Compression-domain Seamless Multiresolution Visualization of Gigantic Triangle Meshes on Mobile Devices

Marcos Balsa Rodríguez
CRS4

Enrico Gobbetti
CRS4*

Fabio Marton
CRS4

Alex Tinti
CRS4

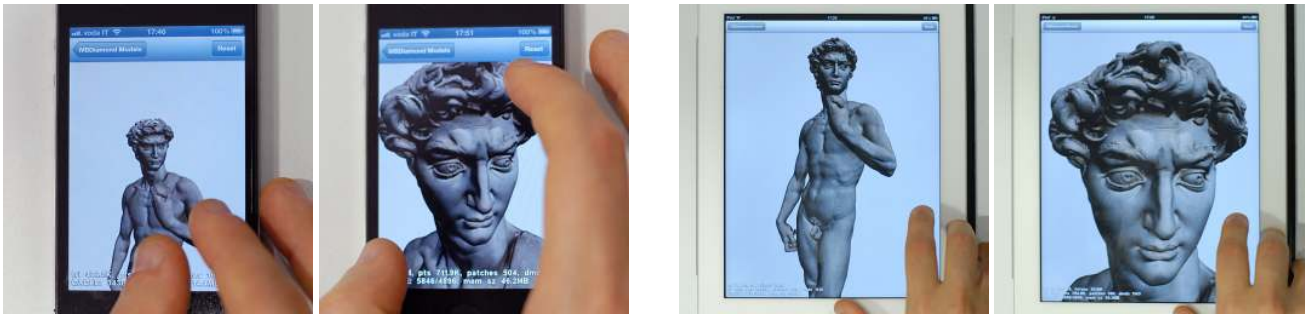


Figure 1: *A 1Gtriangles colored model of Michelangelo's David interactively inspected on a iPhone 4 and on "the new iPad". The iPhone has a 1Ghz Apple A4 processor with 512 MB RAM, a PowerVR SGX535 GPU and a screen resolution of 640x960 pixels, while the iPad has a 1Ghz Dual-core Apple A5X processor with 1GB RAM, a PowerVR SGX543MP4 GPU and a screen resolution of 2048 x 1536 pixels.*

Abstract

We present a software architecture for distributing and rendering gigantic 3D triangle meshes on common handheld devices. Our approach copes with strong bandwidth and hardware capabilities limitations in terms with a compression-domain adaptive multiresolution rendering approach. The method uses a regular conformal hierarchy of tetrahedra to spatially partition the input 3D model and to arrange mesh fragments at different resolution. We create compact GPU-friendly representations of these fragments by constructing cache-coherent strips that index locally quantized vertex data, exploiting the bounding tetrahedron for creating local barycentic parametrization of the geometry. For the first time, this approach supports local quantization in a fully adaptive seamless 3D mesh structure. For web distribution, further compression is obtained by exploiting local data coherence for entropy coding. At run-time, mobile viewer applications adaptively refine a local multiresolution model maintained in a GPU by asynchronously loading from a web server the required fragments. CPU and GPU cooperate for decompression, and a shaded rendering of colored meshes is performed at interactive speed directly from an intermediate compact representation using only 8bytes/vertex, therefore coping with both memory and bandwidth limitations. The quality and performance of the approach is demonstrated with the interactive exploration of gigatriangle-sized models on common mobile platforms.

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: mobile graphics, multiresolution, massive models, compressed streaming

*CRS4 Visual Computing, POLARIS Ed. 1, 09010 Pula, Italy www: http://www.crs4.it/vic/ e-mail: {mbalsa|gobbetti|marton|talex}@crs4.it

1 Introduction

We are currently immersed in an information society in which everyone is continuously connected to a vast information landscape through mobile devices and pervasive high-speed Internet. In many domains, such as cultural heritage, detailed high density 3D models are an important ingredient of the information flow that needs to be made available to the public.

Despite the impressive continuous improvements of mobile hardware, however, mobile 3D graphics is still constrained, compared to the desktop counterparts, by limited resources of low computing powers, low memory bandwidths, small amounts of memory, and limited power supply. Streaming and rendering extremely detailed 3D models, such as the ones created by 3D scanning and required for cultural heritage applications, thus requires particularly well crafted algorithms and data structures.

In this paper, we present a software architecture capable of distributing and rendering gigantic dense 3D triangle meshes on common handheld devices. Our approach copes with the strong platform limitations using a compression-domain adaptive multiresolution rendering approach.

The method builds on Adaptive TetraPuzzles (*ATP*) [Cignoni et al. 2004] by using a regular conformal hierarchy of tetrahedra to spatially partition the input 3D model and to arrange mesh fragments at different resolution in an implicit diamond. In this work, however, tetrahedra not only partition but also clip the original triangulation. We can thus create compact GPU-friendly representations of each fragment by constructing cache-coherent strips that index compact interleaved quantized vertex data, exploiting the bounding tetrahedron for creating local barycentric parametrization of the geometry. Appropriate boundary constraints are introduced in the splitting, simplification, and quantization steps to ensure that all conforming selective subdivisions of the hierarchy of tetrahedra lead to correctly matching surface fragments. For the first time, this approach supports local quantization in a fully adaptive seamless 3D mesh structure. For web distribution, further compression is obtained on top of the compact GPU-friendly representation by exploiting local data coherence using a low-complexity coding approach based on a wavelet transformation followed by entropy coding of coefficients.

At run-time, mobile viewer applications adaptively refine a local multiresolution model by managing a local GPU cache and asyn-

chronously loading on-demand from a web server the required fragments. CPU and GPU cooperate for decompression, and a shaded rendering of colored meshes is performed at interactive speed directly from an intermediate compact representation that uses only 8bytes/vertex, therefore coping with both memory and bandwidth limitations. Keeping data compact is of particular importance with current mobile devices, with extremely large screen resolutions, which dictate large rendering working sets, but limited main memory sizes. For instance, the current iPad generation sports a 3Mpixel display, but has a RAM capacity of only 1GB. Moreover, by decoding compressed data on-the-fly on graphics hardware, we can not only reduce local memory consumption, but also power consumption, thanks to reduced memory access and data transmission through the system bus.

As highlighted in the short overview of related work (Sec. 2), while certain other approaches share some of our method's properties, they typically do not meet the capability to rapidly generate adaptive seamless meshes by rendering from a very compact representation on a mobile platform.

The efficiency of the approach has been successfully evaluated with a number of large models, including a massive 1G triangle colored model of Michelangelo's David (Sec. 7).

