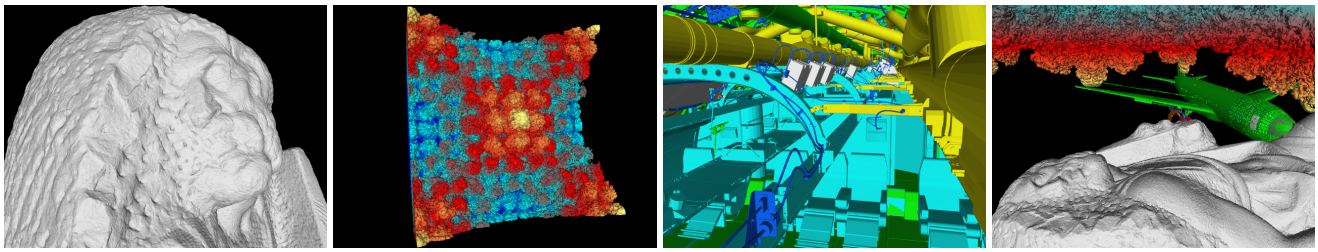
Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms

Enrico Gobbetti and Fabio Marton
CRS4 – Visual Computing Group *



(a) St. Matthew (372M triangles approximated with 1.5M voxels and 345K triangles)

(b) Richtmyer-Meshkov isosurface (472M triangles approximated with 4.2M voxels and 137K triangles)

(c) Boeing 777 (350M triangles approximated with 1M voxels and 3.4M triangles)

(d) All three datasets at once (1.2G triangles approximated with 4.0M voxels and 595K triangles)

Figure 1: **View-dependent rendering of extremely complex models.** These multi-gigabyte datasets cover a wide range of model classes. We can render them interactively at 1pixel tolerance on a Xeon 2.4 GHz PC with 1GB RAM and a NVIDIA GeForce 6800GT AGP8X graphics board.

Abstract

We present an efficient approach for end-to-end out-of-core construction and interactive inspection of very large arbitrary surface models. The method tightly integrates visibility culling and out-of-core data management with a level-of-detail framework. At preprocessing time, we generate a coarse volume hierarchy by binary space partitioning the input triangle soup. Leaf nodes partition the original data into chunks of a fixed maximum number of triangles, while inner nodes are discretized into a fixed number of cubical voxels. Each voxel contains a compact direction dependent approximation of the appearance of the associated volumetric subpart of the model when viewed from a distance. The approximation is constructed by a visibility aware algorithm that fits parametric shaders to samples obtained by casting rays against the full resolution dataset. At rendering time, the volumetric structure, maintained off-core, is refined and rendered in front-to-back order, exploiting vertex programs for GPU evaluation of view-dependent voxel representations, hardware occlusion queries for culling occluded subtrees, and asynchronous I/O for detecting and avoiding data access latencies. Since the granularity of the multiresolution structure is coarse, data management, traversal and occlusion culling cost is amortized over many graphics primitives. The efficiency and generality of the approach is demonstrated with the interactive rendering of extremely complex heterogeneous surface models on current commodity graphics platforms.

CR Categories: I.3.3 [Computer Graphics]: Picture and Image Generation—; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—.

Keywords: Out-Of-Core Algorithms, Level of Detail

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

1 Introduction

Many important application domains, including 3D scanning, computer aided design, and numerical simulation, require the interactive inspection of huge geometric models. Despite the rapid improvement in hardware performance, rendering today's multi-gigabyte datasets at interactive rates largely overloads the performance and memory capacity of state-of-the-art hardware platforms. To overcome this limitation, researchers have proposed a wide variety of output-sensitive rendering algorithms, i.e., rendering techniques whose runtime and memory footprint is proportional to the number of image pixels, not to the total model complexity. Very few techniques exist, however, that tightly integrate visibility culling and out-of-core rendering with level-of-detail management, limiting the applicability of the different approaches to only specific classes of objects. The lack of one of these techniques, or their independent application within a rendering engine, poses important problems when dealing with datasets that combine complicated geometry and appearance with a large depth complexity. Consider, for instance, the most advanced methods for rendering large scale models with fine geometric details [Yoon et al. 2004; Guthe et al. 2004; Cignoni et al. 2004], which are based on multiresolution point- or vertex- hierarchies constructed off-line through a geometric simplification process. Typically, these methods, that repeatedly merge nearby surface points or mesh vertices based on error minimization considerations, perform best for highly tessellated surfaces that are otherwise relatively smooth and topologically simple, since it becomes difficult, in other cases, to derive good "average" merged properties. Moreover, and most importantly, the off-line simplification process that generates the multiresolution hierarchy from which view-dependent levels of detail are extracted at rendering time is essentially unaware of visibility. When approximating very complex models, however, resolving the ordering and mutual occlusion of even very close-by surfaces, potentially with different shading properties, is of primary importance (see figure 2).

Main contributions. We present a new efficient approach for end-to-end out-of-core construction and view-dependent rendering of very large arbitrary surface models on commodity graphics platforms. The method tightly integrates visibility culling and out-of-core data management with level-of-detail construction and rendering. The underlying idea is to depart from current point- or triangle-based multiresolution surface models and adopt a volumetric approach based on more complex rendering primitives. At preprocessing time, we generate a coarse volume hierarchy by a binary

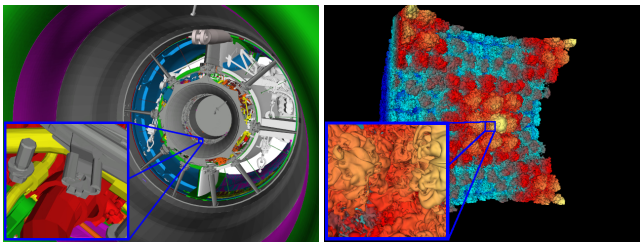


Figure 2: **Boeing 777 engine details (left) and isosurface details (right).** These kinds of object, composed of many loosely connected interweaving detailed parts of complex topological structure, are very hard to simplify effectively using off-line geometric simplification methods that do not take into account visibility.

space partitioning process of the input dataset. Leaf nodes simply split the original data into chunks of a fixed maximum number of triangles, while inner nodes are discretized into a fixed number of cubical voxels. Each voxel contains a compact direction dependent approximation of the appearance of the associated volumetric subpart of the model when viewed from a distance. The approximation is constructed by a visibility aware algorithm that fits parametric shaders to samples obtained by casting rays against the full resolution dataset. At rendering time, the volumetric structure, maintained off-core, is refined and rendered in front-to-back order, exploiting vertex programs for GPU evaluation of view-dependent voxel representations, hardware occlusion queries for culling occluded subtrees, and asynchronous I/O for avoiding out-of-core data access latencies. Since the granularity of the multiresolution structure is coarse, data management, traversal and visibility culling costs are amortized over many graphics primitives, and disk/CPU/GPU communication can be optimized to fully exploit the complex memory hierarchy of modern graphics PCs.

