companies and individuals are building virtual worlds (e.g., Second Life hosts 270 terabytes of user-generated content in 2009 [Lab 2009], and this is growing by approximately 100% every year).

With the increasingly widespread introduction of mobile terminals and WebGL enabled browsers, 3D graphics over the Internet is expected to attract a lot of additional attention. Still, unlike what has happened for standard media, which have converged high quality compressed formats specifically designed for storage and streaming, essentially based on the same small set of concepts, distributing and rendering non-trivial 3D models, especially on low-cost or mobile platforms, is still challenging. Detailed 3D models are heavy, non-trivial to render, and are experienced in a highly non-linear interactive way. These characteristics impose fast incremental loading and reasonable compression, GPU accelerated rendering methods, and adaptive view-dependent culling techniques. While a lot of generic solutions have been presented for general "desktop" platforms [Yoon et al. 2008], there is now an increasing interests for techniques tuned for lightweight, interpreted, and scripted environments. The limitations of such platforms imposes additional constraints on the 3D streaming formats, which should be based as much as possible upon preexisting components in order to avoid the overhead of coding complex decoders in non-optimized programming environment, such as JavaScript.

Contribution. We introduce a remote rendering approach in which a large class of textured geometric models are converted into compact multiresolution representations suitable for storage, distribution, and real-time rendering on modern commodity/web platforms. Our method extends and combines recent results in geometric processing, real-time rendering, and web programming. In particular: we exploit recent results on surface reconstruction and isometric parametrization to transform the point cloud into a two-manifold mesh whose parametrization domain is a small collection of 2D square regions; we encode the resulting parameterized mesh into a very compact multiresolution structure composed of variable resolution quad patches whose geometry and texture is stored in a tightly packed texture atlas; we adaptively stream and render variable resolution shape representations using a GPU-accelerated adaptive tessellation algorithm with negligible CPU overhead. Real-time performance is achieved on portable GPU platforms using OpenGL, as well as on exploiting emerging web-based environments based on WebGL. Although not all the techniques presented here are novel in themselves, their elaboration and combination in a single system is non-trivial and represents a substantial enhancement to the state-of-the-art.

Advantages. The pipeline is fully automatic and targets densely tessellated models, such as those created by 3D scanning or modeling systems such as ZBrush. Our approach bridges the gap that currently exists from general-purpose meshes to rendering oriented structures based on real-time tessellation with normal/bump maps, which are typical of modern gaming platform but currently require considerable human effort to create. The simplicity of a regularly remeshed representation has many benefits. In particular it reduces random memory accesses and eliminates the indirection and storage of per triangle vertex indices and per vertex texture coordinates. The resulting representation is compact, can be built on top of existing image representations, and is very well suited to streaming. Due to the negligible run-time CPU overhead, real-time performance is achieved both on conventional GPU platforms using OpenGL, and on the emerging web-based environments based on WebGL.

Promising applications of the technology range, thus, from the automatic creation of rapidly renderable objects for local and online games to the set-up of browsable 3D models repositories in the web.

Limitations. The proposed method is not general purpose, but targets meshes defininig closed objects with large components (i.e., typical solid objects without fine topological details). As for other compressed streamable formats, we do not strive to exactly replicate the original geometry and color, but only to visually approximate them in a faithful way. As a result, and similarly to compressed video/image formats, our representation is lossy, and thus not applicable in situations where precise measures of the original geometry are required (e.g., CAD systems).

Despite these limitations, as demonstrated by our results, the method provides good scalability in the distribution of compact streamable and renderable 3D representations of objects.

2 Related Work

In the following, we will briefly discuss the approaches that are most closely related with our work. Readers may refer to well established surveys [Yoon et al. 2008] for further details.

Compact mesh models for distribution and rendering. Much of the work in mesh distribution has focused on compression rather than adaptive view-dependent streaming. MPEG-4 is a reference work in the area [Jovanova et al. 2008]. Early methods for view-dependent LOD and progressive streaming over arbitrary meshes use fine grained updates based on edge collapses or vertex clustering [Xia and Varshney 1996; Hoppe 1997; Luebke and Erikson 1997]. These methods are the basis upon which many compression and streaming formats for the web have been built [Maglo et al. 2010; Blume et al. 2011; Niebling et al. 2010]. These approaches are however CPU bound and spend a great deal of rendering time to compute the view-dependent triangulation prior to rendering, making their implementation in a scripting language particularly challenging. As the GPU has become more powerful, more recent methods typically either reduce the per-primitive workload by composing at run-time pre-assembled optimized surface patches [Cignoni et al. 2004; Yoon et al. 2004; Borgeat et al. 2005; Gobbetti and Marton 2004a; Gobbetti and Marton 2004b] or introduce techniques for performing view-dependent refinement within geometry shaders [Hu et al. 2010]. These methods proved very effective in terms of rendering speed, but still require coding of non-trivial data structures and techniques for decompression, leading to potential problems in a script-based web implementation. We therefore employ a solution that encodes much of the shape and appearance of a model into a texture. This is also the goal of *geometry images* [Gu et al. 2002; Sander et al. 2003], which enable the powerful GPU rasterization architecture to process geometry in addition to images, and the networking component to rely on already existing and optimized libraries for compression and streaming of images. Geometry images focus on reparametrizations of meshes onto regular grids, while we focus on developing a specific multiresolution structure on top of a reparameterized model. Our quad-based parametrization leads in addition to a tighter texture packing and a simple handling of chart boundaries.

Parameterization and remeshing. Representing complex two-manifold models as a collection of quads requires a parametrization of input models (refer to [Sheffer et al. 2006] for a survey). The simplest approach is single-disk parameterization [Floater and Hormann 2005], which, however, can be applied only to genus-0 meshes and leads to high distortions unless the mesh has almost zero Gaussian curvature everywhere. In this work, we take the approach of using a base mesh to parameterize the model [Pietroni et al. 2010]. While in these base meshes approach the 3D parametrization domain is often based on triangles [Lee et al. 1998; Praun and Hoppe 2003; Khodakovsky et al. 2003; Schreiner et al. 2004; Kraevoy and Sheffer 2004], there are clear advantages in our case in adopting a quad-based domain, since it provides tight packing and simple handling of chart boundaries.

