Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Graphics 28 (2004) 815–826
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models ☆

Enrico Gobbetti*, Fabio Marton

CRS4 Visual Computing Group, POLARIS Edificio 1, 09010 Pula (CA), Italy

Abstract

We recently introduced an efficient multiresolution structure for distributing and rendering very large point sampled models on consumer graphics platforms [1]. The structure is based on a hierarchy of precomputed object-space point clouds, that are combined coarse-to-fine at rendering time to locally adapt sample densities according to the projected size in the image. The progressive block based refinement nature of the rendering traversal exploits on-board caching and object based rendering APIs, hides out-of-core data access latency through speculative prefetching, and lends itself well to incorporate backface, view frustum, and occlusion culling, as well as compression and view-dependent progressive transmission. The resulting system allows rendering of complex out-of-core models at high frame rates (over 60 M rendered points/second), supports network streaming, and is fundamentally simple to implement. We demonstrate the efficiency of the approach on a number of very large models, stored on local disks or accessed through a consumer level broadband network, including a massive 234 M samples isosurface generated by a compressible turbulence simulation and a 167 M samples model of Michelangelo's St. Matthew. Many of the details of our framework were presented in a previous study. We here provide a more thorough exposition, but also significant new material, including the presentation of a higher quality bottom-up construction method and additional qualitative and quantitative results.

Keywords: Point-based graphics; Large datasets; Out-of-core algorithms; Level of detail

1. Introduction

Multiresolution hierarchies of point primitives have recently emerged as a viable alternative to the more traditional mesh refinement methods for interactively inspecting very large geometric models [2]. These methods are based on the assumption that for a wide class models, mainly arising from three-dimensional (3D) photography, 3D scanning, or numerical simulation, the sampling rate is so high that triangles are projected to very small screen areas at rendering time. Thus, the advantage of scan-line coherence are lost, and appropriately selected point samples are sufficient to accurately reproduce the model. One of the major benefits of this approach is its simplicity, stemming from the fact that there is no need to explicitly manage and maintain mesh connectivity during both preprocessing and rendering.

☆Expanded version of a paper presented at The First Eurographics Symposium on Point Based Graphics (Zurich, Switzerland, June 2004).

*Corresponding author.
E-mail addresses: gobbetti@crs4.it (E. Gobbetti), marton@crs4.it (F. Marton).
URL: http://www.crs4.it/vic/.

0097-8493/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2004.08.010

Unfortunately, current dynamic multiresolution algorithms for large models are very CPU intensive: nowadays, consumer graphics hardware is able to sustain a rendering rate of tens of millions of point primitives per second, but current multiresolution solutions fall short of reaching such performance. This is because the CPU is not able to generate/extract point samples from the out-of-core structure and send them fast enough to the graphics hardware in the correct format and through a preferential data path.

We have recently proposed a new breed of solutions for interactive and accurate visualization of very large surface models on consumer graphics platforms. The underlying idea of the proposed methods is to depart from current point- or triangle-based multiresolution models and adopt a cluster-based data structure, from which view-dependent representations can be efficiently extracted by combining precomputed surface chunks. Since each chunk is composed of a few thousands graphics elements, the multiresolution extraction cost is amortized over many graphics primitives, and CPU/GPU communication can be optimized to fully exploit the complex memory hierarchy of modern graphics platforms. We have demonstrated the performance of the approach with specialized methods for regional- [3] to planetary-scale [4] phototextured digital terrain models, as well as for general triangle meshes [5]. In this article, we focus on a simple point-based solution for high performance view dependent visualization of very large static point sampled models on consumer graphics platforms. In this case, we reduce the per-primitive structure overhead by moving the grain of the multiresolution model from a hierarchy of point samples to a hierarchy of precomputed object-space point clouds. At rendering time, the clouds are combined coarse-to-fine with a stateless top-down structure traversal to locally adapt sample densities according to the projected size in the image. The progressive block based refinement nature of the rendering traversal is well suited to hiding out-of-core data access latency, and lends itself well to incorporate backface, view frustum, and occlusion culling, as well as compression and view-dependent progressive transmission. The resulting system allows rendering of local and remote models of hundreds of millions of samples at high frame rates (over 60 M splat/second), supports network streaming and is fundamentally simple to implement.

Many of the details of this framework were presented in [1]. We provide here a more thorough exposition, but also significant new material, including the presentation of a higher quality bottom-up construction method and additional qualitative and quantitative results. Finally, we have attempted to further clarify the steps in our algorithms to facilitate their implementation and to make the transfer between abstract concepts and actual code as straightforward as possible.

The rest of the paper is organized as follows. Section 2 reviews related works. The details of the proposed data structure are presented in Section 3, while Section 4 describes algorithms for view-dependent refinement and rendering, and Section 5 propose a out-of-core technique for constructing the multiresolution model. The efficiency of the approach is demonstrated with the inspection of a number of very large models, including a massive 234 M samples isosurface generated by a compressible turbulence simulation, that exhibits a huge (>100) depth complexity, and a 167 M samples model of Michelangelo's St. Matthew (Section 6).