Advantages. The resulting technique, dubbed *Far Voxels*, has the following properties: it is applicable to a wide range of model classes, that include very detailed colored objects composed of many loosely connected interweaving detailed parts of complex topological structure; it is fully adaptive and is able to retain all the original topological and geometrical detail even for massive datasets; it is strongly GPU bound, since its coarse grained structure successfully exploits the batched rendering model of current commodity graphics platforms; multiresolution representations can be constructed in parallel with a out of core algorithm.

Limitations. As for all current large scale model rendering approaches, our method has also limitations: it has been designed for static models, editing is not supported; the sampling and fitting process is general and supports many model kinds, but at the expense of high preprocessing costs or aliasing problems if not enough rays are shot; the particular rendering primitives employed in this paper are phenomenological and appropriate mostly for diffuse materials – generalizing them is an important avenue of future work; the splatting method used for rendering does not correctly handle transparency; our current renderer implementation strives to maintain interactivity rather than ensuring strict guarantees on image quality; we have currently not implemented out-of-core compression and speculative prefetching – these are orthogonal to our method, but important to reduce perceived refinement latency.

Despite these limitations, the current method and prototype system is of immediate practical use and provides unprecedented performance in rendering very large complex models. As highlighted in section 2, while certain other methods share some of *Far Voxel*'s advantages, they typically do not meet its capability in all of the areas. A general overview of our approach is presented in section 3, while section 4 introduces a general out-of-core technique for constructing the multiresolution model, and section 5 describes view-dependent refinement and rendering algorithms. The efficiency and generality of the approach has been successfully evaluated with extremely complex CAD, isosurface, and scanned models (section 6).

2 Related Work

Rapidly rendering adaptive representations of large models is a very active research area. In the following, we will discuss the approaches that are most closely related with our work. Readers may refer to recent surveys (e.g., [Chiang et al. 2003; Cohen-Or et al. 2003]) for further details.

Out-of-core view-dependent simplification. The vast majority of view-dependent simplification methods for general meshes are based on constructing a graph of possible refinement/coarsening operations at the point, vertex, or triangle level. Up until recently, most techniques required an incore preprocessing step, even though rendering was performed out-of-core. QSplat [Rusinkiewicz and Levoy 2000], based on a out-of-core hierarchy of bounding spheres traversed at run-time to generate point splats, was the first fully out-of-core point-based method, while Lindstrom's [2003] scheme based on vertex clustering on a rectilinear octree was the first fully out-of-core mesh-based technique. As for all classic adaptive rendering techniques, these methods spends a great deal of rendering time to compute the view-dependent representation and do not scale well to gigantic meshes. A number of authors have thus proposed various ways to push the rendering performance limits in particular situations. The randomized z-buffer [Wand et al. 2001] uses a hierarchical traversal of a structure where the leaf nodes contain arrays of random point samples. They focus on scenes with many instances of simple objects. Stamminger and Drettakis [2001] dynamically adjusts the point sampling rate for rendering complex procedural geometry at high frame rates, but they require a parameterization of the model, while we focus on arbitrarily complex environments. A number of authors have recently proposed techniques reducing the per-primitive workload by using coarse multiresolution structures that compose at run-time pre-assembled optimized primitive groups [Erikson et al. 2001; Levenberg 2002; Cignoni et al. 2003; Yoon et al. 2004; Guthe et al. 2004; Cignoni et al. 2004; Gobbetti and Marton 2004]. We also follow this approach. All the mentioned methods employ, however, error metrics defined on the boundary surface of the objects, and are thus optimized for small numbers of high-complexity, densely-tessellated objects, rather than for many objects forming a topologically rich assembly. A number of authors have proposed topology reducing techniques using intermediate volumetric representations (e.g, [Andujar et al. 2002]). These methods, however, are view independent, and have been applied mostly to off-line mesh simplification.

Visibility culling. Visibility culling approaches are broadly classified into from-point and from-region visibility algorithms [Cohen-Or et al. 2003]. From-region algorithms compute a potentially visible set (PVS) for cells of a fixed subdivision of the scene and are applied offline in a preprocessing phase. From-point algorithms are applied online for each particular viewpoint. For general environments, accurate PVSs are hard to compute [Bittner et al. 2004]. Image-based occlusion representations are thus widely used, and the most recent algorithms exploit graphics hardware to perform online visibility culling [Bittner et al. 2004; Yoon et al. 2004; Zhang et al. 1997; Klosowski and Silva 2001]. Our method combines an off-line phase, integrated with level-of-detail generation, that removes voxels practically always occluded, with an on-line phase that exploits hardware occlusion queries to perform coarse hierarchical visibility culling. Spatiotemporal coherence is exploited as in [Bittner et al. 2004] to optimize the scheduling of queries. Few approaches exist that integrate LODs with occlusion culling both in the construction and rendering phases. Notable exceptions are hardly visible sets [Andujar et al. 2000], and visibility guided simplification [Zhang and Turk 2002], which, however, are non-conservative techniques that favor model simplification in areas that are likely to be occluded.

Hybrid rendering approaches. Many hybrid algorithms have been proposed that combine multiple techniques to render massive models. A representative example is the MMR/Gigawalk system [Aliaga et al. 1999; Govindaraju et al. 2003], which combines static LODs, HLODs [Erikson et al. 2001] and image based impostors with occlusion culling and out-of-core computation and is applicable to large CAD models that can be naturally partitioned into rectangular cells. El-Sana et al. [2001] and the iWalk system [Corrêa et al. 2003] combined view-dependent rendering with approximate occlusion culling for highly occluded scenes. Chen and Nguyen [Chen and Nguyen 2001], Cohen et al. [2001] and Guthe et al. [2004] presented methods for combining multiresolution polygon and point rendering applicable densely sampled surfaces. These systems are optimized for particular model classes, and are hardly applicable to extremely detailed models with variable degrees of occlusion. A few systems based on volumetric representations have been proposed. Wimmer et al [2001] followed an approach similar to ours for data resampling, but focused on encoding the appearance of pre-shaded models as seen from a view cell. Livnat and Tricoche [2004] presented an isosurface extraction technique that, similarly to our work, approximates internal nodes of a LOD BSP tree with a point when a node projects to less than a pixel. Decaudin and Neyret [2004] also used a volumetric approach, but their system is specialized to procedural forest scenes obtained by repeated instancing of volumetric texture tiles.