Details and Adaptive mesh refinement on GPU. An approach to the problem of rendering generalized displacement mapped surfaces by GPU raycasting was proposed in [Oliveira et al. 2000; Wang et al. 2003; Wang et al. 2004]. Other generalizations involve replacing the orthogonal displacement with inverse perspective [Baboud and Décoret 2006], replacing the texture plane with a quadric [Manuel M.Oliveira 2005], handling self shadowing in general meshes [Policarpo et al. 2005]. The evolution of graphics hardware has allowed many surface tessellation approaches to migrate to the GPU, including subdivison surfaces [Shiue et al. 2005], NURBS patches [Guthe et al. 2005], constrained urban models [Cignoni et al. 2007], and procedural detail [Boubekeur and Schlick 2005; Boubekeur and Schlick 2008]. This makes it possible to generate geometric details directly in the vertex shader. We adapt semi-uniform adaptive patch tessellation [Dyken et al. 2009] to handle quad patches with textured detail. Whereas previous approaches are typically used to amplify coarse geometry, our end-to-end framework is designed to faithfully reproduce a resampled high-resolution model.

3 Pipe-line Overview

Our contribution is an unattended software pipeline for automatically converting a large variety of textured geometric models into compact multiresolution representations suitable for storage, distribution, and real-time rendering on modern commodity/web platforms. Fig. 1 illustrates the main components of our pipeline.

The pipeline takes as input a dense point sampling of the original model. This kind of sampled representation can be created from a large variety of models - point clouds, meshes, or parametric objects. A two-manifold triangular mesh is first fit to the point cloud using a surface reconstruction and topology cleaning step, and the resulting two-manifold mesh is parameterized (see Sec. 4). Our parameterization domain D consists in a small collection of almost isometric square patches. Since each of these patches can be sampled on a grid with lines parallel to its sides, storing 3D positions, normals and colors of the associated point on the mesh in a $N \times N$ square patch, the overall shape representation consists of M square patches of $N \times N$ samples. This regular structure is then encoded into a compact multiresolution structure composed of variable resolution quad patches assembled in 2D images. Geometry and texture are stored in a tightly packed multiresolution texture atlas, which can be streamed over the network for generating variable resolution shape representations using a GPU-accelerated adaptive tessellation algorithm (see Sec. 5). The resulting rendering subsystem has negligible CPU overhead and is heavily built on top of consolidated 2D image representations. It can thus be efficiently implemented both on conventional commodity platforms and on the newly emerging scripting platforms for the web. The various steps of our pipeline are detailed in the following sections.

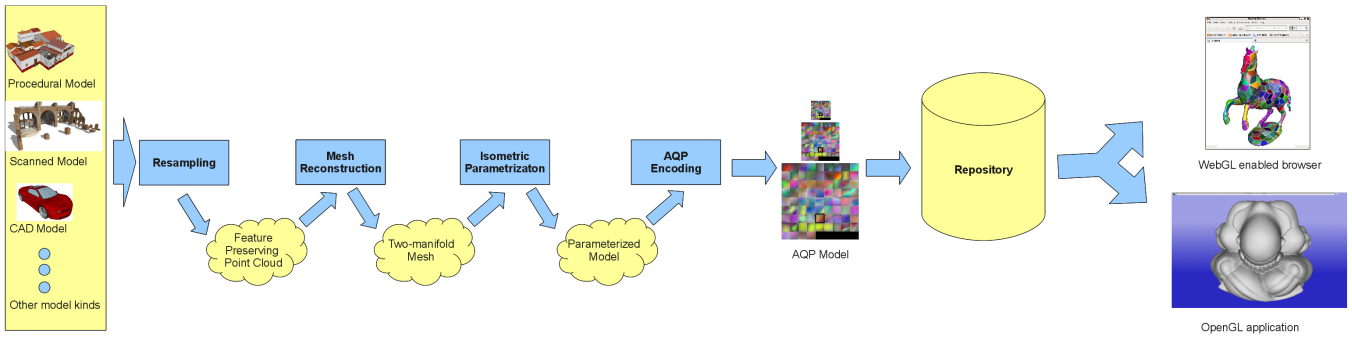


Figure 1: ***The AQP pipeline.*** *We take as input renderable models and generate compact adaptive streamable representations.*

4 Surface Reconstruction, Parametrization, and Quad Remeshing

We assume that the input to our pipeline is a point cloud or a tessellated mesh, possibly with artifacts such as non manifoldness.

The first phase of the method transforms the input in a clean manifold triangle mesh. As a first step, we use Poisson reconstruction [Kazhdan et al. 2006] to obtain a manifold and watertight version of the input mesh, which is saved in a streaming format. We then perform a single streaming pass over the generated triangle mesh, and discard from it the connected components with less than a prescribed number of triangles, to remove topological noise. It should be noted that these reconstruction and filtering steps may not be necessary if the input mesh is already two-manifold, or it may be replaced with other reconstruction or topological repair techniques.

The second phase consists of parameterizing the mesh on a simple quad-based domain. We first construct an almost isometric triangle mesh parameterization through abstract domains [Pietroni et al. 2010] (see Fig. 2.(b)), which maps the original mesh to a simplified parametrization domain made of equilateral triangles. The method works by applying local simplification operations to the input mesh, and remapping the triangles of the original region onto those of the simplified region. By iterating the simplification and remapping process, the algorithm ends with a small parametric domain consisting of a simple triangle mesh. This domain is in turn remapped into a collection of 2D square regions by adding a vertex in the barycenter of each triangle and building a quad for each edge (see Fig. 2.(c)). The produced parameterization exhibits very low isometric distortion, because it is globally optimized to preserve both areas and angles. In order to manage larger models than those handled by the original method [Pietroni et al. 2010], we have heavily reduced memory usage by employing a multiple-choice approach instead of a global queue to select the edge collapses during the simplification phase [Wu and Kobbelt 2002].