2. Related work

Point-based 3D graphics techniques for processing and rendering of dense models are an old idea [6,7], that has found many successful applications, including point-based modeling, high quality and interactive rendering, as well as coding and transmission of point-based models.

Our focus is the development of systems for the distribution and high speed interactive visual inspection of very large models on commodity graphics platforms. QSplat [2] is the reference system in this particular area. The system is based on a hierarchy of bounding spheres maintained out-of-core, that is traversed at run-time to generate points. This algorithm is CPU bound, because all the computations are made per point, and CPU/GPU communication requires a direct rendering interface, thus the graphic board is never exploited at his maximum performance. In Streaming QSplat [8], the QSplat data structure is subdivided into chunks, that are however only used for streaming objects over networks. The rendering procedure remains a hierarchical traversal executed on the CPU, with the additional book-keeping required to check the local availability of data. Kalaiah and Varshney [9] have recently proposed to improve the geometry bandwidth bottleneck by working on a compressed point sample geometry model obtained by principal component analysis. Even if they use a large cache of 40 M points, the need to regenerate a large number of small point clusters per frame from statistical information leads to a rendering speed which is roughly half the speed of QSplat. We exploit instead a partitioning of the model into clouds to improve the efficiency of CPU/GPU communication through a batched communication protocol and to support conservative occlusion culling for high depth complexity models. This provides at least an order of magnitude improvement in rendering rate on current commodity graphics platforms. The superior CPU/GPU communication efficiency of this approach is confirmed by recent benchmarks by Sainz et al. [10], that recently compared a number of state-of-the-art hardware

accelerated point rendering algorithms, measuring a maximum throughput of about 10 M rendered points/s for small in-core models of up to 4 M samples on a GeForce FX 5900, while we can render out-of-core models of over 200 M samples at about 70 M points/s using a GeForce FX 5800U graphics board.

A number of authors have also proposed various ways to push the rendering performance limits in particular situations. The randomized z-buffer [11] uses a hierarchical traversal of a structure where the leaf nodes contain arrays of random point samples. They focus on large triangle meshes, while we work directly on dense point sampled models. Stamminger and Drettakis [12] dynamically adjusts the point sampling rate for rendering complex procedural geometry at high frame rates. They require a parameterization of the model, while we focus on unstructured point samples. Dachsbacher et al. [13] recently presented a hierarchical LOD structure for points that is adaptively rendered by sequential processing done on the GPU. They report a peak performance of over 50 M unfiltered points per second, which is similar to ours, but they are limited in the size of the rendered model, which must fit into the video card memory, while our work focuses instead on very large local and remote models. Moreover, their technique is not fully output-sensitive, since it does not support any visibility culling on the CPU before submitting the entire conservative range of points for the GPU for sequential processing. As a result, zoomed views of fine details of a large model are extremely unefficient [10].

There is a large body of work that aims at improving the rendering quality of point-sampled models. For dense models, these include using spheres [2], tangential disks [14,15], or high degree polynomials [16] instead of raw point primitives, as well as improving filtering in image space [15] or object space [17]. Such work is orthogonal to ours, which focuses on finding simple ways to improve raw rendering performance on very large models by amortizing costs on groups of many graphics primitives. Merging these two directions, possibly by exploiting GPU programming as in [18], is a main avenue for future work.

3. Multiresolution model

We assume that the input model is represented by a set of N sample points uniformly distributed over its surface, with an average spacing between samples equal to r. Each sample point is associated with a set of surface attributes, including position, normal, and possibly color information.

Our multiresolution approach creates a hierarchy over the samples of the datasets, simply by reordering and clustering them into point clouds of approximately constant size arranged in a binary tree. In other words, the final multiresolution model has exactly the same points of the input model, but grouped into chunks and organized in a level of detail representation. The root of the level of detail tree represents the entire model with a single cloud of $M_0 = M < N$ uniformly distributed samples. The remaining points are equally subdivided among the two subtrees using a spatial partition, with, again, M uniformly distributed points directly associated to the root of each subtree, and the rest redistributed in the children. The leaves are terminal clusters, which are further indivisible and whose size is smaller than the specified limit M.

Variable resolution representations of the models are obtained by defining a *cut* of the hierarchy and merging all nodes above the cut. This way, each node acts as a *refinement* of a small contiguous region of the parent. This is different from most other hierarchical schemes, where only the leaf nodes of the cut hierarchy are used.