Interactive ray-tracing approaches. An alternative to rasterization is to use ray tracing techniques. Thanks to spatial indexing, ray queries can be answered in logarithmic time, and only those parts of the scene that are visible need to be accessed. A number of authors have thus designed raytracing systems for massive model visualization. Pharr et al. [1997] proposed a caching scheme that reorders the rays in a way that minimizes disk I/O. DeMarle et al. [2004] presented parallel techniques for efficiently distributing models and tasks on a large distributed cluster. Wald et al. [2004] presented a heavily optimized system that exploits SIMD instructions for rendering large polygonal models on a dual Opteron PC with 6GB RAM. A small number of in-core volumetric proxies is exploited for representing not-yet-loaded geometry in order to hide disk I/O latencies. While the results obtained with this method are impressive, achieving 3-7 fps at video resolution for the Boeing 777 dataset, the lack of multiresolution data forces the algorithm to access large parts of the dataset for producing a single frame (e.g., over 2GB for a 640x480 1 ray/pixel overview of the Boeing 777 dataset). Moreover, the method does not employ GPU acceleration. By fully exploiting current GPUs, we demonstrate that it is possible to achieve nearly one order of magnitude higher frame rates on lower end 32 bit machines. Given current CPU/GPU performance trends, this gap is likely doomed to widen.

3 Multiresolution model overview

Our approach, that exploits the programmability and batched rendering performance of current GPUs, is based on the idea of moving the grain of the multiresolution surface model up from points or triangles to small volumetric clusters, which represent spatially localized dataset regions using groups of (procedural) graphics primitives. The clusters provide the capability of performing coarse-grained view-dependent refinement of the model and are also used for on-line visibility culling and out-of-core rendering.

Figure 3 provides an overview of the approach. To generate the clusters, the model is hierarchically partitioned with a axis-aligned BSP tree. Leaf nodes partition full resolution data into fixed triangle count chunks, while inner nodes are discretized into a fixed number of cubical voxels arranged in a regular grid.

Finding a suitable voxel representation is challenging, since a voxel region can contain arbitrarily complex geometry. To simplify

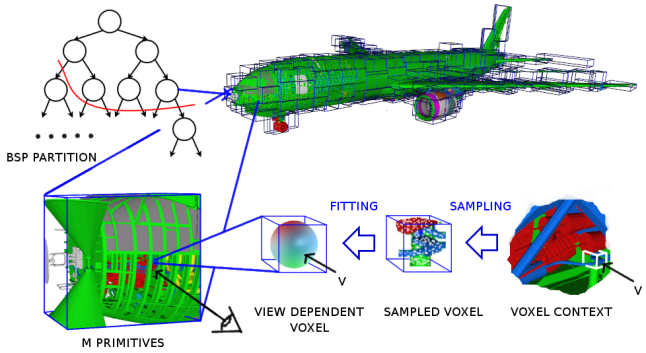


Figure 3: **Multiresolution structure overview.** The model is hierarchically partitioned with a axis-aligned BSP tree. Leaf nodes are rendered using the original triangles, while inner nodes are approximated using view-dependent voxels.

the problem, we assume that each inner node is always viewed from the outside, and at a distance sufficient to project each voxel to a very small screen area (say, below one image pixel). This constraint can be met with a suitable view-dependent refinement method, that refines the structure until a leaf is encountered or the image of each voxel is small enough (see section 5). Under this condition, a voxel always subtends a very small viewing angle, and a purely direction dependent representation of shading information is thus sufficient to produce accurate visual approximations of its projection. This approximation is similar in spirit to the lumispheres [Wood et al. 2000] employed for surface light field rendering, except that we do not associate shaded illumination samples to every ray originating from a surface, but rather prefiltered voxel shading information.

To construct a view-dependent voxel representation, we employ a visibility aware sampling and reconstruction technique detailed in section 4. We first acquire a set of shading information samples by ray casting the original model from a large number of appropriately chosen viewing positions. Each sample associates a reflectance and a normal to a particular voxel observation direction. We then compress these samples to an analytical form that can be compactly encoded and rapidly evaluated at run-time on the GPU to compute voxel shading given a view direction and light parameters.

4 Construction

The off-line component of our method constructs a space partitioned multiresolution structure starting from the full resolution model, that we assume, without loss of generality, represented as a *triangle soup*, i.e., a flat list of triangles with direct vertex information.

4.1 Mesh partitioning and deep BSP construction

The first preprocessing phase consists in spatially partitioning the scene according to an axis aligned BSP tree, whose root coincides with the mesh bounding box and whose leaves contain less than a small predefined number of mesh triangles. The BSP tree constructed in this phase will be exploited to accelerate the ray casting process and serves as a basis for constructing the multiresolution volumetric structure. The BSP is constructed out-of-core according to the surface-area heuristic [MacDonald and Booth 1990], which produces subdivisions that closely encompass the model, and stored in memory coherent order [Havran 1999]. Each BSP leaf points to the list of triangles that pass trough it, in a format optimized for ray-triangle intersection [Arenberg 1988]. The final structure, stored on disk in binary form, can be used for out-of-core sampling of the scene, by mapping it to the user address space and letting the OS manage demand loading on a per-page basis.

4.2 Level-of-detail hierarchy construction

Once the deep BSP tree is available, we exploit it for constructing the level-of-detail structure for the scene. We first generate the final

multiresolution structure layout by constructing a coarse hierarchical structure on top of the deep BSP hierarchy. In this layout, empty nodes are removed, while leaf nodes are associated with subtrees of the deep BSP that approximately contain a number of triangles corresponding to the specified cluster size. We then traverse the coarse hierarchy and generate for each node the final representation, which is immediately stored on disk. Nodes are visited level by level, and, at each level, in order of geometric proximity. To obtain that, the nodes at a given level are sorted by increasing Morton code of their center point. This traversal order optimizes memory coherence at construction time, and even more importantly, at rendering time. To further increase memory coherence, the hierarchical data structure is split on disk into two files: an index tree and a data repository. The index tree has a small footprint, since it contains, for each node, just the data required for traversal (bounding box, voxel size, and index of the two children), and refers to the associated rendering data through a 64 bit offset into the repository and a 32 bit size. The repository contains all the data required for rendering the node.

Generating rendering data is the main construction task. In our approach, each leaf node is encoded as a generalized triangle strip covering the geometry contained in it, thus retaining all the original topological and geometrical detail. To generate the strip, triangles are extracted from the deep BSP subtree associated to the generated node and clipped to the node's bounding box. Connectivity is reconstructed from independent triangles by vertex position hashing. Non-leaf nodes contain instead a volumetric simplification, generated from a discretization of the bounding box into a fixed number of (approximately) cubical voxels arranged in a regular grid.