Once we have obtained the quad-based parametrization of the input model, we resample each quad, taking the samples from the original geometry. The sampling phase, which works on the point cloud representation used as input for the reconstruction step, associates to each quad a regular grid of samples (position, color, normals). This final collection of regular grids (see Fig. 2.(d)) is used as input for the multiresolution structure creation phase.

5 Quad-based Multiresolution Structure

The previous steps of the pipeline are able to produce a parametrized mesh made of a set of equally sized quad patches, each of them composed of $w \times w$ samples. Vertex, color and normal information are available for each sample.

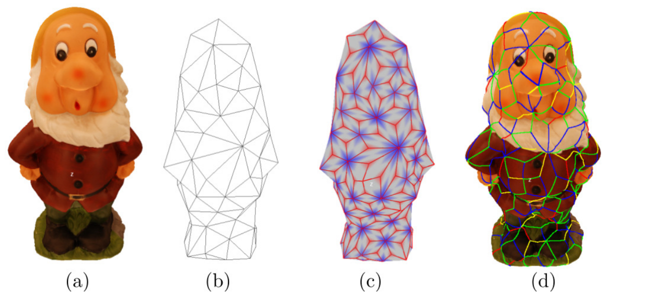


Figure 2: ***Reconstruction steps.*** *The original model (a) is parameterized on a simple quasi isometric triangulation (b), which is in turn remapped to a collection of 2D square regions (c), used as a basis for resampling the model (d).*

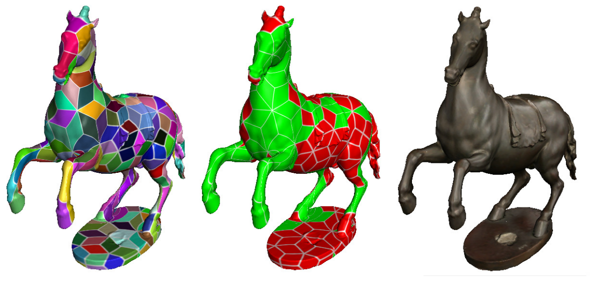


Figure 3: ***Rampant model.*** *Example of rendering with patch color, level color and with original color.*

In order to achieve adaptivity, we encode the resulting parametrized mesh into a very compact multiresolution structure composed of a collection of variable resolution quad patches, whose geometry and texture is stored in a tightly packed texture atlas. At run time, we exploit this structure to rapidly distribute and generate seamless view-dependent multiresolution visualizations. These view-dependent representations are constructed by adaptively loading and combining patches at different resolutions, depending on viewing parameters. Surface continuity is guaranteed by carefully handling patch boundaries (see Fig. 3).

5.1 Quad Structure

The main advantage of our quad parametrization is that a complex surface can be compactly and efficiently by storing geometry in a tightly packed texture atlas. Since all quads have the same size, packing is trivial and very efficient.

Each image quad represents a square surface patch, and is made independent from the others by replicating in it the boundary vertices. The patch triangulation is implicit. A surface is thus represented by a 2D texture that contains a number of patches, arranged in a 2D grid. Because of the GPU maximum texture size limitation (generally $4K$ or $8K$), a single large model can be split into a number of texture pyramids, each of them with the maximum resolution lower than the limit. A multi-resolution representation is constructed from the high-resolution representation by building a texture mip-map through a filtering operation (see Fig. 4(a)). Each half resolution representation can be constructed by a simple average filter, with special care taken only for properly handling samples at quad boundaries (see Sec. 5.2). Inside a single pyramid, square patches are simply organized as a square 2D matrix of $N \times M$ patches (see Fig. 4(a)).

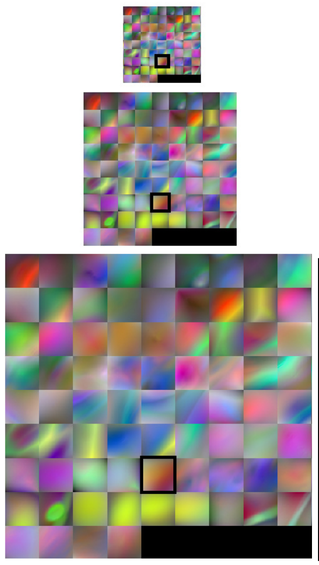
The geometry mipmap is enriched by parallel color and a normal mipmaps, which are based on the same concept of quad patch simplification. These two mipmaps are not constrained in our system to have the same resolution of the geometry. In general, they will have higher resolution with respect to the geometry one, allowing us to achieve the same effect of surface texturing with color and normal maps (which typically are at higher resolution than the geometry).

5.2 Preprocessing

For each pyramid we build three mipmap hierarchies: a geometry, a color and a normal mipmap. Processing of the three structures share some aspects: they start sampling the input dataset on a patch basis, and then build inner mipmap level, with a patch filter approach that maintains continuity among boundary samples of adjacent patches. Inner patch samples are simply averaged from 4 children samples, instead boundary samples are averaged without taking into consideration the 2 samples which do not belong to the boundary. The corner samples which are shared among 4 adjacent patches, are filtered with pure sub-sampling for the same reason, see figure 4(b). Operating without this special care would produce corresponding boundary samples with different contribution for adjacent patches, thus loosing continuity.