The root node is the coarsest available model representation, with an average sample spacing of $r_0 = r\sqrt{N/M_0}$. Each node j, then, locally refines its parent by adding additional M_j samples to the representation, increasing the density in that region to a value r_j. This value can be computed off-line from the partitioned point cloud (see Section 5). By storing at each node j the value of r_j along with its point cloud, we can thus rapidly obtain a variable accuracy representation by traversing top down the hierarchy, while accumulating point clouds until the desired density is reached (see Fig. 1). Since we are interested in view-dependent

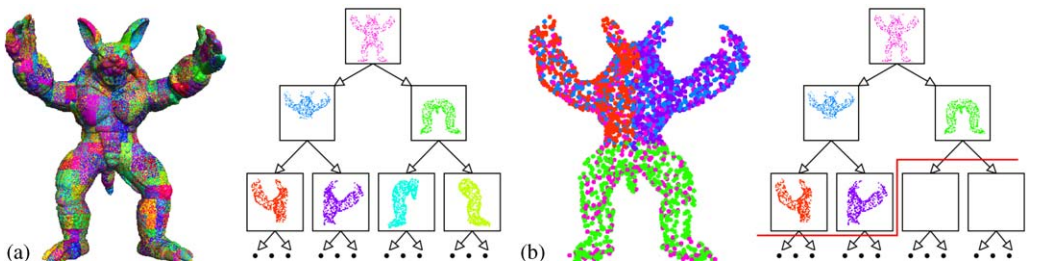

Fig. 1. Layered point cloud structure. The multiresolution model has exactly the same points of the input model, but grouped into constant size chunks and organized in a level of detail representation. Variable resolution representations of the models are obtained by defining a cut of the hierarchy and merging all nodes above the cut. (a) Full structure, and (b) adaptive resolution.

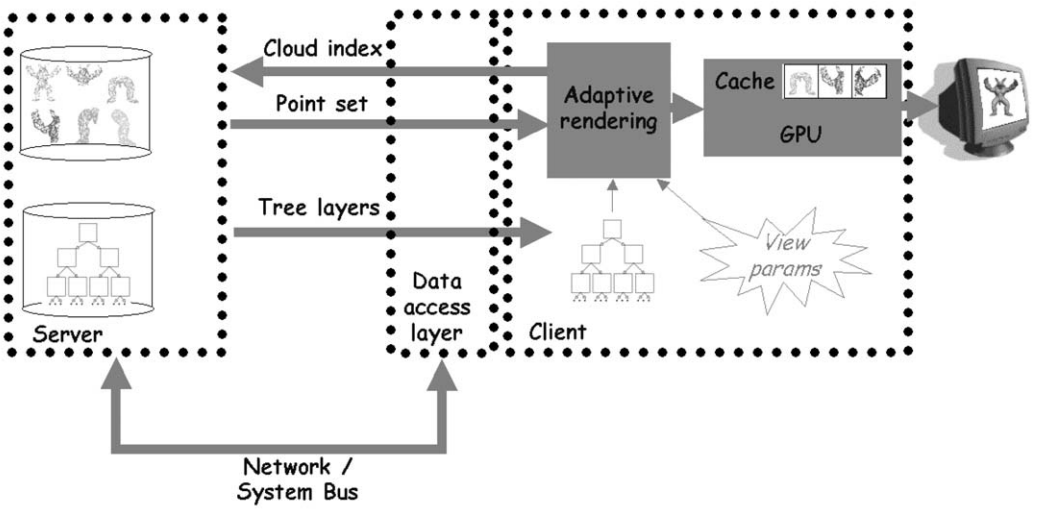

Fig. 2. The rendering pipeline. The client traverses the index tree coarse-to-fine in a view-dependent manner, requesting point clouds to the server. To maximize rendering performance and minimize traffic, point clouds are cached on board using a LRU strategy.

representations, we also precompute the bounding sphere and bounding cone of normals of each node. These are used for projecting the mean sample distance to the screen, as well as for view-frustum, backfacing, and occlusion culling (see Section 4).

The benefits of this approach are that the workload required for a unit refinement/coarsening step is amortized on a large number of point primitives, and that the small point clusters can be optimized off-line for best performance in host-to-graphics and network communication. By tuning the value of parameter M, we can vary the granularity of the structure from a total multi-resolution model (e.g., QSplat for $M = 1$) to a single-resolution model for point rendering ($M = N$). The choice of parameter M is dictated by performance considerations. In particular, if M is too large, the model becomes less adaptive, and switching from a resolution level to the next leads to a high latency. On the other hand, if M is too small, the model is more adaptive but CPU costs become non negligible. On current graphics platforms, we have empirically determined that the best performance trade-offs are obtained for values of M ranging from 512 to 8192.

4. The rendering pipe-line

Our adaptive rendering algorithm works on a standard PC, and data is assumed to be either locally stored on a secondary storage unit directly visible to the rendering engine or remotely stored on a network server (see Fig. 2).

4.1. Data layout and data access

The hierarchical data structure is split into an index tree and a point cloud repository. The index tree has a small footprint, since it contains, for each node, just the data required for traversal (sample spacing, bounding sphere, bounding cone of normals, and index of the two children), and refers to the associated point cloud through a 32 bit index that uniquely identifies the cloud in the repository. The repository is organized so that the data storage order reflects traversal order, which is coarse to fine and by physical position in space. We thus sort point clouds in the repository using as a primary key their tree level, and as a secondary key the Morton index of their bounding sphere center [19]. For disk storage and data transmission, each cloud is managed in compressed form. The point cloud is spatially sorted, then each attribute is quantized, delta encoded, and then entropy encoded with the LZO compressor.[1] Access to the point cloud repository is made through a data access layer, that masks to the application whether the repository is local or remote. This layer makes it possible to asynchronously move in-core a point cloud by fetching it from the repository, to test whether a point cloud is immediately available, and to retrieve its representation. We have implemented two versions of this access layer: the first one provides direct disk access through memory mapping functions and is used for

[1] LZO is a data compression library based on a Lempel Ziv variant which is suitable for data decompression in real-time. The library source is available from http://www.oberhumer.com/opensource/lzo/.

local files as well as remote NFS mounted files. The second one is based on the HTTP 1.1 protocol and, similarly to Streaming QSplat [8], fetches data from a standard HTTP server using range requests and permanent connections.