2 Related Work

Building an efficient system for the exploration of massive 3D meshes on mobile devices requires the improvement and combination of state-of-the-art results in a number of technological areas. In the following, we briefly discuss only the approaches most closely related to ours. Readers may refer to established surveys on massive model rendering [Gobbetti et al. 2008], compression [Alliez and Gotsman 2003; Peng et al. 2005], and mobile graphics [Capin et al. 2008] for further details.

Adaptive 3D model rendering on mobile devices. While many examples exist for rendering light 3D models on portable platforms (e.g., MeshPad [ISTI-CNR Visual Computing Lab 2012] for meshes or PCL [Marion 2012] for points), exploring massive models on mobile devices is still a hot research topic. Much of the work in model distribution has focused so far on compression of mesh structures rather than adaptive view-dependent streaming. MPEG-4 is a reference work in the field [Jovanova et al. 2008]. Classic methods for view-dependent LOD and progressive streaming of arbitrary meshes were built on top of fine-grained updates based on edge collapses or vertex clustering [Xia and Varshney 1996; Hoppe 1997; Luebke and Erikson 1997]. Many compression and streaming formats for the web have been built upon them [Maglo et al. 2010; Blume et al. 2011; Niebling et al. 2010]. These methods, however, are CPU-bound and spend a great deal of rendering time computing a view-dependent triangulation prior to rendering, making their implementation in a mobile setting particularly challenging. With the increasing raw power of GPUs, the currently higher-performance methods typically reduce the per-primitive workload by pre-assembling optimized surface patches [Cignoni et al. 2004; Yoon et al. 2004; Cignoni et al. 2005; Borgeat et al. 2005; Gobbetti and Marton 2004a; Gobbetti and Marton 2004b; Goswami et al. 2013], although this kind of approaches has been demonstrated to work on mobile devices only in the context of point-based rendering [Balsa Rodriguez et al. 2012]. Recently, Gobbetti et al. [2012] have proposed an efficient image-based mesh representation that, however, only works for models for which an isometric quad parametrization exists. We propose, instead, a general multiresolution structure based upon tetrahedral space partitioning specifically tailored for mobile devices. The method is based on ATP [Cignoni et al. 2004], and improves over it by introducing a compressed GPU-friendly representation that ensures crack-free surfaces while using local quantization. To our knowledge our method is the first one fully supporting local quantization in a general adaptive 3D mesh structure.

Streaming and rendering using compact representations. Compressed graphics data potentially enable mobile application to better utilize the limited storage space and bandwidth at all levels of the pipeline. Many mesh compression algorithms offer good performance in compression ratio for both topology and vertex attributes. MPEG-4 [Jovanova et al. 2009] is a reference work in the field, and includes 3D mesh coding (3DMC) algorithms based on topological surgery algorithm [Taubin and Rossignac 1998] and progressive forest split [Taubin et al. 1998]. State-of-the-art topology coders [Rossignac 2001] are capable to achieve the theoretical minimum of 1.62 bpt (bits/triangle), approximately 3.24 bpv (bits/vertex). The decoding processes are however rather complicated and do not construct structures suitable for fast direct rendering. We focus, instead, in computing a representation for geometry that reduces the bandwidth required to transmit it to the graphics subsystem. This is achieved by constructing, for each mesh fragment, compressed primitive-topology representations that ensure high vertex coherence, as well as reducing vertex data size. For topology, Chhugani et al. [2007] presented an algorithm tailored for hardware decompression with 8 bpt (16bpv) by maintaining a cache coherent triangle sequence, and Meyer et al. [2012] proposed a coding technique reaching 5 bpt (10 bpv), which, however, requires CUDA for decompression. Similarly to Chhugani et al. [2007], we sort topology and vertex data after computing a cache-coherent rendering sequence, using, however, a generalized strip optimized for the post-transform vertex cache rather than a triangle list. Hardware-compatible vertex data compression is typically achieved in this context by attribute quantization. Since global position quantization [Calver 2002; Purnomo et al. 2005; Lee et al. 2009] provides poor rate-distortion performance for large meshes, recent efforts have concentrated on local quantization techniques [Lee et al. 2010], which, however, lead to cracks for multiresolution meshes. In our work, we improve over these local quantization approaches by expressing positions of mesh fragment vertices in the barycentric coordinate system relative to the containing tetrahedron. Hardware-friendly normal compression is achieved through an octahedral parametrization of normals [Meyer et al. 2010]. For network transmission, as for most compression schemes, we exploit high correlation between adjacent vertices by using predictive and entropy coding of prediction residuals. Rather than using the typical linear prediction schemes [Peng et al. 2005], we use a non-linear approach based on wavelet lifting [Senecal et al. 2004], and apply it to both topology and vertex data.

3 Pipeline overview

Starting from a high-resolution triangle mesh, we build, using a parallel out-of-core process, a hierarchical multiresolution structure based upon a conformal tetrahedra partitioning of the model's bounding box, similarly to what is proposed by the *ATP* approach [Cignoni et al. 2004].

The leaves of the multiresolution structure contain the full resolution original model while inner nodes contain simplified representations of the geometry with appproximatively half of the number of triangles contained into the children. The building process is performed off-line by iteratively inserting triangles from the input mesh into the hierarchical structure, which is recursively refined in order to maintain a maximum triangle count in the leaves. Then, coarser representations are built bottom-up by recursively merging children nodes, with a maximum triangle count and a representation error threshold as constraints.

A two-stage compression schema is used to transform input data to a compact representation suitable for GPU rendering and then further compressing this structure to reduce network traffic. The tetrahedral structure is exploited to encode vertex positions with barycentric coordinates. Tetrahedral barycentric coordinates express the position of a vertex inside a tetrahedron, as the combination of its four corners. These coordinates can be quantized locally for each tetrahedra. To produce a conforming mesh, input triangles, differently from ATP, need to be clipped against the tetrahedra faces, thus each tetrahedron geometry is fully self contained. The continuity among adjacent tetrahedra is ensured when quantizing positions, by the fact that barycentric coordinates of vertices lying on tetrahedra faces are expressed as combination of only the three corners defining that face, which are the same among neighboring tetrahedra in a conformal hierarchy. Normal and colors are also encoded to produce a compact quantized representation which is further compressed for storage and streaming (see Sec. 4).