We exploit the geometry patch structure to encode the positions as a map of 3D displacements with respect to the bilinear interpolation of the patch corners at the corresponding u,v parametric coordinates. The corners of all the square patches are stored quantized at 16 bits per component, in a root file, which would correspond to the coarsest level of our multiresolution structure. All other levels, which represent displacements with increasing resolution with respect to the root, are stored quantized at 8 bits per component. Instead of using a global, per level, uniform quantization range, which would introduce too many discretization artifacts, we decided to modulate the quantization range per generated vertex.

In a first step, quad quantization ranges are computed for each patch, by taking the minimum and the maximum differences between positions inside the patch and predictions obtained through bilinear interpolation of corner positions. In order to avoid discontinuities caused by different per-patch quantization, we move quantization information to the patch corners. We thus determine for each patch corner the minimum of all the adjacent quad minimum values, and the maximum of all the maximum values. Using these corner values, the quantization range for a sample of a patch at parametric coordinates u,v is given by the bilinear interpolation at u,v of all the for corner quantization factors. This way, quantization on edges depends only by the two corners defining the edge, and thus is shared among the two adjacent patches, solving the quantization continuity problem.


(a) Geometry Mipmap

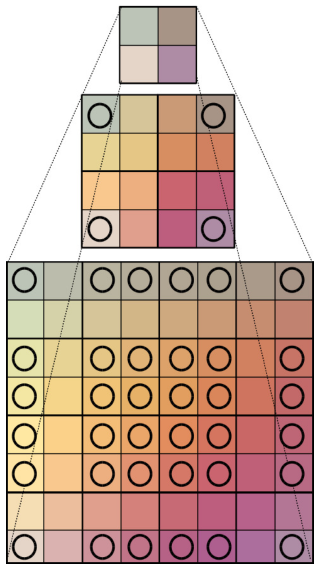

(b) Patch filtering

Figure 4: ***Multiresolution structure.*** *In figure 4(a) there is an example of three levels of a geometry mipmap with* 76 *patches on a grid* 9×9*, with highlighted the filtered patch of figure 4(b). Figure 4(b) shows three levels of a quad patch. Circles inside quads shows which samples contribute to the generation of the parent sample: one for the corner (sub-sampling), two for edges, and four for inner sample. Upper level correspond to the root patch representation with only the four corners.*

Geometry is finally stored using PNG compression to avoid to introduce further artifacts due to possibly uncontrolled lossy compression. For colors and normals, instead, we can choose between storing them as PNG files or using DXT1 (for colors) and DXT5 (for normals) compression. Using these hardware-supported compressed formats is only possible when using our OpenGL renderer, since WebGL currently lacks support for them. Data is stored in separate files: each mipmap is subdivided by levels, and then the level is split into tiles if its width is bigger than a predefined value, 512 samples in the current implementation. This approach is useful to avoid to require a complete level at a single time, which surely would be too big for the finest level of details. Tile width is a multiple of the patch finest width to avoid to split a patch into separate files.

5.3 Adaptive seamless rendering

We adaptively stream and render variable resolution shape representations using a GPU-accelerated adaptive tessellation algorithm with negligible CPU overhead. Seamless rendering is substantially performed by the GPU through a vertex/fragment shader pair, leaving to the CPU only the tasks of selecting the proper level of details for each patch, and of querying missing data from a server. Adaptive tessellation of a coarse mesh could be done exploiting the geometry shader, but this GPU stage cannot output more than a certain number of primitives, (1024 in the original specification) thus limiting the subdivision levels. We preferred to use the instancing approach, creating during the initialization a small number of subdivision regular grids, containing the (u, v) parametric coordinates of the vertices and an index telling where the vertex re-

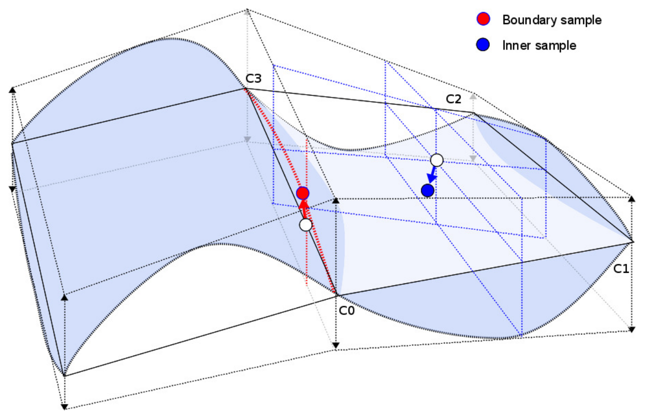


Figure 5: ***Seamless point dequantization.*** *Vertices on the boundary of the two adjacent patches, like the red one, share the same dequantization values derived by the linear interpolation of the same two corners $C0, C3$. Inner vertices quantization min, max are derived from the bilinear interpolation of the 4 corners min,max values. White circles show corners interpolations on the patch. Vertical arrows shows the corner min,max ranges, used for dequantization.*

sides (inside or on the boundary) relatively to the patch. We use $K = log2(w) - 1$ vertex buffer objects, (being w the linear size of a patch at maximum resolution) to tessellate the patches from a size of 4 linear samples to the maximum size w, with resolutions doubled for each level. Let's say that root is at level 0, first patch at level 1, and finest patch at level K. The renderer preallocates for each pyramid three texture mipmaps (geometry, color and normals) initialized only with the root data and which will contain the patches at various resolutions, once they will be available. At each frame the renderer selects the proper level of detail for each patch, if it is not available chooses the finest available level for it and posts a request for the tile containing the desired data (see Sec. 5.4). To produce a continuous representation patches must match perfectly along the edges, so they must have the same level of resolution for each edge. The patch LOD evaluation first computes the desired LOD for each edge of the quad by projecting it to the screen and comparing it with the desired screen tolerance. Edge LOD cannot be higher than the minimum of the two finest level of available data of the two adjacent patches along this edge. The quad patch LOD is set to the maximum (finest) of the 4 edge LODs. A texture is filled at each frame with the 4 edge LODs for each quad.