Sampling process. To construct a view-dependent voxel representation, shading information samples are acquired by casting rays against the deep BSP of the original model from a large number of possible viewing position. To generate the samples, we exploit the fact that a given node will be always viewed from a distance sufficient to have each voxel subtend a very small viewing angle. We provide the preprocessor with a worst case value θ_{max} for this angle, which will correspond to the coarsest possible accuracy for view-dependent rendering (see section 5). Given the voxel radius r_{vox}, we derive the minimum viewing distance $d_{min} = r_{vox}/\tan(\theta_{max}/2)$, and cast against the deep BSP a large number of random rays originating on the surface S at distance d_{min} from the node's bounding volume B and randomly directed toward it (see figure 4). For each hit position inside B, we compute the voxel index, and store in an associated list a record containing the intersected surface reflectance, its normal, and the ray direction. Hits outside B are simply discarded. At the end of the process, each potentially visible voxel will be associated to a list of samples covering all the unoccluded directions. An important property of the sampling strategy is that, given a sufficiently large number of rays, visibility problems are conservatively solved for arbitrary view positions outside S by using information from the entire environment enclosed by S. In particular, voxels that do not contain surface boundaries, as well as voxels that are fully occluded, will be empty. At the same time, for each voxel, only visible surfaces are sampled, and only from those directions where they can be seen. As demonstrated in section 6, this visibility aware sampling strategy is extremely effective for complex models, as environmental occlusion leads to eliminate a large portion of the voxels (over 40% for the Boeing 777 dataset) and minimizes artifacts due to leaking of occluded objects colors through nearby occluding surfaces.

This ray casting based sampling strategy is very general and supports many model kinds, but at the expense of high preprocessing costs or aliasing problems if not enough rays are shot. For simplicity of implementation, we decided to cast a fixed number of rays per volume, even though classic raytracing sampling methods that cast more rays in high variance regions could definitely be used to improve accuracy. A promising direction to reduce processing times while keeping aliasing problems under control would be to construct the structure in a bottom-up fashion, and cast rays against the already constructed multiresolution structure instead of sampling the original model at each node.

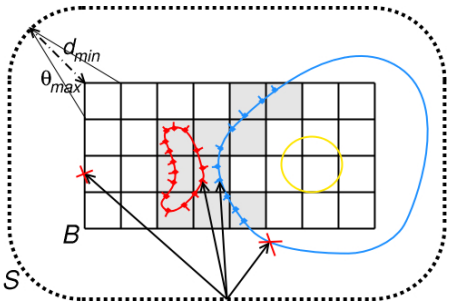


Figure 4: **Sampling process.** Shading information samples are acquired by ray casting the model from a large number of possible viewing positions at distance d_{min} from the volume. Environmental occlusion is taken into account to remove always occluded voxels arnd to restrict the sampling to potentially visible surfaces. In the image, the blue object hides the yelloow one, and only gray voxels are considered non-empty.

Parametric shader compression. The sample list for each voxel provides all the information for computing view-dependent shading. To generate a more compact and efficient representation, we compress these samples by fitting them to simple parameterized shader models, and choosing the shader that provides the best approximation. Each shader consists in a function that returns a color attenuation given its internal parameters, a view direction $\mathbf{v}$ and a light direction $\mathbf{l}$, i.e., $Shader_i(\mathbf{v},\mathbf{l}) = BRDF_i(\mathbf{v},\mathbf{l})\max(\mathbf{n}(\mathbf{v})\cdot\mathbf{l}),0)$, where $\mathbf{n}(\mathbf{v})$ is the surface normal seen from $\mathbf{v}$. Instead of deriving a general purpose shader, we assume that a small number of shader classes can be used to model common situations (see figure 5). In our current prototype, we have implemented the following classes:

K1a A flat shader, parameterized by a plane normal, a front material, and a back material, that implements the standard two-sided Lambert reflection model. The normal is found by averaging all sampled normals, while the colors are found as the front and back average.

K1b The same as above, with the normal computed by principal component analysis of the sampled hit positions.

K2 A smooth shader, parameterized by 6 reflectance and 6 normal control points associated to the main viewing directions $(\pm x, \pm y, \pm z)$, found by averaging sampled values according to the associated ray direction. Lambert shading parameters are found from v by summing the values at the three nearest reference directions weighted by the respective direction cosine.

The shader selected for a particular voxel is found by constructing an instance of each shader class k using a random subset of the samples. We then use the remaining samples to measure the difference in shading for a number of random light directions $\mathbf{l}_j$: $\varepsilon^{(k)} = \sum_i \sum_j \left(BRDF_i^{(sampled)}(\mathbf{v}_i,\mathbf{l}_j)\max(\mathbf{n}_i\cdot\mathbf{l}_j,0) - Shader^{(k)}(\mathbf{v}_i,\mathbf{l}_j)\right)^2$. The shader instance with minimum error is then selected and its representation is stored.

Even though the shader sampling and fitting method is very general, the particular shaders described here are phenomenological and appropriate mostly for diffuse materials. They have been chosen mostly because of ease of implementation and efficiency of computation. Generalizing them is an important avenue of future work.

Output data encoding. Since our current renderer is based on voxel splatting and vertex shaders (see 5), we only store non-empty voxels, and encode each shader parameter in a specific vertex array,

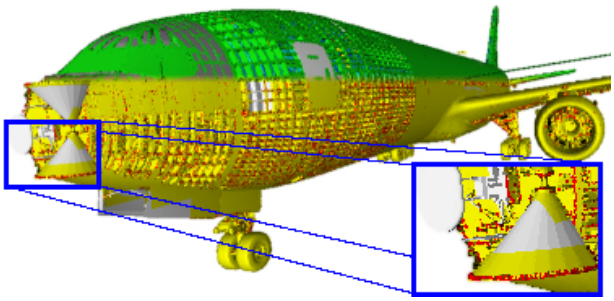


Figure 5: **Primitive distribution.** The top part of the image shows the model as presented to the viewer. The bottom part illustrates the primitive class distribution using a color code: red for K2 shaders, which tend to accumulate on complex voxels, yellow for K1 shaders, which tend to accumulate on almost planar surfaces, and white for triangles, used for full resolution leafs.

much in the same way we encode triangle strip data for leaf nodes (see figure 6). In our current implementation, we do not compress data to a compact external format, but rather directly encode each value in OpenGL format, using 4 bytes for colors, 6 bytes for normals, and 6 bytes for relative positions inside a box. Exploiting compression techniques for reducing disk usage and I/O needs is a major avenue for future work.