At rendering time, multiple clients can access the data through a server farm where the multiresolution models are stored (see Sec. 5). On the client, an adaptive rendering approach incrementally updates the representation that best fits the current point of view and retrieves the required data from the server in an incremental fashion. Differently from ATP, where the multiresolution structure was encoded by six binary trees of disjoint tetrahedra, we base our run-time structure on *diamonds*. Each diamond is composed by the set of all the tetrahedra sharing their longest edge. Using a diamond based structure, see Weiss and De Floriani [2010], dependencies are implicitly encoded into the hierarchy, and refinement is interruptible, producing a conforming mesh also when children data is not available. This feature perfectly fits in a mobile rendering architecture, where is common to experience data fetch delays due to bandwidth limitations. The working set is kept to a fixed small size by keeping data directly in a compact GPU format suitable for direct rendering through specific *shaders* that do the decoding directly in the *Vertex Shader* (see Sec. 6).

4 Building the multiresolution structure

The construction process of the multiresolution structure starts from a *triangle soup*, i.e., a flat list of triangles with direct vertex information, together with a list of boundary vertices. The process is composed of two main phases: a first one where the dataset is partititoned into a tetrahedra hierarchy, and a second phase where data is simplified in a bottom-up fashion to build inner node representations.

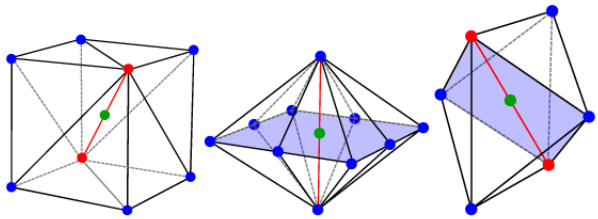
4.1 Tetrahedra Partitioning

The first phase starts with the insertion of the input triangles into the root diamond, which is generated by partitioning the bounding box of the mesh into six tetrahedra sharing a major box diagonal as their longest edge. Thereupon, in a top-down manner, input triangles are inserted into the tetrahedra hierarchy, which is recursively refined in order to maintain a maximum triangle count per leaf node. In contrast with ATP [Cignoni et al. 2004], when a new triangle is to be inserted, the triangle is clipped against each of the leaf nodes it overlaps. The generated triangles are then inserted into the corresponding node. In such a way, each tetrahedron fully contains its geometry. Whenever the number of triangles contained in a node exceeds a given limit, the tetrahedron σ is split by the plane passing through the midpoint of its longest edge and the opposite edge in σ. Then, the triangles contained in the tetrahedron are redistributed among the two children tetrahedra. Towards guaranteeing the conformality of the resulting tetrahedra mesh after this splitting, all the tetrahedra belonging to the same diamond of σ, are split at the same time.

After all the triangles have been inserted into the hierarchy, leaves contain the original geometry. Nonetheless, due to quantization errors, this representation slightly deviates from the input mesh. In order to be able to reproduce the original geometry, leaf nodes are further refined, recursively splitting triangles and inserting them into children tetrahedra until the quantization error is below a user defined threshold, generally a fraction of the average edge length. After this first phase, the original mesh is represented by the leaf nodes, while inner nodes are empty.

Triangle clipping requires special care to avoid producing disjoint vertices due to precision error on shared geometry (i.e., when splitting an edge shared by two neighbor tetrahedra). Before performing the clipping, the edge vertices are sorted according to a simple less operator which take into account also the plane orientation, thus obtaining a repeatable procedure.

This preprocessing builds a diamond hierarchy where each diamond consists of a set of tetrahedra. The diamond hierarchy contains three types of diamonds formed by 6, 4 and 8 tetrahedra, and contain respectively 8, 6 and 10 vertices [Weiss and De Floriani 2010], see Fig. 2. These three diamond configurations occur repeatedly in sequence during hierarchy traversals. Diamonds and tetrahedra corners are expressed on an integer grid, where diamonds are uniquely identified by their center, and tetrahedra are identified by their level and center integer coordinates. It is possible through a lookup table to go from a diamond to its children, parents, and constituent tetrahedra. Thus, we can avoid to store pointers in the preprocessing and also in the runtime structure.



(a) Level 0 (b) Level 1 (c) Level 2

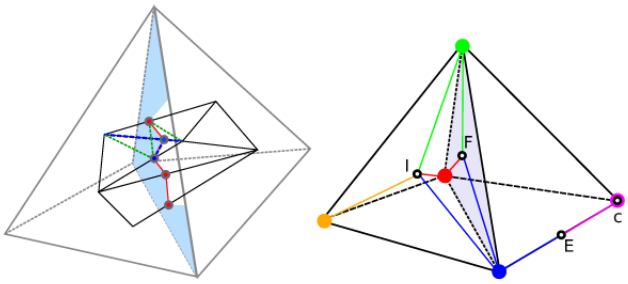
Figure 2: ***Sequence of diamond configurations.*** *There are three different diamond configurations that occur iteratively during the hierarchy refinement.*

4.2 Simplification

In a second phase, the hierarchy is traversed bottom-up in order to generate inner nodes simplified geometry representation. Each diamond is simplified independently from the other diamonds, with the constraint that vertices lying on boundary faces are left unchanged to maintain continuity among neighboring diamonds of the same level and adjacent levels of resolution.

There are three different cases to take into account: (a) inner boundary vertices, i.e., vertices near the inner faces of the diamond children, connecting with triangles from other tetrahedra inside the diamond; (b) outer boundary indices, i.e., vertices near the faces of the diamond, connecting with triangles of other diamonds or tetrahedra; (c) inner vertices: all other vertices. Vertices of type (a) can only be simplified against other (a) vertices in order to maintain the same representation in all neighboring tetrahedra sharing these vertices, so the reparameterization of this vertices to barycentric coordinates remains the same. Vertices of type (b) are fixed and cannot be simplified to ensure continuity with neighbor diamonds. Inner vertices of type (c) can be simplified in any manner. In brief, since

triangles are clipped to their containing tetrahedra, in contrast with ATP, during the simplification the only constraint is not to modify: external edges on the faces of the diamond that are shared with other diamonds, and internal edges on the faces of tetrahedra that are shared among two tetrahedra in the diamond. The latter is required to ensure that edge vertices have the same coordinates in both tetrahedra when reparameterized into barycentric coordinates, see Fig. 3(a). Because diamond topology changes at each level, there are no vertices that remain locked up to the root and all the data can be simplified to get a uniformly sampled model.