In the draw procedure for each patch the tessellation corresponding to the selected quad LOD is drawn with a proper vertex/fragment shader pair. The tessellation vertices are triple with (u, v, e) where e represent the edge to which belongs that vertex $(0, 1, 2, 3)$ or 4 for inner vertices. The vertex shader convert the (u, v) and quad patch id to the corresponding coordinates in the texture mipmap, where it can fetch the geometry displacement. When a vertex belongs to the inner part is simply a matter of scaling and translating (u, v) to fetch proper data. Instead when detecting edge vertices we need to handle them properly before fetching data, to be able to stitch together adjacent patches. Edge LOD is always coarser or equal than patch LOD. To get a seamless representation we snap boundary patch vertex parametric coordinates (u, v) at the edge resolution, which is the same for adjacent patches also if their quad LOD is different. The snap procedure identifies the edge id from the third component of the vertex and read the corresponding LOD value from the edge LOD texture at (quad, edge id) texture coordinates. Then snaps the vertex (u, v) parametric coordinates from current quad LOD to the

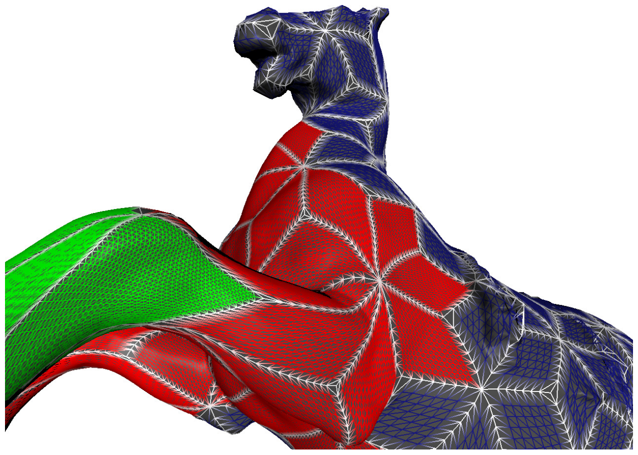


Figure 6: ***LOD seamless tessellation.*** *Seamless tessellation among patches at different LODs: vertices are snapped on the edges at the edge LOD, which depends from the projected edge length on the screen.*

edge LOD using following equations:

$$edgesize = 2^{edgelevel+1} - 1$$

$$\mathbf{uv} = \frac{round(\mathbf{uv} \cdot edgesize)}{edgesize}$$

Once modified (u, v), and set LOD as the current edge LOD, the sampling procedure is the same as for inner vertices. The dequantization is performed with a scale factor that depends from (u, v) as highlighted in 5.2: we need to get the 4 quad corners quantization min and max values, and bilinearly interpolate them. The resulting min,max pair is used to dequantize the vertex displacement. The quantization factors obtained in such a way permits to have the same values all over the edge between two adjacent patches, because they derives only from the interpolation of the two corners defining that edge. Corners min,max quantization factors are four pairs of 16 bit values stored for each quad of each levels into a static texture which is loaded at initialization and reused at each frame. The base quad position is given by the bilinear interpolation of the 4 quad corners at the possibly modified (u, v) coordinates. Then, if vertex is not one of the four corners, its value is offset by the vector found in the geometry texture mipmap at remapped uv coords, considering the quad offset and the quad size (see Fig. 5). The resulting rendering is seamless (see Fig. 6).

Color and normal (u, v) coordinates are found in a similar way, except for the snap step, which revealed to be not necessary for these attributes. Then these coordinates are passed to the fragment shader which takes care of properly sampling color and normal mipmaps to perform per pixel texturing and shading.

5.4 Adaptive streaming

Our compressed representation forms the basis of a scalable streaming system able to adapt to client characteristics and to exploit available network bandwidth.

The server component provides access to tile repositories, without differentiate among position, normal, or color components. From the server's point of view, a repository is just a database with a unique key for indexing a block of bytes containing an encoded bit-stream representing a compressed wavelet coefficient matrix. In order to increase server-side scalability, no processing is done in the server, whose only behavior is to return a block of bytes if present. This approach makes it possible to leverage existing database components instead of being forced to implement a specific storage manager. In this work, storage management is done through Berkeley DB, and data serving is done through an Apache2 server extended with an appropriate module.

The client implements streaming using asynchronous data fetching during rendering. During rendering, requests for missing patches are remapped to unique identifiers built from pyramid ids and tile ids and stored in a request queue. The priority is the difference between the desired patch LOD and the currently available one. At the end of the frame, only as many new requests as those allowed by the estimated network bandwidth are issued and managed by a separate network access thread, and the remaining ones are ignored.

A separate thread takes care of getting data from the server and possibly decompresses them as in the case of PNG tiles. When data becomes available it is inserted into the proper pyramid texture mipmap.

6 Implementation and Results

An experimental software library and viewer applications supporting the AQP technique have been implemented both using the OpenGL and WebGL environments. The OpenGL version, implemented in C++, works both on Linux and Windows platforms, and can be also used as a web-browser plugin using QT 4.8. The WebGL version is written in JavaScript on top of the publicly available SpiderGL library [Di Benedetto et al. 2010].

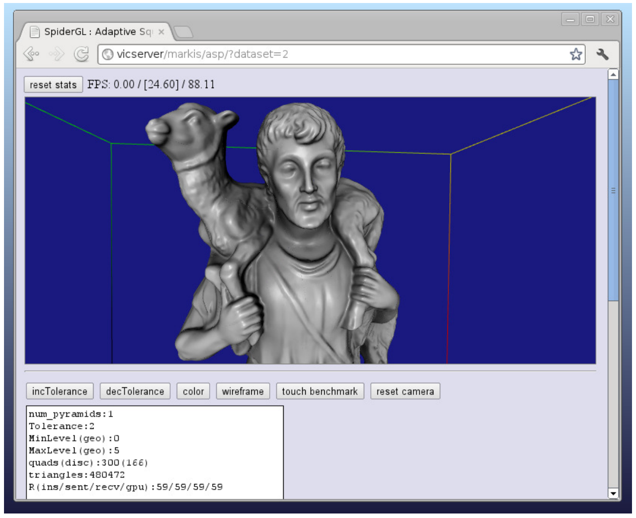
We have extensively tested our system with a number of datasets. In this paper, we discuss the results obtained with the models in Fig. 7, all coming from laser scanning acquisitions. The complexity of the input datasets ranges from 8Mtriangles to 90Mtriangles (left to right). Given the scope of this paper, we restrict the evaluation to the WebGL version of our code.