4.2. Progressive view-dependent refinement

The traversal algorithm, which extracts a view dependent representation of the multiresolution model from the current point of view, is based on a stateless coarse-to-fine refinement of our structure, that exploits the progressive nature and coarse granularity of the multiresolution hierarchy to reduce CPU refinement costs and to improve repository-to-host and host-to-graphics communication. In particular, asynchronous repository requests hide out-of-core data access latency, and communication with the GPU is made exclusively through a retained mode interface, which reduces bus traffic by managing a least-recently-used cache of point clouds maintained on-board as OpenGL *Vertex Buffer Object*. Pseudo-code for the method is listed in Fig. 3.

The user selected pixel threshold is the value that drives the refinement of the rendering algorithm: this value represents the required average sample distance between adjacent splats on the screen, and it is used as splat size. The refinement algorithm performs a single pass recursive traversal of the multiresolution structure. For each node, we use its bounding sphere and normal cone to test whether the node is totally outside the view frustum or totally backfacing. In this case, recursion stops, discarding the entire branch of the tree, otherwise we can render the node and, eventually, continue the refinement with its children. It is important to emphasize

```
lpc_refine(eye, node, threshold) {
  if visible(eye, node->sphere, node->normal_cone) {
    projected_size = project(eye,
                             node->sphere,
                             node->sampling_distance)
    if node is leaf {
      update_gpu_cache(node->cloud)
      set_point_size(projected_size)
      render(node->cloud)
    } else if any child is not present {
      for each child in children(node) {
        async_request(child)
      }
      update_gpu_cache(node->cloud)
      set_point_size(projected_size)
      render(node->cloud)
    } else {
      update_gpu_cache(node->cloud)
      set_point_size(threshold)
      render(node->cloud)
      if projected_size > threshold {
        for each child in children(node) {
          lpc_refine(eye, node, child)
        }
      }
    }
  }
}
```

Fig. 3. View-dependent refinement and progressive download. The user selected pixel threshold is the value that drives the refinement of the rendering algorithm.

that, differently from most other hierarchical refinement schemes, all visited nodes are rendered during the refinement traversal. Since we are focusing on high speed visualization, our current implementation simply uses OpenGL hardware supported points for point cloud rendering. This fact limits our ability to correctly treat texture and transparency. Using ellipsoidal splats computed on the GPU, as in, e.g., [18], would resolve these problems.

At node rendering time, we project the node's hierarchical average sample distance to the screen to obtain its splat size. A consistent upper bound on the projected size is obtained by measuring the apparent size of a sphere with diameter equal to the object space average sample distance and centered at the bounding sphere point closest to the viewpoint. If the projected splat size is less than the threshold, we render the node's point cloud with the prescribed splat size and stop recursion, otherwise a refinement is needed. In that case, to avoid blocking the renderer because of data access latency, especially in the case of rendering data over wide-area networks, we first check whether the node's children data is immediately available, i.e., if it is already in the GPU cache or considered in-core by the data access layer. If so, we continue recursion, otherwise recursion stops and the node is rendered with an increased splat size, equal to its projected mean sample distance, to cover holes left by children unavailability. Fetch requests are then pushed in a priority queue. Similarly to streaming QSplat [8], the 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 network bandwidth, and forgetting the remaining ones. Since the repository is sorted coarse to fine and by physical position in space, prioritizing the queue by node's index provides a simple compromise that is both I/O efficient and promises to download the most relevant data as soon as possible while being enough space coherent to minimize visual distraction.

4.3. Rendering on a budget

For interactive applications, it is often useful to have direct control on rendering time, instead of the control on rendering quality provided by prescribing a screen error tolerance for the refinement method. In addition to adjusting error tolerance per frame in a feedback loop, we can exploit the fact that our hierarchy is shallow to implement a predictive technique. Given a desired number of points per frame, we perform a binary search of the associated pixel threshold, by repeatedly traversing the index tree with the same refinement logic used for rendering, while only counting the number of generated primitives.