(a) Inner boundary (b) Barycentric coordinates

Figure 3: ***a) Geometry simplification.*** *An edge shared between two neighboring tetrahedra should be simplified only over the plane defined by the face connecting the tetrahedra. This way, the barycentric coordinates of the vertices will depend only on the coordinates of the vertices defining face and so will be common for both tetrahedra. The blue edge dissapears when blue vertices are merged. Green edges appear during the re-triangularization of the clipped geometry.* ***b) Barycentric coordinates.*** *Four examples of vertices with their positions with respect to the tetrahedron: I inner, F on face, E on Edge, C on corner. For each vertex, we show the corners which provide non null barycentric coordinate contribution.*

4.3 Barycentric parametrization and quantization

Before emitting tetrahedron data its geometry is reparameterized into barycentric coordinates. Each inner point is expressed as a linear combination of the 4 tetrahedron corners, while points lying on tetrahedron faces are expressed as combination of only the three corners defining the face. The latter, ensures continuity between neighboring tetrahedra since the points on shared faces will be defined as combination of the same three vertices, see Fig. 3(b). We chose this representation because it offers a compact representation that can be locally quantized without producing cracks between adjacent tetrahedra. However, due to precision error, boundary points could be placed not exactly on the faces, thus producing different values on different tetrahedra, once data is quantized, see Fig. 4.

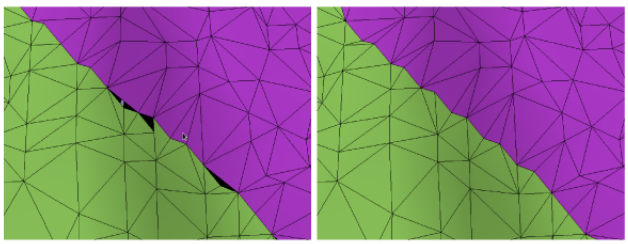
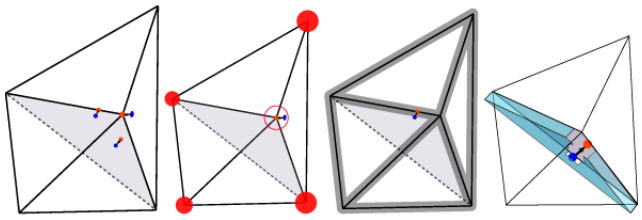


Figure 4: ***Geometry quantization.*** *Projecting boundary vertices on tetrahedron faces permit to solve mesh discontinuity among adjacent tetrahedra, which are due to quantization of sligtly different values.*

Hence, to ensure consistent quantizations, we subdivide the points in 4 cases depending on their position with respect to the containing tetrahedra: near a corner, near an edge, near one face, and inner point. In the first case the point is represented with only 1 not null coefficient, 2 coefficients for the second case expressing linear interpolation among 2 points, 3 in the third case for barycentric coordinates over a 3D triangle, and finally 4 for inner points. These cases are checked in this order, and with decreasing epsilon, to be sure to behave in the same manner on adjacent tetrahedra for the same points, see Fig. 5.



(a) Input vertex snapping (b) Snap to corners (c) Snap to edges (d) Snap to faces

Figure 5: ***Vertex snapping:*** *(a) Shows sample vertices that must be snapped onto (b) corners; (c) edges or (d) faces.*

The quantized geometry is stripified in a cache coherent way to maximize the number of cache hits of transformed vertices. Thereafter, vertices are reordered to match the order of appearance in the strip. This sorting permits to reflect the spatial position in the memory layout, thus keeping similar values (i.e., neighboring vertices) near each other, which will be useful for the compression step.

4.4 Compression

The quantized vertex coordinates are encoded together with normals and colors into a compact 64bit representation suitable for direct rendering, where 3 bytes are used for position, 2 bytes for normal and 3 bytes for color. Position is parameterized with 4 barycentric coordinates, but only 3 components are required since the four components sum to one. This 64bit/vertex encoding provides an extremely compact aligned representation that can be efficiently accessed on current GPUs, which typically require vertex data to be aligned on 32 bit boundaries. For a colored representation, 64bit/vertex is thus an optimal size. Normals are encoded using the *octahedron normal vector* approach [Meyer et al. 2010], which maps unit vectors to two parametric coordinates. This encoding consists in projecting the normals onto the octahedron by normalizing them with the 1-norm. The octahedron is unwrapped to a square and the $[u, v]$ parameters in the plane are quantized to 8 bits, leading to sub-degree precision [Meyer et al. 2010]. Decompression is numerically stable and requires only few basic operations which can be exectuted in the vertex shader.

This compact representation is well suited for rendering through a simple shader (see Sec.6), but higher compression rates can be achieved using a low-complexity codec applied to both vertex data and topology. In order to maximize data correlation for the entropy coding, color is transformed to the YCoCg reversible format [Malvar et al. 2008], and all of the vertex attributes (i.e., position, normal and color) are deinterleaved and separated into their components and stored as a sequence of streams, to which strip indices are also appended (i.e., $[P_0^0..P_n^0]$, $[P_0^1..P_n^1]$, $[P_0^2..P_n^2]$, $[N_0^u..N_n^u]$, $[N_0^v..N_n^v]$, $[Y_0..Y_n]$, $[Co_0..Co_n]$, $[Cg_0..Cg_n]$, $[I_o..I_m]$). Each stream is then transformed using a reversible n-bit to n-bit wavelet based on the Haar wavelet transform in order to reduce entropy [Senecal et al. 2004]. This approach uniformely treats topology and vertex data, and generalizes the usual linear prediction

methods typically applied to vertex positions. The low-pass coefficients produced by the wavelet transformation are iteratively filtered by the same wavelet until we remain with a single (root) approximation coefficient. The resulting approximation and detail coefficients are then transformed with a range codec: integers are arithmetically coded using a single symbol for the value 0, while other values are encoded using exponent, mantissa and sign, with different context for each encoded bit. This is a variation of the symbol encoding method used in the FFV1 Video Codec [Martin 1979]. During decompression all the steps are undone in reverse order.

In the course of the building process we also maintain a temporary version of the simplified data stored uncompressed on an external memory data repository, which is used to build coarser level simplifications without accumulating quantization errors.

4.5 Parallel processing

Simplification and encoding can be easily parallelized, being each diamond independent from the others. After the recursive subdivision, the simplification starts from the leaves and goes up to the root. A master process takes care of assembling each diamond of the current level, fetching corresponding tetrahedra geometry from the uncompressed data repository, and assigning it to a worker process. On the worker process, inner node diamonds get their geometry simplified, encoded and compressed. For leaf nodes, only encoding and compression are performed. After this job the worker sends the compressed diamond geometry back to the master node. After all diamonds of the current level are processed, the master starts to process the next coarser level and, level by level, the dataset is simplified up to the root. At that point, since children nodes are no more needed, their uncompressed representations can be discarded.