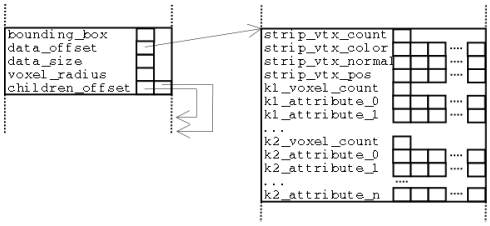


Figure 6: **Output data encoding.** All data required for rendering is encoded in vertex array format.

Parallel out-of-core construction. The hierarchy construction phase dominates, by far, the overall processing cost, mainly because of the scene sampling process. The whole process is however inherently massively parallel. For this paper, we have adopted the simple solution of separating the input models into chunks of 20-30M triangles each with a very coarse BSP partitioning and to distribute the chunks to N machines that execute in parallel, and out-of-core, the rest of the preprocessing. As a result, we obtain a forest instead of a single tree.

5 View-dependent rendering

At rendering time, the volumetric structures, maintained off-core, are refined and rendered in front-to-back order, exploiting vertex programs for GPU evaluation of view-dependent voxel representations, hardware occlusion queries for culling occluded subtrees, and asynchronous I/O requests for avoiding out-of-core data access latency.

Refinement algorithm. The user selected pixel threshold is the value that drives the refinement: this value represents the maximum required projected voxel size on the screen. The algorithm takes as input a forest of multiresolution hierarchies and performs a breadth-first front-to-back traversal, making use of the following data structures: a *node priority queue*, for sorting visited nodes in front-to-back order; an *occlusion query queue*, for storing pending occlusion queries; a *GPU cache*, based on OpenGL's *Vertex Buffer Objects* extension, for storing the most recently *rendered* nodes; a *RAM cache*, for storing the most recently *visited* nodes; a *fetch request priority queue*, for storing asynchronous I/O requests for nodes not yet available.

The traversal is initiated by inserting the roots of each tree into the node priority queue, and, if this is the first frame, into the RAM cache. We then iteratively remove and process the highest priority node from the node priority queue or from the occlusion query queue until both queues are empty.

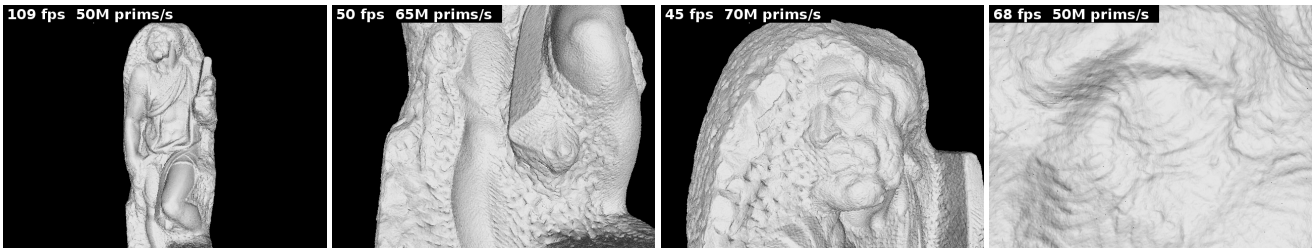
Each time a node is extracted from the node priority queue, we mark it by default as invisible for the current frame. We then check whether its bounding box falls totally outside the view volume. If so, we simply stop with this node, culling away its entire subtree. If instead the node is at least partially within the view frustum, we test whether to stop refinement, either because the node is a leaf, or its projected voxel size falls below the screen space threshold, or its children are not yet present in the RAM cache. In that case, the node is rendered, while issuing a hardware occlusion query and storing it in the occlusion query queue. If data is missing, fetch requests for missing children are pushed in the fetch request priority queue. When continuing refinement, we test whether the node was marked visible at the previous frame. If so, we avoid testing for occlusion and immediately push its children in the node priority queue. For previously invisible nodes, on the other hand, we issue an occlusion query for their bounding box, which is then stored in the query queue.

The nodes queried for occlusion are processed only as soon as the result for their occlusion query is available or there are no other nodes to traverse. Each time an item is extracted from the query queue, we check the occlusion query result. If the number of visible pixels returned by the query is zero, we simply stop with this node, culling away its entire subtree because of occlusion. If instead some pixels were visible, we mark the node and its ancestors as visible. If the node was not already rendered, it is because it is an inaccurate non-terminal node with RAM cached children. We thus continue refinement by pushing its children in the node priority queue.

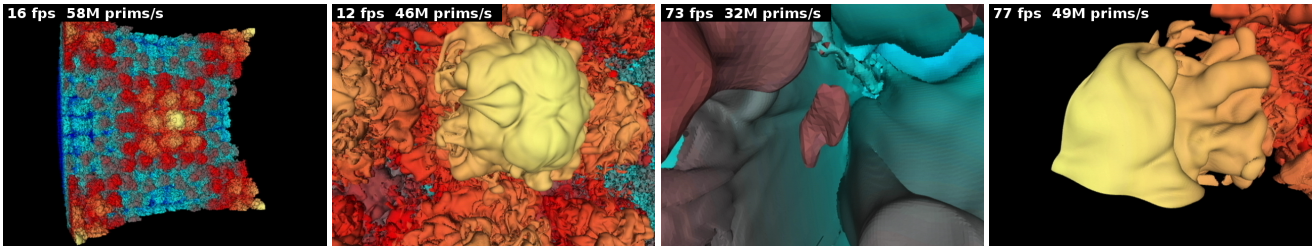
Since the structure is coarse grained and the refiner never stalls because of I/O requests the method is GPU bound.

Node rendering. Our current implementation renders view-dependent voxels using a splatting method that draws an antialiased OpenGL point primitive per voxel. At renderer initialization, vertex shader programs specific to each primitive class are compiled and loaded on board. At node rendering time, we iterate on all possible primitive representations. If the number of voxels/vertices for the given primitive class is non-null, we bind the associated program, load the vertex attribute data into the appropriate program parameters, and issue a `glDrawArrays` to draw all voxels at once. Triangle strip rendering code for leaf nodes follows the same pattern. To minimize bus traffic, each time a node is rendered, we reuse the GPU cached version if present, otherwise we render it and cache its representation in place of the oldest one. In the current implementation, the primitives that compose a particular node are rendered in an arbitrary order, and without taking into account their opacity. This provides a rendering quality similar to that of one-pass point splatting methods, which is sufficient for our target applications, but limits our ability to correctly treat high resolution textures and transparency. We are currently exploring ways to use a texture-based volume rendering approach to solve this problem.