4.4. Occlusion culling

A number of complex dense models, such as large isosurfaces deriving from numerical simulation of turbulence (e.g., [20,21]) have an important depth complexity. For these models, efficiently culling the invisible portion of the rendered model is of primary importance to avoid uploading, refining, and rendering unnecessary data (see Fig. 4). Since our structure is coarse grained and provides a spatial partition, we can adapt to a point rendering framework visibility techniques developed for rendering scenes composed of many objects. Similarly to the approach introduced by Toon et al. [22] for complex CAD environments, our rendering algorithm exploits frame-to-frame coherence in occlusion culling, by using the set of visible point clouds from the previous frame as the occluder set for the current frame. At each frame, we render the object in three phases. In the first phase, we perform the usual refinement algorithm, but accumulate the clouds that would be rendered in a list of potentially visible objects, while only rendering the point clouds that were visible in the previous frame. In a second phase, we traverse the entire list of accumulated point sets, generating a hardware occlusion query for the object's bounding sphere (approximated by an icosahedron), using OpenGL ARB_occlusion_query extension to track the number of fragments that pass the depth test. In a third

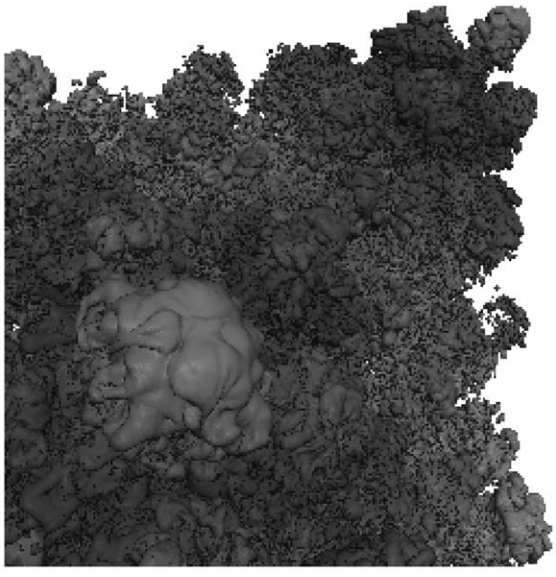

Fig. 4. Occlusion culling. Closeup view of an isosurface feature in the mixing interface of two gases for a simulation of a Richtmyer–Meshkov instability in a shock tube [21] rendered at 1 pixel tolerance on a 335×335 window. Without occlusion culling: 12 976 patches, 24 M splats, 1.7 fps; with occlusion culling: 3490 patches, 6.3 M splats, 5.5 fps.

and final pass, we traverse again the list of clouds and query the associated occlusion query object for the number of passed fragments. If this number is above a given threshold, we insert the cloud index in next frame's occluder list and, if the cloud was not among those rendered in the first pass, we proceed render it. With this method, the only additional cost of occlusion culling is the generation and test of occlusion queries. This cost can be further reduced by only checking once every few frames if previous frame occluders are still visible.

5. Conclusion

The multiresolution point-cloud structure has to be constructed off-line starting from a generic point cloud model. In [1], we presented a simple I/O efficient recursive clustering method that generates point clouds on-the fly using a Russian roulette approach at each partitioning step. With this top-down method, the coarsest levels of the structure are created by randomly picking few samples out of very large clouds with a considerable spatial extent. There is thus an inherent lack of control on sample distances, and the resulting irregular spacing may lead to poor results when the model is rendered at very coarse levels of detail.

In this paper, we present a bottom-up construction method, that retains the original simplicity of implementation, while producing higher quality results (see Fig. 5). The method is implemented with a single out-of-core component: a standard C++ array (compatible with `std::vector`), that encapsulates a resizable file accessed through system memory mapping functions. The procedure consists of two phases.

5.1. Top-down: partitioning

The partitioning procedure takes as input an external memory array of uniformly distributed point samples, together with its bounding box, and recursively generates a tree structure by binary space partitioning. At each bisection, a point cloud P_j is split if its point count N_j is larger than a threshold quantity $M_j^{(tot)}$. In case of splitting, the bounding box is bisected at the midpoint of its longest axis, the point cloud is subdivided among the two sub-boxes, and the partitioning procedure continues with the two sub-clouds. When recursion stops, the point cloud is stored in a point cloud repository maintained in an external memory array. The end result of this recursive scheme is a binary tree of nodes, that describes the subdivision structure, where each leaf of the tree corresponds to a cluster of size $N_j \leqslant M_j^{(tot)}$. The quantity $M_j^{(tot)}$, that drives the partitioning procedure, is computed at each node, following directly the definition of our layered point cloud structure. It represents the total number of points that the refinement procedure will generate in the region of node j. Since in our structure, differently from most other hierarchical schemes, each node j locally refines its parent by adding additional M samples to the representation, the total number of samples $M_j^{(tot)}$ extracted in the region associated to node j is not constant, but is rather recursively defined as $M_j^{(tot)} = \frac{N_j}{N_{parent(j)}-M} M_{parent(j)}^{(tot)} + M$, where N_k is the total number of samples in the subtree rooted at k, and for the root $M_0^{(tot)} = M$.