5 Server

On the server side, data for each model is stored in separated databases. In the pursuit of scalability, the server acts as a repository of data with zero processing overhead. An abstraction layer handles communication processes through different transport protocols, such as HTTP or direct connection through TCP. A simple module for Apache2 is in charge of handling HTTP requests, which relies upon a local database to efficiently locate the requested data. Berkeley DB is used for storage, accessing and caching data in the server side due to its open source license and its matureness as embeddable database. Berkeley DB provides an efficient and scalable transactional database engine with high reliability and availability, able to handle up to terabytes of data. It also allows configuration of per-process replicated cache and shared index memory among different database instances. Altogether, provides an scalable architecture with reduced memory load for servers when dealing with hundreds of clients in parallel. On the other side, Apache2 is a mature and open source server which provides an efficient, secure and extensible architecture for developing HTTP services. Its scablable and multithreaded architecture together with features like persistent server processes and load balancing are essential to the performance of our application. A custom Apache module implements a connectionless protocol based on HTTP which receives queries composed of database name and node identifier. This module extracts the query parameters, retrieves the corresponding data from the DB, and sends back either node's data or an empty message if it is not present. This architecture relies on mature components that have been widely tested and provide good scalability and performance when dealing with thousands of clients.

6 Client architecture description

Embedded devices such as Apple iPhone/iPad or Android devices in general, offer support for OpenGL ES, the specification for embedded devices. There are now two available versions exposed on those platforms: 1.1 and 2.0. OpenGL speficications for Embedded Devices have been defined to be a fully functional subset of its desktop counterparts where only the more general functionality has been included in order to minimize circuitry complexity and energy consumption.

The ES 1.1 version offers a lighter version of OpenGL 1.5, where inmediate mode has been supressed together with complex primitives such as quads or polygons. The functionalities include Vertex Buffer Objects (VBO) and Vertex Arrays to feed the GPU with geometric primitives In the ES 2.0 version, based upon the 2.0 desktop specification, the whole fixed pipeline functionality has been removed in favor of the shader based pipeline, where *Vertex* and *Fragment* shaders must be provided giving more flexibility. The GLSL speficication for ES has also been modified adding control for data precision.

Embedded GPUs typically focus on high efficiency and low power consumption, although nowadays they are able to offer decent computational power in comparison with desktop GPUs. The SnapDragon 2XX GPU integrated in Qualcomm processors, among other GPUs used in current mobile devices, use Tile Based Rendering (TBR). Only once all the primitives have been submitted the driver splits the geometry into tiles which are then rendered using a small amount of in-core memory. The PowerVR SGX5XX, used in the various iPhone/iPad series and some high-class Android devices, go a bit further and use Tile Based Deferred Rendering (TBDR), which delays fragment operations until occlusion tests have been processed avoiding expensive calculations for occluded fragments. This architecture, widely used in embedded GPUs, penalyzes reading back from the frame buffer since it requires waiting for all the tiles to be written prior to reading. In general, current generation of embedded GPUs provide really good performance together with an efficient energy consumption; although the continuously increasing display resolution makes the fragment load to penalyze heavily the rendering performance (i.e, iPad 3 resolution of 2,048 by 1,536 uses a PowerVR SGX543MP4 with 16 unified shader units to render this massive amount of fragments).

Taking into account these architecture constraints, the rendering engine has been designed to minimize fragment processing while feeding the GPU with large geometry batches using cache optimized indexed triangle strips.

6.1 Adaptive view-dependent representation

Each frame, depending on the viewing parameters and a given fixed screen space tolerance, the client performs an adaptive rendering of the multiresolution model. For this purpose, the client relies on a hierarchical multiresolution representation of the model that is incrementally refined depending on the navigation. Initially starting with a coarse representation of the whole model, the hierarchy is traversed for each render view point in order to determine the available working set. The traversal algorithm takes into account diverse parameters: the viewing position, the available GPU resources, the current CPU usage level, and the required network bandwith. Differently from ATP [Cignoni et al. 2004] the refinement is performed on a diamond basis. The viewer maintains the multiresolution structure as a map of diamonds, each of them identified by its center integer coordinates. For each diamond, on creation, there are available through a small lookup table its parents, children and tetrahedra indices. Each of the tetrahedra indices corresponds to an entry

in the cache containing the compact representation of the fragment geometry. To each diamond we associate a view dependent priority which is the projected average edge length if the diamond is visible, or zero otherwise. A diamond is refined if its priority is higher than a user selected pixel tolerance. Refinement of a diamond stops if it is not visible, if it should be refined but children data is still not available, or if it fulfills the viewing constraints. Diamond based refinement is capable of producing a conformal tetrahedral mesh when each diamond is split only if its parent diamonds are already present in the graph. Such a refinement has the valuable benefit of being interruptible, hence we can use memory, triangle and time budgets to limit the used resources and to avoid locks, thus permitting interactive performance. We update the multiresolution structure cut using two diamond heaps: the refinement one, which is sorted with decreasing priority, and contains the leaves of the cut, and the coarsening heap which contains the parents of the leaves, with increasing priority. At each frame, instead of traversing all the hierarchy from the root, we update the priority of each diamond on the two heaps, then we refine the top of the refinement heap until achieving the desired error threshold, or one of the budget constraints is reached. Once over a new frame we also coarsen the top of the coarsening heap to release resources. The two heaps are properly updated for each refinement and coarsening operation.

In RAM memory, we maintain the cache of tetrahedra compact geometries, which are indexed through the diamond graph. The cache implements a LRU policy that maximizes the reuse of nodes while enforcing a resource usage below a given limit. The compact format permits to directly map data as Vertex Buffer Objects, which are ready to be sent to the GPU. This compact representation also permits to perform raycasting without needing a decompression step. Raycasting is used to identify the touch point over the model for interaction purposes. Each tetrahedron also contains a small hierarchical tree of bounding boxes, computed just after loading, which is used to improve raycasting performance. LRU fragments are kept in the cache as long as they are referenced by the diamond graph. After a coarsening operation, when a fragment is no more referenced, it goes toward the end of the LRU and is discarded as soon as new resources are needed.