6.1 Preprocessing and compression rates

Table 1 shows the processing results of the various datasets, using the WebGL version of our code, which uses only PNG compression applied to delta encoded samples. It is clear that compression rates can be heavily improved by using DXT1 and DXT5 to compress attributes, but these compressed encodings are not widely supported in WebGL implementations. For this reason, the benchmarks presented in this paper use plain textures and PNG encoding for transport. Even with such a simple approach, the method is able to encode a sampled geometry in about 15bps for models with positions and normals, and about 24bps for colored models. It should be noted that our objective is not to achieve state-of-the-art compression rates, but, rather, to propose a method supporting adaptive streaming, and variable resolution rendering with an easy implementation in a WebGL context.

Dataset	Input Tri	Out level	Patch Count	Output Samples	Geom. bps	Color bps	Normal bps
Dwarf	8.4M	7	300	6.4M	6.3	9.54	8.53
Shepherd	8.4M	7	300	6.4M	6.3	0	8.71
Horse	8.4M	7	300	6.4M	6.3	9.21	8.42
Head	94.4M	9	180	62.5M	6.3	0	8.40

Table 1: *Processing results.* *Adaptive quad patches representations of the test models.*



FPS: 0.00 / [24.60] / 88.11

```
num_pyramids:1
Tolerance:2
MinLevel(geo):0
MaxLevel(geo):5
quads(disc):300(166)
triangles:480472
R(ins/sent/recv/gpu):59/59/59/59
```

Figure 8: *WebGL implementation running in Chrome.* *3D content can be delivered in a HTML5 canvas. Models are incrementally loaded during rendering.*

6.2 Adaptive rendering

The rendering tests were performed on a 1.6 GHz laptop, 4GB RAM equipped with an Nvidia Geforce GTX 260M with 1 GB video memory, and running a Linux Gentoo 2.6.39 distribution. In the tests presented in this paper we used Chromium browser version 19.0.1084.24 beta. The model exploration has been tested on a variety of situations ranging from far views to strict close-ups with sudden rotations to stress the capabilities of the system. In all cases we were able to sustain interactive rendering rates with 1-pixel accuracy with an average frame rates of 37 fps and never going below 13 fps. Lowest frame rates appear when the renderer is receiving tiles when higher resolution patches are needed. The qualitative performance of our adaptive renderer is illustrated in an accompanying video that shows live recordings of flythrough sequences. Sequences were recorded on a window of 750×350. Fig. 8 shows a frame of the recorded sequence.

6.3 Network streaming

Extensive network tests have been performed on all test models, on an ADSL 8Mbit/s connection, on a mobile broadband connection, as well as on an intranet. Tests have been made for the viewer application under interactive control.

In an interactive setting, since rendering is progressive and, on average, viewpoint motion is smooth, only few new patches per frame need to be refined, and only data for patches not already cached are requested to the server. We have measured the bandwidth required by a client to provide a "no delay" experience in typical inspection sequences. We measured an average bit rate of $312Kbps$ for exploration of areas not previously seen, and peaks of $2.8Mbps$ at viewpoint discontinuities, i.e., when the application has to refine the model all the way to the new viewpoint and the refinement algorithm has to always push new patches to refine in the request queue because of non incremental update.

By introducing client-side or server-side bandwidth limitations, it is possible to reduce burden on network and server, making the system more scalable while maintaining a good interactive quality. Due to the reasonable compression rate and refinement efficiency, we have found that using the system on a 8 Mbps ADSL line produces