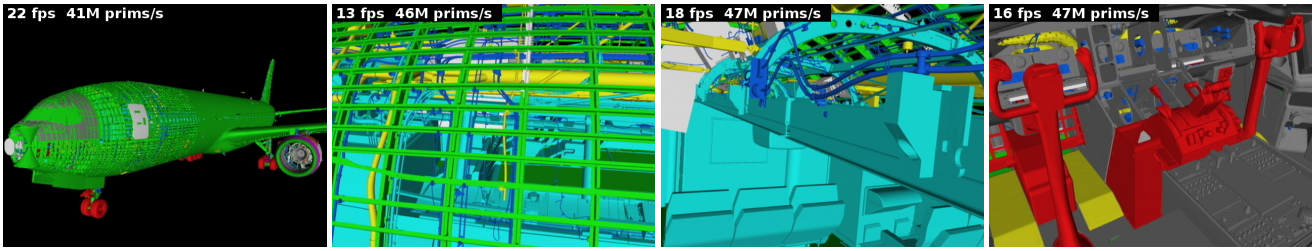
Asynchronous I/O. Similarly to Streaming QSplat [Rusinkiewicz and Levoy 2001], the fetch request queue is traversed in order of priority at the end of the frame, issuing only as many requests as those allowed by the estimated I/O bandwidth, and ignoring the remaining ones. In our current implementation, the priority of a fetch request is given by the node's parent projected voxel size. Access to the data repository is made through a data access layer, that hides from the renderer whether data is local or remote. This layer internally uses memory mapping primitives for local data, and a TCP/IP protocol for remote data. It makes it possible to asynchronously move in-core a node by fetching it from the repository, to test whether a node is immediately available, and to move available nodes to the RAM cache.



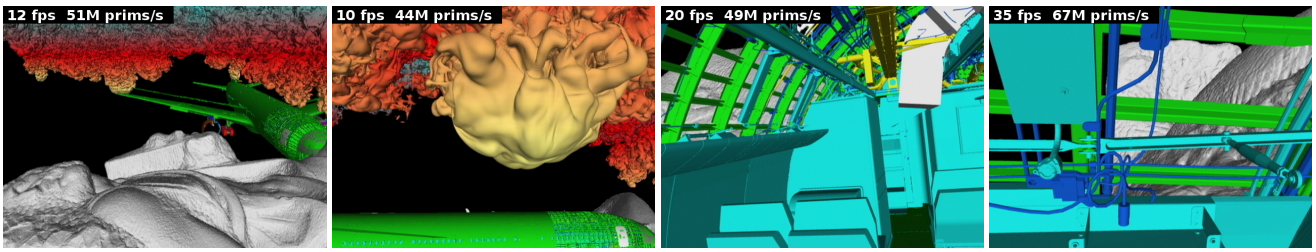
(a) St. Matthew 0.25mm (372M triangles)



(b) Richtmyer-Meshkov Isosurface (472M triangles)



(c) Boeing 777 (350M triangles)



(d) All models at once (1.2G triangles)

Figure 7: **Inspection sequences: selected frames.** All images were recorded live on a Xeon 2.4 GHz PC with 1GB RAM and a NVIDIA GeForce 6800GT AGP8X graphics board using a 1 pixel tolerance.

6 Results

An experimental software library and a rendering application supporting the technique have been implemented on Linux using C++ with OpenGL. We have extensively tested our system with a number of large surface models. The quantitative and qualitative results discussed here are restricted to the three models of figure 7, that possibly represent the most complex benchmark test cases in their respective domains: the St. Matthew 0.25mm dataset (372M triangles) is a very dense high resolution laser scanning model; the Richtmyer-Meshkov Isosurface dataset (472M triangles) is a very convoluted mesh with holes, a huge depth complexity, and a high genus generated from a very high resolution 3D simulation of Richtmyer-Meshkov instability and turbulence mixing; the Boeing 777 dataset (350M triangles) is an exceptionally complex CAD model, composed of many loosely connected interweaving detailed parts of complex topological structure represented as meshes of colored triangles with widely varying aspect ratios.

Preprocessing. Table 1 lists numerical results for our out-of-core preprocessing method for all the test datasets. The tests were executed on a moderately loaded network of 16 PCs running Linux 2.4. Each PC has two CPU Athlon 2200+ CPUs, 1GB DDR memory, a 70GB ATA 133 hard disk, and a Ethernet 100 Mb/s network connection. We constructed all multiresolution structures with a prescribed maximum leaf size of 8K triangles/node, a prescribed non-leaf discretization size of 16K voxels/node, 256K rays/node for sampling, and a maximum voxel viewing angle $\theta_{max} = 0.5$ degrees.

Models	Input file		Deep BSP		Far Voxels			
	M Tri	Size GB	Time sec	Size GB	Time sec	Size GB	Avg.Tri/Leaf	Avg.Vox/Node
St. Matthew	372	14.5	4706	18.6	14592	10.6	4876	1953
Boeing 777	350	13.7	8201	14.9	16461	14.9	5328	2401
Isosurface	472	18.4	5648	26.1	23751	16.1	5822	3425

Table 1: **Numerical results for out-of-core construction.** Tests performed on a network of 16 PCs.

Overall processing times range from about 1K input triangles/s for 1 CPU to 20K input triangles/s for 16 CPU. By contrast, processing times for competing multiresolution approaches based on geometric simplification range from 3K triangles/s [Yoon et al. 2004; Cignoni et al. 2004] for 1 CPU to 30K triangles/s on 16 CPUs [Cignoni et al. 2004]. The Far Voxels method is thus slower, even though it seems to scale better with the number of CPUs.

Our implementation requires on average 70MB per million

vertices. This is comparable to QVDR (88MB) [Yoon et al. 2004], but sensibly higher than the TetraPuzzles representation (32MB) [Cignoni et al. 2004]. Out-of-core model compression is a main avenue of future work.

To test the effectiveness of the strategy employed to cull occluded voxels at preprocessing time, we have also reconstructed the models without taking into account environment occlusion, i.e., by sampling each node separately without considering θ_{min}. The strategy proved very successful, since it removes over 25% of voxels for the isosurface dataset and 43% for the Boeing 777. The scanning model is essentially unaffected due to the low depth complexity.

Adaptive rendering. We evaluated the rendering performance of the technique on a number of inspection sequences on all test datasets, using a Linux PC with a Intel Xeon 2.4 GHz, 1GB RAM, two 70 GB ULTRA SCSI 320 hard drives, AGP 8x and NVIDIA GeForce 6800 GT graphics. The qualitative performance of our adaptive renderer is illustrated in an accompanying video.

The sessions were designed to be representative of typical inspection tasks and to heavily stress the system, and include rotations, rapid changes from overall views to extreme close-ups, and forced visibility discontinuities. To further illustrate the scalability of the method, we have included a test case showing the inspection of a scene containing all three models, for a total of over 1.2GB triangles and 41GB of out-of-core data distributed among the two PC disks. Table 2 lists numerical results for all the sessions.