6.2 Multi-threaded data access layer

The retrieval of data is performed through an asynchronous data access layer which encapsulates the data fetching mechanism and avoids blocking the application when the requested data is not yet available. The main thread, in charge of performing the hierarchy traversal for determining the working set, asks the cache for the nodes required for the current view position. If the requested data is available, the node is returned and so the traversal continues until the best available representation is reached; otherwise, a new request for this node is enqueued and the traversal stops since this is the best available representation. Another thread is responsible of fetching the requested data, contained in a priority queue. Depending on an available bandwith estimation, a given number of requests is sent to the server, while the remaining requests are ignored. Since request priority corresponds to the node's projected error, coarser nodes are always requested first. On each frame, the request queue is cleared and filled again with the nodes needed for that frame, and so will be served at some point only after coarser nodes are available. This thread also handles incoming data and performs the decompression from the entropy coded version to the compact GPU representation, proceeding with the reverse sequence described in the preprocessing phase. Entropy decoding, then per component backward wavelet transform, and finally conversion from YCoCg to RGB. After this decompression, data is stored in an interleaved array of 8 bytes per vertex with 3 bytes for barycentric coordinates, 2 bytes for the octahedron normals, and 3 bytes for the RGB color.

Figure 6: ***Detail of David's eye interactively rendered on a iPad.*** *This 1Gtriangles moddel is colored using post-restoration color data. Note how our compression preserves extremely high quality details in shape, normal, and colors.*

6.3 Rendering process

Before rendering a simple shader is activated. The visible tetrahedra of the current cut are traversed by a visitor which takes care of managing a cache on GPU of Vertex Buffer Objects (VBO). The size of the GPU cache is smaller than the size of the CPU one, thus more memory remains for CPU data, limiting the need of requesting and decoding multiple times data that exited from the limited GPU resources. When a node is visited, if it is not present in the cache, a corresponding VBO is created and inserted into GPU cache and rendered, otherwise only rendering is performed. Rendering consists in binding the buffer, setting up the vertex attribute pointers and drawing the optimized stitched strip sequence present in the geometry indices. For alignment purposes, we address vertex attributes as two 4-bytes words, and let the shader separate the position, normal, and color components.

The shader must transform data expressed in local barycentric coordinates. The transformation is given by this simple equation $\mathbf{v} = ||\mathbf{c_0 c_1 c_2 c_3}|| \cdot |\mathbf{v_b}|$, where $\mathbf{c_i}$ represent the corner ith while $\mathbf{v_b}$ is the vector of the 4 barycentric coordinates. Thus the 4 corners can be replaced by a matrix, which is post-multiplied to the model view matrix. Therefore, rendering from barycentric coordinates is not causing extra per-vertex cost with respect to using Cartesian coordinates. Since color is already in the RGB24 format, the only extra work that nees to be performed is the decoding of normals from the two quantized octahedral map coordinates. From the quantized coordinates remapped into $[-1, 1]$ we compute $n_z = 1.0 - |u| - |v|$. Then if $n_z > 0$ we are on the upper side of the octahedron and $n_{xy} = uv$, otherwise we are on the lower part and we need to revert the n_{xy} components according to these equations: $n_x = (1 - n_y) \cdot sign(u)$ and $n_y = (1 - n_x) \cdot sign(v)$, see [Meyer et al. 2010] for further details. Attribute decoding cost is thus negligible with respect to the other work performed by the shader (in particular, transformation, projection, and shading). Fig. 6 illustrates the quality of rendering that can be achieved using compressed data.

6.4 Graphical User Interface

On the iOS platform, we've taken advantage of the Cocoa Touch UI framework to design a simple Graphical User Interface (GUI)

composed of a Model List Widget and OpenGL Rendering Layer. End users can easily browse and select the desired model through the Model List Widget and interact with the OpenGL Rendering Layer through standard multi-touch gestures. It's possible to rotate the model about its bounding box by moving a single finger on the screen, move the model with two fingers or zoom it in and out by performing a pinch gesture.

Interaction is also possible through an alternative "target-based" approach, with which a single quick tap by the user selects a target point which is attached to the model. This target point, rendered on screen as a small colored sphere, allows the user to easily rotate the model about by moving a single finger on the screen. By tapping the target again instead, it will trigger a smooth animation that moves the camera from its current position toward the target's position. The target sphere can be deactivated anytime by tapping outside of the model.

7 Results

Using this method, we developed a framework composed of a C++ preprocessor, a client iOS app, and a HTTP server. Several tests were performed on pre-processing and rendering of very large models. Here we present results relative to two 3D large models from the Digital Michelangelo Repository of Stanford University with 0.25mm resolution: the David statue with 940 M triangles, and the St. Matthew statue with 374 M triangles. The David model is enriched with the color signal acquired after restoration and blended with geometry with the algorithm proposed by [Pintus et al. 2011], while the St. Matthew has a precomputed ambient occlusion gray scale per vertex.

7.1 Dataset Construction Performance

The preprocessor has been implemented using C++ and the OpenMPI high performance message passing library. Each model has been processed using a single off-the-shelf PC with Linux 3.0.6 (Gentoo distribution) and an Intel(R) Core(TM) i7 CPU 960 @ 3.20GHz with 24GB RAM. We constructed all multiresolution structures with a target maximum leaf size of 8000 triangles/tetrahedron and a leaf quantization tolerance of 0.25mm.

Processing of the David statue took about 10h45m on 8 cores, while about 4h15m for the St Matthew, which correspond to roughly 24k triangles/second.

The data compression rate is 49.1 bits/vertex for the David model and 45.1 bits/vertex for the St. Matthew model, including the mesh topology information, see Table 1. The David's color information is heavier than St.Matthews, since the latter is only a low frequency ambient occlusion component. Instead, normals of the St. Mattew require more bits due to the roughness of the surface with respect to the David. Color encoding is lossless, and normal error induced by octahedron encoding is subdegree, and this represents the limit of this method when using 8 bits per component. Our wavelet transformation and entropy coding step produces a compression of about 2.3x for vertex data and 3.5x for mesh topology with respect to our compact GPU friendly representation.

7.2 Rendering performance

The client was implemented on iOS 6 using C++, OpenGL and Objective-C++/C++. We evaluated the rendering performance of the technique on a number of inspection sequences on an iPhone 4 and on a 3rd generation iPad. The iPhone has a 1Ghz Apple A4 processor with 512 MB RAM, a PowerVR SGX535 GPU and a screen resolution of 640 x 960 pixels, while the iPad has a 1Ghz Dual-core Apple A5X processor with 1GB RAM, a PowerVR SGX543MP4 GPU and a screen resolution of 2048 x 1536 pixels. The two devices were chosen as representative extreme cases. The iPad has the currently largest screen, while the iPhone is an "old generation" phone with average specs. It must be considered that the current generation of mobile devices, such as the Apple iPhone 5, Samsung Galaxy S3 and Note 2, have technical specifications similar or even higher than the iPad 3's, with a sensibly smaller screen resolution. We can thus expect much better results on these newer generation devices.