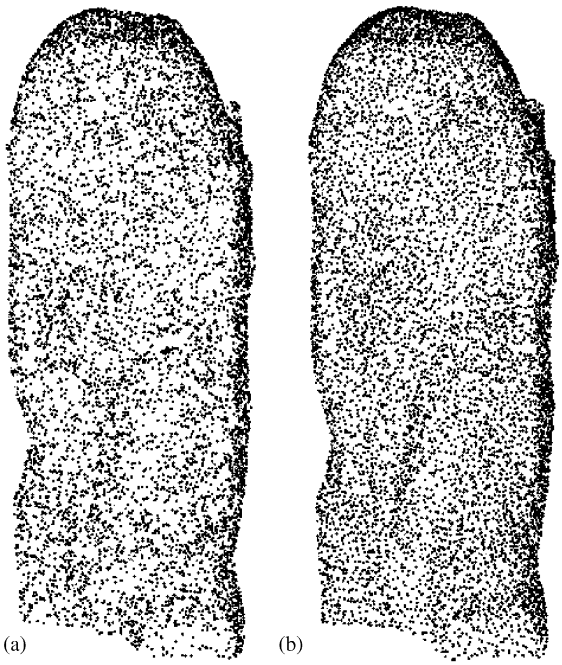

Fig. 5. Top-down vs. bottom up subsampling. The St. Matthew dataset (167 M samples) is subsampled using the same parameters with both techniques. The bottom-up approach visibly produces a more uniform spacing between samples with respect to the top-down approach presented in [1]. (a) Top-down construction result (16 K samples). (b) Bottom-up construction result (16 K samples).

5.2. Bottom-up: subsampling and structure construction

The second and final phase completes the structure with a hierarchy of surface representations by recursively associating to each node a fixed point count representation of the portion of the surface contained in it, along with all the information required for evaluating view dependent errors. This is efficiently done by traversing coarse-to-fine the structure generated by the first phase in the order in which nodes are stored on output (i.e., by reverse tree level and then by Morton code). At each node, we retrieve the associated point cloud from the repository, extract the required M

samples from it by uniform subsampling, and push the remaining ones up in the hierarchy, by storing them in the parent node's bucket. We then compute the index node data (hierarchical sample spacing, bounding sphere, and bounding cone of normals), and convert the point cloud to the final compressed representation. Subsampling, auxiliary value computation and compression are performed locally, by simply scanning the points extracted in the node's associated region and employing in-core methods. In our current implementation, subsampling is performed by hierarchical clustering [23], bounding spheres and cone of normals are computed by finding the minimum enclosing ball of points using a fast combinatorial method from computational geometry [24], while hierarchical sample spacing r_j is estimated using the standard k-nearest neighbor approximation [23]. These techniques are applicable because all points associated to a node's region, i.e., the node's points and the points inherited from its ancestors, are available at node construction time. By contrast, using our previous construction method [1], r_j had to be approximated from area ratios exploiting a uniform sampling assumption, leading to inaccuracies in areas containing irregularly spaced samples.

6. Results

The proposed method has been used to develop a C++ application which makes use of OpenGL on a Linux platform. Several tests have been performed on preprocessing and rendering of a number of very large models (see Fig. 6 and Table 1). Point sampled models were generated from triangulated ones by extracting vertex data. The largest model is a full resolution isosurface of the mixing interface from the Gordon Bell