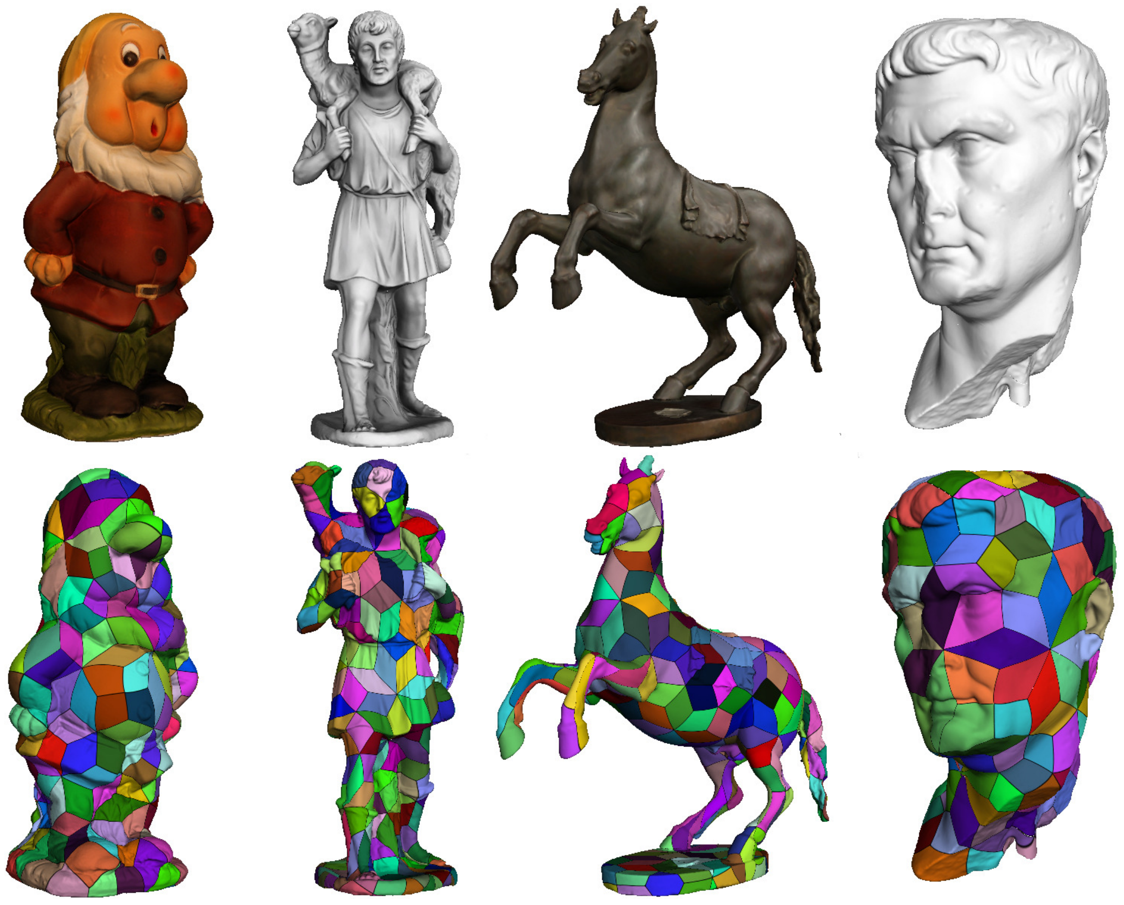


Figure 7: ***Models rendered with the adaptive quad patch method.*** *Top row shows the rendered models at pixel tolerance one. Bottom row shows the patch structure. Complexity ranges from 6.4 to 62.5Msamples.*

nice interactive results. In that case, delays in case of rapid motion become visible, but with little detriment to interaction (e.g., only 2s are needed to produce a fully refined model visualization from scratch). This also allows a single low-end server to manage a large number of clients.

7 Conclusions and Future Work

We have presented an approach for creating and distributing compact, streamable, and renderable 3D model representations. As for other compressed streamable formats, we specialize on a particular kind of models and do not strive to exactly replicate the original geometry and color, but only to visually approximate them in a faithful way. With these constraints, we are able to produce an effective distributed system. Due to the small run-time CPU overhead of the rendering component, and to the simplicity of the structures involved, real-time performance is achieved both on conventional GPU platforms using OpenGL, as well as on the emerging web-based environments based on WebGL.

Besides improving the proof-of-concept implementation, we plan to extend the presented approach in a number of ways. In particular, we plan to explore more aggressive compression techniques based on exploiting the hierarchical representation and on factoring repeated content. We are also currently incorporating occlusion culling techniques, useful for datasets with a high depth complexity, and we plan to introduce more sophisticated shading/shadowing techniques.

Our approach can be seen as step aiming at bridging the gap that currently exists from general-purpose meshes to rendering oriented structures based on real-time tessellation with normal/bump maps, which are typical of modern gaming platform but currently require considerable human effort to create. We see a number of possible applications of this technology. These include the automatic creation of rapidly renderable objects for local and online games and the set-up of browsable 3D models repositories directly available within WebGL-enabled browsers.

Acknowledgments. This research is partially supported by the EU FP7 Program under the DIVA project (290277).

References

BABOUD, L., AND DÉCORET, X. 2006. Rendering geometry with relief textures. In *Graphics Interface*, C. Gutwin and S. Mann, Eds., 195–201.

BLUME, A., CHUN, W., KOGAN, D., KOKKEVIS, V., WEBER, N., PETTERSON, R., AND ZEIGER, R. 2011. Google body: 3d human anatomy in the browser. In *ACM SIGGRAPH 2011 Talks*, ACM, 19.

BORGEAT, L., GODIN, G., BLAIS, F., MASSICOTTE, P., AND LAHANIER, C. 2005. Gold: interactive display of huge colored and textured models. *ACM Trans. Graph. 24*, 3, 869–877.

BOUBEKEUR, T., AND SCHLICK, C. 2005. Generic mesh refinement on gpu. In *Graphics Hardware 2005*, 99–104.

BOUBEKEUR, T., AND SCHLICK, C. 2008. A flexible kernel for adaptive mesh refinement on gpu. *Computer Graphics Forum 27*, 1, 102–114.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2004. Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Trans. Graph. 23*, 3, 796–803.

CIGNONI, P., DI BENEDETTO, M., GANOVELLI, F., GOBBETTI, E., MARTON, F., AND SCOPIGNO, R. 2007. Ray-casted blockmaps for large urban visualization. *Computer Graphics Forum 26*, 3 (Sept.).

DI BENEDETTO, M., PONCHIO, F., GANOVELLI, F., AND SCOPIGNO, R. 2010. Spidergl: A javascript 3d graphics library for next-generation www. In *Web3D 2010. 15th Conference on 3D Web technology*. note.

DYKEN, C., REIMERS, M., AND SELAND, J. 2009. Semi-uniform adaptive patch tessellation. *Computer Graphics Forum 28*, 8 (Dec.), 2255–2263.

FLOATER, M. S., AND HORMANN, K. 2005. Surface parameterization: a tutorial and survey. In *Adv. in Multires. for Geom. Model.*, Math. and Vis. Springer, 157–186.

GOBBETTI, E., AND MARTON, F. 2004. Layered point clouds. In *Proc. Eurographics Symposium on Point Based Graphics*, 113–120, 227.

GOBBETTI, E., AND MARTON, F. 2004. Layered point clouds – a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers and Graphics 28*, 6, 815–826.

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry images. In *Proc. SIGGRAPH*, J. Hughes, Ed., 335–361.