	David bpv	St Matthew bpv
Position	10.9	10.9
Color	9.1	3.8
Normal	8.6	9.7
Mesh Topology	20.5	20.7
Total	49.1	45.1

Table 1: ***Bit rates.*** *Bits per vertex subdivided per position, color, normal and mesh topology for the two processed models.*

The quantitative results presented here in details were collected during interactive inspections with pixel tolerance 3 of the David and St. Matthew models. The sessions were designed to be representative of typical mesh inspection tasks and to heavily stress the system, and includes rotations and rapid changes from overall views to extreme close-ups. The qualitative performance of our adaptive renderer is also illustrated in an accompanying video, that shows live recordings of the analyzed sequences. Representative frames are shown in Fig. 7.

On the 3rd generation iPad we are able to render models with an average throughput of 30 Mtriangles/second, with an average rendering frame-rate of 37 fps, which eventually drops to 15 for full refined views, when the number of triangles reaches the 2 Mtriangles maximum triangle budget. As expected, the iPhone 4 got slightly worse results in terms of interactivity, with a throughput of 2.8 Mtriangles/second, with an average frame-rate of 10 fps, and a worst case of 2.8 fps for views with maximum of 1 M Triangle budget. As demonstrated in the video, performance is perfectly adequate for interactive inspection tasks. The quality of representation is extremely high. An example is presented in Fig. 6.

7.3 Streaming performance

The latency time needed to fully refine the data at the application start-up and to refine the model during the exploration, is probably one of the most critical issues that a mobile device need to deal with. Of course, this time is independent from the rendering thread but only depends on the network bandwidth. The multiresolution structure along with the output-sensitive technique adopted, allow the client to only need a working set which depends on the screen resolution. Hence, the latency time to download the current working set is proportional to the maximum resolution of the mobile device. We have measured performance with a wireless connection of a Linksys WAP 200 802.11 b/g access point 54 Mbps, as well as with UMTS/HSPA connections. The wireless network was shared among many clients, and we measured its peak performance to be 17 Mbps.

With the iPad, at start-up we need to load about 14.5MB to see the whole David statue in full screen (1.1Mtri), and 19.9 MB to see the St. Matthew (1.8Mtri). Our application performs data fetching asynchronously in a separate thread to avoid delaying interactive rendering. We measured data fetching speed to be of about 4.8Mbps on the wireless network. We are thus able to use about 35% of

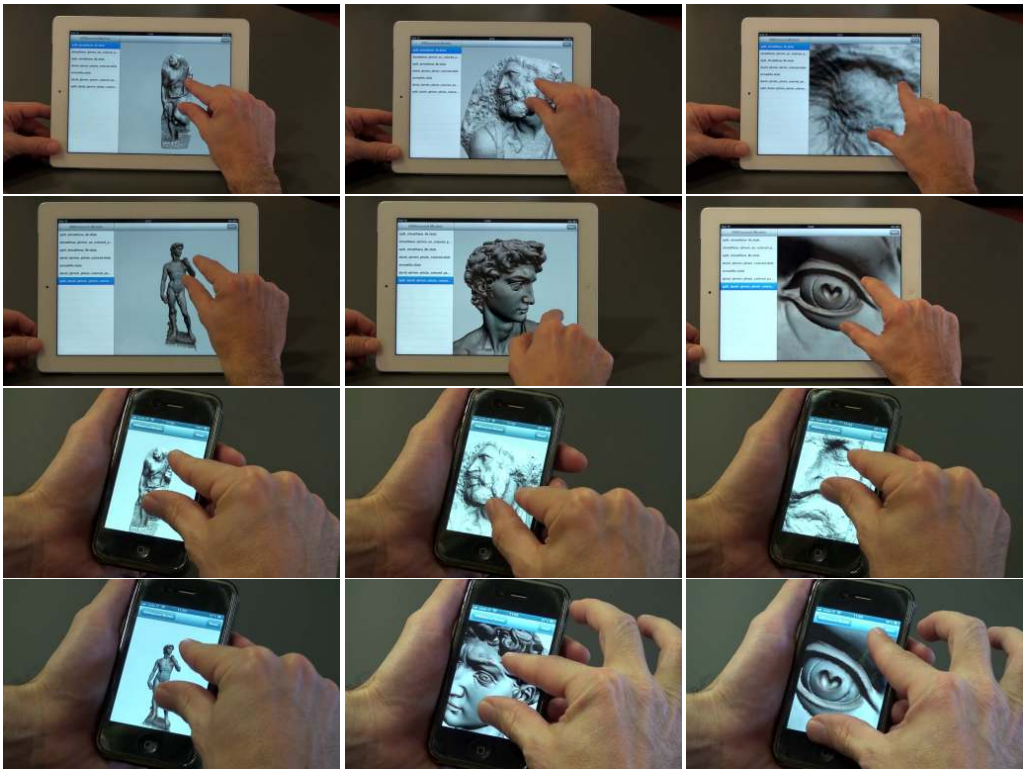


Figure 7: ***St. Matthew and David on a 3rd generation iPad and a iPhone 4.*** *Representative frames from the accompanying video illustrating the interactive remote exploration of the colored David (1Gtri) and St. Matthew (374Mtri) datasets. The average frame rate is 37fps on the iPad and 10fps on the iPhone. Triangle throughputs vary from 30Mtri/s on iPad to 2.8Mtri/s on iPhone.*

the available bandwidth. Full refinement takes about 30s for both statues. Due to progressive refinement, after a couple of seconds, however, the statues are already visible with a reasonable quality. On the UMTS/HSPA, the data fetching speed was measured to be about 3.3Mbps, for a full refinement latency of about 45s. The iPhone4 is about 1.5x slower, which is mostly due to the lower CPU performance, which leads to incresed decoding time.

8 Conclusions

We have presented an architecture capable of distributing and rendering gigantic 3D triangle meshes on common handheld devices. Our architecture exploits the properties of conformal hierarchies of tetrahedra to produce a data structure which is adaptive, compact, and GPU friendly. By combining CPU and GPU compression technology with our multiresolution data representation, we are able to incrementally transfer, locally store and render extremely detailed models on hardware-constrained mobile devices with unprecedented performance.

Besides improving the proof-of-concept implementation, we plan to extend the presented approach in a number of ways. In particular, we are currently incorporating occlusion culling techniques, useful for datasets with a high depth complexity, and we plan to introduce more sophisticated shading/shadowing techniques.