Input			Window size	Pixel tolerance			Frames/s		Mprim/s	Max Resident Set Size MB
Models	M tri	Size GB		Target	Avg	Max	Min	Avg	Avg	
St. Matthew	372	10.6	640x480	1	0.8	2.4	9	45	51	102
Boeing 777	350	14.9	640x480	1	0.9	8.1	8	44	42	151
Isosurface	472	16.1	640x480	1	1.0	6.6	7	34	41	229
All	1194	41.6	640x480	1	0.9	12.2	6	20	42	172
All	1194	41.6	2x1024x768	1	1.1	18.0	3	20	40	218

Table 2: **Numerical results for out-of-core rendering.** Tests performed on a Linux PC with a Intel Xeon 2.4 GHz, 1GB RAM, two 70 GB ULTRA SCSI 320 hard drives, AGP 8x and NVIDIA GeForce 6800 GT graphics.

As illustrated in this table and in the accompanying video, our system is able to maintain interactivity in all test cases, while producing detailed images. As explained in section 5, our current renderer favors interactivity rather then ensuring strict bounds on accuracy, since we stop refinement and issue anynchronous I/O requests when data has to be fetched from disk. However, even though the maximum projected size has peaks caused by the rendering of coarser nodes when data is not immediately available, the average projected voxel size is, as expected, close (or below) the target tolerance (1 pixel), since due to temporal coherence only few nodes per frame cause RAM/GPU cache misses. The highest maximum values are for the larger models, which have longer load times due to increased disk footprint. The popping artifacts caused by rendering inaccurate nodes could be reduced by incorporating compression, speculative prefetching, or using disk RAIDs to serve the data.

During the entire inspection sequences, the resident set size of the application never exceeded 1.5% of the out-of-core data size, demonstrating the effectiveness of out-of-core data management.

We can sustain an average rendering rate of around 45M primitives/s independently of the model, counting as one primitive a single voxel or triangle. Throughput comparisons with classic tessellation techniques are difficult, since we employ heavier primitives than pure Gouraud shaded triangle strips. In our current unoptimized implementation, shading requires 19–28 vertex program instructions depending on primitive type. For simple models, such as the laser scanning test-case, it is likely that methods such as TetraPuzzles, which was able to sustain 70M tri/s on a GeForce Ultra FX 5800 GPU thanks to cache-coherent triangle strips [Cignoni et al. 2004], would provide better performance. Multiresolution mesh techniques would however be hardly applicable to the other examples demonstrated here. Even in the current implementation, the primitive rate is high enough to render over 4.5M primitives/frame at interactive rates. It is thus possible to use very small pixel thresholds, virtually eliminating popping artifacts without the need to resort to costly geomorphing techniques. The raw performance of the system is particularly useful for large scale display situations, where interactive raytracing solutions have problems meeting real-time constraints because of the large number of pixels to be covered. Figure 8 shows the largest scene examined on a large scale stereoscopic display assembled from off-the-shelf components, i.e., two 1024x768 DLP projectors connected to two outputs of the graphics card, polarizing filters with matching glasses, and a backprojection screen that preserves polarization. In this setting, a single PC is able to render two 1024x768 images per frame at an average of 20 Hz with a 1 pixel rendering tolerance, with a worst case of 3 Hz. Lowering the tolerance to 2 pixels increases the minimum frame rate to 9 Hz with little visual impact.

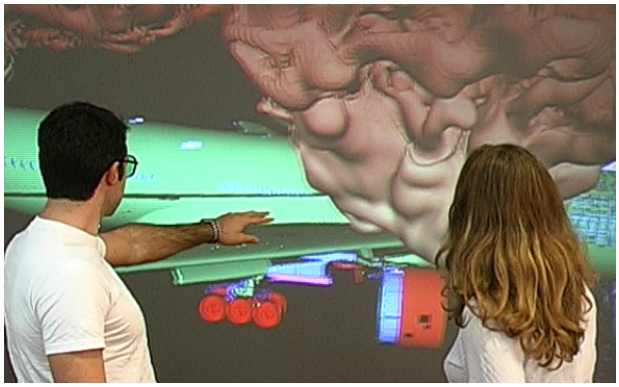


Figure 8: **Large scale stereoscopic display.** The largest test case (1.2G triangles) interactively inspected on a large scale stereoscopic display driven by single PC, which renders two 1024x768 images per frame with a 1 pixel tolerance.

Network streaming. Some network tests have been performed on all test models, on a local area network at 100Mb/s using the TCP/IP protocol to access the data. As illustrated in the video, rendering rate remains the same as that of the local file version, but updates asynchronously arrive with increased latency. The effect is illustrated in figure 9, which shows the progressive refinement of the Boeing 777 dataset on a machine connected to a moderately loaded Linux box serving the models. Even though our current uncompressed model encoding is far from being optimal for the task, the application remains usable even for very large models on standard network connections.

7 Conclusions

We have presented an efficient technique for end-to-end out-of-core construction and view-dependent rendering of very large heterogeneous surface models on commodity graphics platforms. The main benefit of the method lies in its performance and applicability to a wide variety of model classes. As a result, we obtain an unprecedented spatiotemporal quality in the interactive inspection of massive models that exhibit complicated geometry and topology, heterogeneous material attributes, as well as large variations in depth complexity.

Besides improving the proof-of-concept implementation, we plan to extend the presented approach in a number of ways. In particular, we are currently working on the following aspects: compression of the output data representation to reduce disk usage and I/O latency; exploration of alternate view-dependent voxel representations; implementation of multi-resolution sampling methods; exploration of higher quality texture-based volume rendering methods in place of the current point splatter; prefetching and prediction of occlusion/visibility events to reduce artifacts due to the display of coarser than needed newly visible objects.

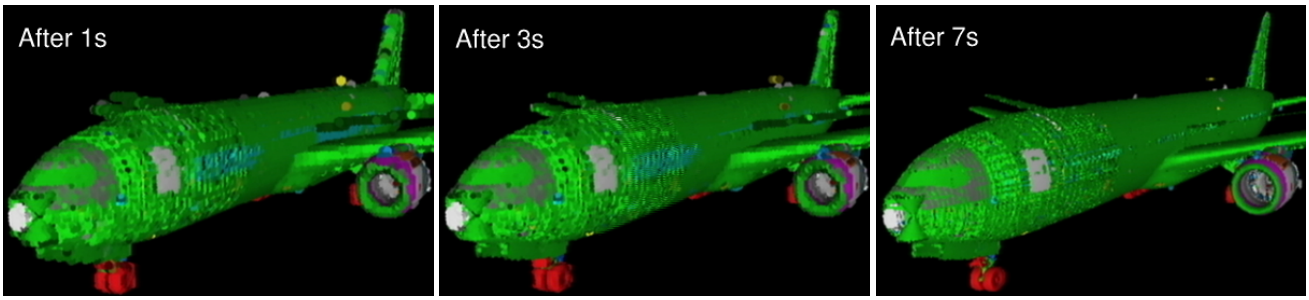


Figure 9: **Streaming.** Progressive refinement of the Boeing 777 dataset (350M triangles) on a 100Mb/s connection.