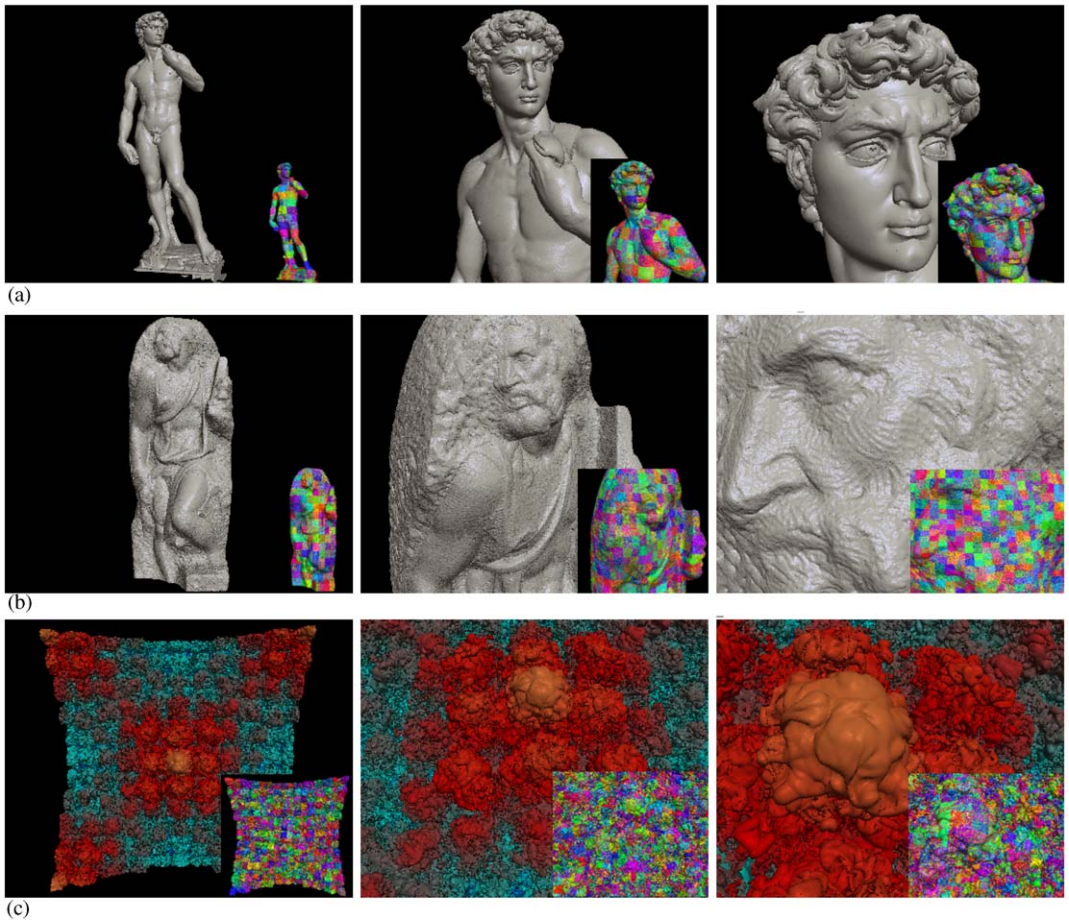

Fig. 6. Test models. The main images show the models as presented to the user during interactive inspection sessions, while the inset images illustrate the subdivision structure. (a) David 2mm (4M samples) and 1 mm (28M samples), (b) St. Matthew 0.25 mm (167M samples, and (c) Mixing interface isosurface (234M samples).

Prize winning simulation of a Richtmyer–Meshkov instability in a shock tube experiment [21], that consists of over 234 M sample points extracted from a $2048 \times 2048 \times 1920$ 8 bit grid. This model is convoluted and has a huge depth complexity (>100) from all viewpoints. The other test cases are high resolution scans of the St. Matthew and David statues from The Digital Michelangelo repository.

In addition to discussing raw experimental performance data for preprocessing, rendering, and streaming situations, we compare our results with those gathered from QSplat [2] and the adaptive tetrapuzzles [5] implementations on the same machines. QSplat is the reference system for rendering point-based models, while adaptive tetrapuzzles is a state-of-the-art technique for out-of-core construction and accurate view-dependent visualization of very large triangle meshes. The method uses a regular conformal hierarchy of tetrahedra to spatially partition the model. Each tetrahedral cell contains a precomputed simplified version of the original model, represented using cache coherent indexed strips for fast rendering. Appropriate boundary constraints are introduced in the simplification to ensure that all conforming selective subdivisions of the tetrahedron hierarchy lead to correctly matching surface patches. For each frame at runtime, the hierarchy is traversed coarse-to-fine to select diamonds of the appropriate resolution given the view parameters. Our layered point clouds and adaptive tetrapuzzles implementations share the same rendering and data access backends.

Table 1
Test models characteristics

Model	Samples	Triangles	Extent (mm)
David 2 mm	4,138,653	8,277,479	5200
David 1 mm	28,120,980	56,230,343	5200
St. Matthew	167,324,853	372,767,445	2700
PPM isosurface	234,717,830	469,381,488	2048

6.1. Preprocessing

Table 2 shows numerical results for the out-of-core preprocessing for the layered point clouds and adaptive tetrapuzzles techniques, relative to all test cases. The preprocessing has been evaluated on a PC running Linux 2.4, with two Athlon 2200+ CPUs, 1 GB DDR memory, a 70 GB ATA 133 hard disk. All the multiresolution models have been constructed with M set to 2 K samples/node for Layered Point Clouds and 2 K triangles/node for adaptive tetrapuzzles, using 16 bit/normal quantization and a position quantization ensuring a quantization error inferior to half of the input sampling distance (i.e., practically lossless quantization).

Overall processing times for layered point clouds range from 18 K samples/s to 30 K samples/s depending on the processor load, and is dominated by disk access times and LZO compression. Preprocessing speed is similar to that of QSplat (about 30 K samples/s). Compression rates exceed those of QSplat (around 50 bits/sample) and other similar systems based on a point hierarchy, but do not match those of state-of-the-art compression systems, since our current implementation has favored ease of coding through the exploitation of general purpose compression libraries. They could be improved by exploiting the locality of each patch, quantizing attributes relative to each cluster's contents.

On the other hand, the adaptive tetrapuzzles technique is much slower (about 2 K triangles/s on a single CPU) and produces files that are about 3 times larger. This is mainly because adaptive tetrapuzzles has to manage additional connectivity data and because the

Table 2
Numerical results for out-of-core construction

Model	Quantization		LPC				Tetrapuzzles			
	Position (bits)	Normal (bits)	Total time (s)	Disk usage (MB)		Output bit/sample	Total time (s)	Disk usage (MB)		Output bit/sample
				In	Out			In	Out	
David 2 mm	3×13	16	142	95	20	40	3735	379	70	142
David 1 mm	3×14	16	1233	644	126	37	24,499	2574	470	140
St. Matthew	3×15	16	8820	3830	777	39	92,255	17,063	3034	152
PPM isosurface	3×13	16	13,357	5372	925	33	198,199	22,530	3882	139

Tests performed on a single PC.

generated structure stores the full original model at the leaves and intermediate representations at inner nodes, while layered point clouds just redistributes input model points in a level of detail representation.