This enabling technology is intended to be a high performance building block for mobile 3D graphics. A major application area of massive model rendering is cultural heritage, where highly detailed representations area required to reproduce the unique aura of real objects. We also plan to better integrate this technology with web infrastructurs by providing an implementation running in WebGL.

Acknowledgments. This work is partially supported by the EU FP7 Program under the DIVA project (REA Agreement 290277). We also acknowledge the contribution of Sardinian Regional Authorities.

References

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

BALSA RODRIGUEZ, M., GOBBETTI, E., MARTON, F., PINTUS, R., PINTORE, G., AND TINTI, A. 2012. Interactive exploration of gigantic point clouds on mobile devices. In *The 14th International Symposium on Virtual Reality, Archaeology and Cultural Heritage*, 57–64.

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 (July), 869–877.

CALVER, D. 2002. Vertex decompression in a shader. *ShaderX: Vertex and Pixel Shader Tips and Tricks*, 172–187.

CAPIN, T., PULLI, K., AND AKENINE-MOLLER, T. 2008. The state of the art in mobile graphics research. *Computer Graphics and Applications, IEEE 28*, 4, 74–84.

CHHUGANI, J., AND KUMAR, S. 2007. Geometry engine optimization: cache friendly compressed representation of geom-

etry. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '07, 9–16.

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 polygonal models. *ACM Trans. Graph. 23*, 3 (August).

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2005. Batched multi triangulation. In *Proceedings IEEE Visualization*, IEEE Computer Society Press, Conference held in Minneapolis, MI, USA, 207–214.

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 & Graphics 28*, 1 (February), 815–826.

GOBBETTI, E., KASIK, D., AND YOON, S.-E. 2008. Technical strategies for massive model visualization. In *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, ACM, New York, NY, USA, SPM '08, 405–415.

GOBBETTI, E., MARTON, F., BALSA RODRIGUEZ, M., GANOVELLI, F., AND DI BENEDETTO, M. 2012. Adaptive Quad Patches: an adaptive regular structure for web distribution and adaptive rendering of 3D models. In *Proc. ACM Web3D International Symposium*, ACM Press, 9–16.

GOSWAMI, P., EROL, F., MUKHI, R., PAJAROLA, R., AND GOBBETTI, E. 2013. An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. *The Visual Computer 29*, 1, 69–83.

HOPPE, H. 1997. View-dependent refinement of progressive meshes. In *Proc. SIGGRAPH*, 189–198.

ISTI-CNR VISUAL COMPUTING LAB, 2012. MeshLab for iOS: A powerful easy-to-use 3D mesh viewer for iPad and iPhone. www.meshpad.org.

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.

JOVANOVA, B., PREDA, M., AND PRETEUX, F. 2009. MPEG-4 Part 25: A graphics compression framework for xml-based scene graph formats. *Image Commun. 24*, 1-2 (Jan.), 101–114.

LEE, H., LAVOUÉ, G., AND DUPONT, F. 2009. Adaptive coarse-to-fine quantization for optimizing rate-distortion of progressive mesh compression. In *Proc. VMV*, 73–82.

LEE, J., CHOE, S., AND LEE, S. 2010. Compression of 3D mesh geometry and vertex attributes for mobile graphics. *Journal of Computing Science and Engineering 4*, 3, 207–224.

LUEBKE, D., AND ERIKSON, C. 1997. View-dependent simplification of arbitrary polygonal environments. In *Proc. SIGGRAPH*, 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.

MALVAR, H. S., SULLIVAN, G. J., AND SRINIVASAN, S. 2008. Lifting-based reversible color transformations for image compression. 707307–707307–10.

MARION, P., 2012. Point cloud streaming to mobile devices with real-time visualization. www.pointclouds.org.

MARTIN, G. N. N. 1979. Range encoding: an algorithm for removing redundancy from a digitised message. In *Video and Data Recording Conference*.

MEYER, Q., SUESSMUTH, J., SUSSNER, G., STAMMINGER, M., AND GREINER, G. 2010. On floating-point normal vectors. *Computer Graphics Forum 29*, 4, 1405–1409.

MEYER, Q., KEINERT, B., SUSSNER, G., AND STAMMINGER, M. 2012. Data-parallel decompression of triangle mesh topology. *Computer Graphics Forum 31*, 8 (Dec.), 2541–2553.

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.

PENG, J., KIM, C.-S., AND JAY KUO, C. C. 2005. Technologies for 3D mesh compression: A survey. *J. Vis. Comun. Image Represent. 16*, 6 (Dec.), 688–733.

PINTUS, R., GOBBETTI, E., AND CALLIERI, M. 2011. Fast low-memory seamless photo blending on massive point clouds using a streaming framework. *ACM Journal on Computing and Cultural Heritage 4*, 2, Article 6.

PURNOMO, B., BILODEAU, J., COHEN, J. D., AND KUMAR, S. 2005. Hardware-compatible vertex compression using quantization and simplification. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, 53–61.

ROSSIGNAC, J. 2001. 3D compression made simple: Edgebreaker with Zip&Wrap on a corner-table. In *Proceedings of the International Conference on Shape Modeling & Applications*, IEEE Computer Society, Washington, DC, USA, SMI '01, 278–.

SENECAL, J. G., LINDSTROM, P., DUCHAINEAU, M. A., AND JOY, K. I. 2004. An improved N-bit to N-bit reversible Haar-like transform. In *12th Pacific Conference on Computer Graphics and Applications*, 371–380.

TAUBIN, G., AND ROSSIGNAC, J. 1998. Geometric compression through topological surgery. *ACM Trans. Graph. 17*, 2 (Apr.), 84–115.

TAUBIN, G., GUÉZIEC, A., HORN, W., AND LAZARUS, F. 1998. Progressive forest split compression. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '98, 123–132.

WEISS, K., AND DE FLORIANI, L. 2010. Simplex and diamond hierarchies: Models and applications. In *Eurographics 2010 - State of the Art Reports*, Eurographics Association, Norrkping, Sweden, H. Hauser and E. Reinhard, Eds., 113–136.

XIA, J., AND VARSHNEY, A. 1996. Dynamic view-dependent simplification for polygonal models. In *Proc. IEEE Visualization*, 327–334.

YOON, S.-E., SALOMON, B., GAYLE, R., AND MANOCHA, D. 2004. Quick-vdr: Interactive view-dependent rendering of massive models. In *Proceedings of the conference on Visualization '04*, IEEE Computer Society, Washington, DC, USA, VIS '04, 131–138.