GUTHE, M., BALÁZS, A., AND KLEIN, R. 2005. Gpu-based trimming and tessellation of nurbs and t-spline surfaces. *ACM Transactions on Graphics 24*, 3 (Aug.), 1016–1023.

HOPPE, H. 1997. View-dependent refinement of progressive meshes. In *SIGGRAPH 97 Conference Proceedings*, Addison Wesley, T. Whitted, Ed., Annual Conference Series, ACM SIGGRAPH, 189–198. ISBN 0-89791-896-7.

HU, L., SANDER, P., AND HOPPE, H. 2010. Parallel view-dependent level-of-detail control. *IEEE Trans. on Visualization and Computer Graphic*.

JOVANOVA, B., PREDA, M., AND PRETEUX, F. 2008. Mpeg-4 part 25: A generic model for 3d graphics compression. In *Proc. 3DTV*, IEEE, 101–104.

KAZHDAN, M., BOLITHO, M., AND HOPPE, H. 2006. Poisson surface reconstruction. In *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 61–70.

KHODAKOVSKY, A., LITKE, N., AND SCHRÖDER, P. 2003. Globally smooth parameterizations with low distortion. *ACM Trans. Graph. 22*, 3, 350–357.

KRAEVOY, V., AND SHEFFER, A. 2004. Cross-parameterization and compatible remeshing of 3d models. *ACM Trans. Graph. 23*, 3, 861–869.

LAB, L., 2009. 1 billion hours, 1 billion dollars served: Second life celebrates major milestones for virtual worlds. `http://lindenlab.com/pressroom/releases/22_09_09`. Retrieved on 15 Sep. 2010.

LEE, A. W. F., SWELDENS, W., SCHRÖDER, P., COWSAR, L., AND DOBKIN, D. 1998. Maps: Multiresolution adaptive parameterization of surfaces. *Comp. Graph. Proc.*, 95–104.

LUEBKE, D., AND ERIKSON, C. 1997. View-dependent simplification of arbitrary polygonal environments. In *ACM Computer Graphics Proc., Annual Conference Series, (SIGGRAPH 97)*, 199–208.

MAGLO, A., LEE, H., LAVOUÉ, G., MOUTON, C., HUDELOT, C., AND DUPONT, F. 2010. Remote scientific visualization of progressive 3d meshes with x3d. In *Proc. Web3D*, 109–116.

MANUEL M.OLIVEIRA, F. P. 2005. An efficient representation for surface details. Tech. Rep. RP 351, Universidade Federal do Rio Grande, January.

NIEBLING, F., KOPECKI, A., AND BECKER, M. 2010. Collaborative steering and post-processing of simulations on hpc resources: Everyone, anytime, anywhere. In *Proceedings of the 15th International Conference on Web 3D Technology*, ACM, 101–108.

OLIVEIRA, M. M., BISHOP, G., AND MCALLISTER, D. 2000. Relief texture mapping. In *Proceedings of the Computer Graphics Conference 2000 (SIGGRAPH-00)*, ACMPress, New York, S. Hoffmeyer, Ed., 359–368.

PIETRONI, N., TARINI, M., AND CIGNONI, P. 2010. Almost isometric mesh parameterization through abstract domains. *IEEE Transactions on Visualization and Computer Graphics 16*, 4, 621–635.

POLICARPO, F., OLIVEIRA, M. M., AND COMBA, J. L. D. 2005. Real-time relief mapping on arbitrary polygonal surfaces. *ACM Trans. Graph 24*, 3, 935.

PRAUN, E., AND HOPPE, H. 2003. Spherical parametrization and remeshing. *ACM Trans. Graph. 22*, 3, 340–349.

SANDER, P. V., WOOD, Z. J., GORTLER, S. J., SNYDER, J., AND HOPPE, H. 2003. Multi-chart geometry images. In *Eurographics Symposium on Geometry Processing*, 146–155.

SCHREINER, J., ASIRVATHAM, A., PRAUN, E., AND HOPPE, H. 2004. Inter-surface mapping. *ACM Trans. Graph. 23*, 3, 870–877.

SHEFFER, A., PRAUN, E., AND ROSE, K. 2006. Mesh parameterization methods and their applications. *Foundations and Trends in Computer Graphics and Vision 2*, 2, 105–171.

SHIUE, L.-J., JONES, I., AND PETERS, J. 2005. A realtime gpu subdivision kernel. *ACM Transactions on Graphics 24*, 3 (Aug.), 1010–1015.

WANG, L., WANG, X., TONG, X., LIN, S., HU, S.-M., GUO, B., AND SHUM, H.-Y. 2003. View-dependent displacement mapping. *ACM Trans. Graph. 22*, 3, 334–339.

WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2004. Generalized displacement maps. In *Proceedings of the 2004 Eurographics Symposium on Rendering*, Eurographics Association, D. Fellner and S. Spencer, Eds., 227–234.

WU, J., AND KOBBELT, L. 2002. Fast mesh decimation by multiple-choice techniques. In *Proceedings of 7th International Fall Workshop on Vision, Modeling, and Visualization*, 241–248.

XIA, J., AND VARSHNEY, A. 1996. Dynamic view-dependent simplification for polygonal models. In *IEEE Visualization '96 Proc.*, R. Yagel and G. Nielson, Eds., 327–334.

YOON, S.-E., SALOMON, B., GAYLE, R., AND MANOCHA, D. 2004. Quick-VDR: Interactive view-dependent rendering of massive models. Tech. Rep. TR04-011, Department of Computer Science, University of North Carolina - Chapel Hill, Apr. 12. Mon, 12 Apr 2004 17:34:34 UTC.

YOON, S., GOBBETTI, E., KASIK, D., AND MANOCHA, D. 2008. *Real-time Massive Model Rendering*, vol. 2 of *Synthesis Lectures on Computer Graphics and Animation*. Morgan and Claypool, August.