Acknowledgments. The source 3D datasets are provided by and used with permission of the Boeing Company, the Digital Michelangelo project, and the Lawrence Livermore National Laborarories. The authors would also like to thank Fabio Bettio (CRS4), Paolo Cignoni (ISTI-CNR), Francesca Frexia (CRS4), David Kasik (Boeing), Gianni Pintore (CRS4), and Alan Scheinine (CRS4).

References

ALIAGA, D., COHEN, J., WILSON, A., BAKER, E., ZHANG, H., ERIKSON, C., HOFF, K., HUDSON, T., STÜRZLINGER, W., BASTOS, R., WHITTON, M., BROOKS, F., AND MANOCHA, D. 1999. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *1999 Symposium on Interactive 3D Graphics*, 199–206.

ANDUJAR, C., SAONA, C., NAVAZO, I., AND BRUNET, P. 2000. Integrating occlusion culling and levels of detail through hardly-visible sets. *Computer Graphics Forum 19*, 3, 499–506.

ANDUJAR, C., BRUNET, P., AND AYALA, D. 2002. Topology-reducing surface simplification using a discrete solid representation. *ACM Trans. Graph. 21*, 2, 88–105.

ARENBERG, J., 1988. Re: Ray/triangle intersection with barycentric coordinates. Ray Tracing News, 1.

BITTNER, J., WIMMER, M., PIRINGER, H., AND PURGATHOFER, W. 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3, 615–624.

CHEN, B., AND NGUYEN, M. X. 2001. Pop: a hybrid point and polygon system for large data. In *Proc. IEEE Visualization*, 45–52.

CHIANG, Y.-J., EL-SANA, J., LINDSTROM, P., PAJAROLA, R., AND SILVA, C. T. 2003. Out-of-core algorithms for scientific visualization and computer graphics. IEEE Visualization 2003, Tutorial 4 Course Notes.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2003. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum 22*, 3, 505–514.

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 Transactions on Graphics 23*, 3 (August), 796–803. Proc. SIGGRAPH 2004.

COHEN, J. D., ALIAGA, D. G., AND ZHANG, W. 2001. Hybrid simplification: combining multi-resolution polygon and point rendering. In *VIS '01: Proceedings of the conference on Visualization '01*, IEEE Computer Society, 37–44.

COHEN-OR, D., CHRYSANTHOU, Y. L., AND SILVA, C. T. 2003. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9*, 3, 412–431.

CORRÊA, W. T., KLOSOWSKI, J. T., AND SILVA, C. T. 2003. Visibility-based prefetching for interactive out-of-core rendering. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 1–8.

DECAUDIN, P., AND NEYRET, F. 2004. Rendering forest scenes in real time. In *Proc. Rendering Techniques*, 93–102.

DEMARLE, D. E., GRIBBLE, C., AND PARKER, S. 2004. Memory-savvy distributed interactive ray tracing. In *Proc. Eurographics Symposium on Parallel Graphics and Visualization*, 93–100.

EL-SANA, J., SOKOLOVSKY, N., AND SILVA, C. T. 2001. Integrating occlusion culling with view-dependent rendering. In *VIS '01: Proceedings of the conference on Visualization '01*, IEEE Computer Society, 371–378.

ERIKSON, C., MANOCHA, D., AND BAXTER, W. 2001. HLODs for faster display of large static and dynamic environments. In *Proc. ACM Symposium on Interactive 3D Graphics*, 111–120.

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*, 6 (December).

GOVINDARAJU, N. K., SUD, A., YOON, S.-E., AND MANOCHA, D. 2003. Interactive visibility culling in complex environments using occlusion-switches. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, 103–112.

GUTHE, M., BORODIN, P., BALÁZS, A., AND KLEIN, R. 2004. Real-time appearance preserving out-of-core rendering with shadows. In *Proc. Eurographics Symposium on Rendering*. June, 69–79 + 409.

HAVRAN, V. 1999. Analysis of cache sensitive representations for binary space partitioning trees. *Informatica 29*, 3, 203–210.

KLOSOWSKI, J. T., AND SILVA, C. T. 2001. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics 7*, 4, 365–379.

LEVENBERG, J. 2002. Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings IEEE Visualization '02*, IEEE, 259–266.

LINDSTROM, P. 2003. Out-of-core construction and visualization of multiresolution surfaces. In *ACM 2003 Symposium on Interactive 3D Graphics*, 93–102,239.

LIVNAT, Y., AND TRICOCHE, X. 2004. Interactive point based isosurface extraction. In *Proc. IEEE Visualization*, 457–464.

MACDONALD, J. D., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 6, 153–165.

PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. 1997. Rendering complex scenes with memory coherent ray tracing. In *Proc. SIGGRAPH*, 101–108.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. QSplat: A multiresolution point rendering system for large meshes. In *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH 00)*, ACM Press, 343–352.

RUSINKIEWICZ, S., AND LEVOY, M. 2001. Streaming QSplat: A viewer for networked visualization of large, dense models. In *Symposium for Interactive 3D Graphics Proceedings*, 63–68.

STAMMINGER, M., AND DRETTAKIS, G. 2001. Interactive sampling and rendering for complex and procedural geometry. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, Springer-Verlag, 151–162.

WALD, I., DIETRICH, A., AND SLUSALLEK, P. 2004. An interactive out-of-core rendering framework for visualizing massively complex models. In *Proc. Eurographics Symposium on Rendering*, 81–92.

WAND, M., FISCHER, M., PETER, I., AUF DER HEIDE, F. M., AND STRASSER, W. 2001. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *SIGGRAPH 2001 Proceedings*, 361–370.

WIMMER, M., WONKA, P., AND SILLION, F. 2001. Point-based impostors for real-time visualization. In *Proc. Rendering Techniques*, 163–176.

WOOD, D. N., AZUMA, D. I., ALDINGER, K., CURLESS, B., DUCHAMP, T., SALESIN, D. H., AND STUETZLINGER, W. 2000. Surface light fields for 3d photography. In *Proc. SIGGRAPH*, 287–296.

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

ZHANG, E., AND TURK, G. 2002. Visibility-guided simplification. In *Proc. IEEE Visualization*, 267–274.

ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF III, K. 1997. Visibility culling using hierarchical occlusion maps. In *Proc. SIGGRAPH*, 77–88.