6.2. View-dependent refinement

We evaluated the performance of our view-dependent refinement technique on a number of inspection sequences over the test case models. The results were collected on a Linux PC with a Intel Xeon 2.4 GHz, 2 GB RAM, two Seagate ST373453LW 70 GB ULTRA SCSI 320 hard drives, AGP 8× and NVIDIA GeForce FX 5800 Ultra graphics. During the entire walkthrough, the resident set size of the application for the largest test case never exceeded 242 MB, i.e. less than 27% of the out-of-core data size, demonstrating the effectiveness of out-of-core data management. The qualitative performance of our view-dependent refinement is illustrated in an accompanying video that shows recorded live sequences.[2] As demonstrated in the videos on the 3D scanning models, that do not employ occlusion culling, we can sustain an average rendering rate of around 40M rendered points per second, with peaks exceeding 68 M. By comparison, on the same machine, the peak performance of QSplat, was measured at roughly 3.6 M rendered points per second when using the GL_POINTS rendering primitive. As in [10], this figure corresponds to the number of selected points divided by the frame-to-frame time. For the inspection of the 234 M samples isosurface, which has a huge depth complexity, we have enabled occlusion culling. On average, 50% of the patches are detected as occluded, strongly diminishing data access and rendering times. The average rendering rate drops in this case to around 30 M rendered points per second, which is still about an order of magnitude faster than that of QSplat. For the same view, and with the same screen space tolerance, we have measured that our method renders up to 10% more points than QSplat when occlusion culling is not enabled. This is because grouping points into clouds for all operations forces us to be more conservative in the projection. The increase in number of points is however compensated by a much larger increase in rendering speed. This is particularly useful for large scale display situations, where standard point-based solutions have problems to meet real-time constraints because of the large number of pixels to be covered. Fig. 7 shows the St. Matthew dataset examined on a large scale stereoscopic display assembled from off-the-shelf components, i.e., two 1024 × 768 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 1024 × 768 images per frame at 20 Hz with an adaptive rendering budget set to 1.5 M samples per eye. Under the same conditions, QSplat's visible frame rate drops down to about one frame per second, which is not sufficient for interactive applications.

Fig. 7. Large scale stereoscopic display. The St. Matthew dataset presented on a large scale stereoscopic display. A single PC is able to render two 1024 × 768 images per frame at 20 Hz with an adaptive rendering budget of 1.5 M samples per eye.

We have also compared the rendering performance and quality of QSplat and LPC with that of adaptive tetrapuzzles. The tests indicate that the adaptive tetrapuzzles is generally faster (70–90 M triangles/s). This is because mesh based solutions are able to fully exploit the post-transform-and-lighting cache with cache-coherent indexed triangle strips. This is not possible for point-based techniques, because each rendered point is independent of the others. Rendering quality of adaptive tetrapuzzles is also generally slightly better, especially for close-up views, than that of point-based solutions, because of the higher continuity. Point rendering quality could be improved by using oriented splats and blending, but this would reduce rendering speed. It should be noted, however, that, even though the quality of point based rendering does not match that of the triangle based one, it appears sufficient for most interactive display applications (see Fig. 8).

6.3. Network streaming

Some network tests have been performed on all test models, on a local area network at 100 M bps and on a ADSL at 1.2 M bps, using both NFS mounts and HTTP 1.1 connections. 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 Fig. 9, which shows the progressive refinement of the largest dataset on a machine connected through ADSL to a moderately loaded Linux box running a Apache web server. Even though the HTTP 1.1 is far from being optimal for the

[2]The video is available from: http://www.crs4.it/vic/multimedia/.

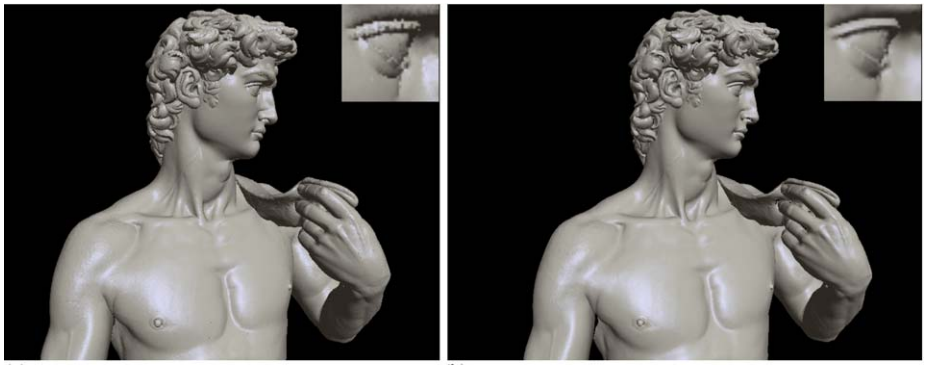

Fig. 8. Rendering quality. Renderer tolerances configured to generate approximately the same number of primitives. Inset images show a detail with a 4× magnification. Even though the quality of the point based version does not match that of the triangle based one, it appears sufficient for most interactive display applications. (a) Layered point clouds (1345 K & splats), and (b) adaptive tetrapuzzles (1421 K triangles).

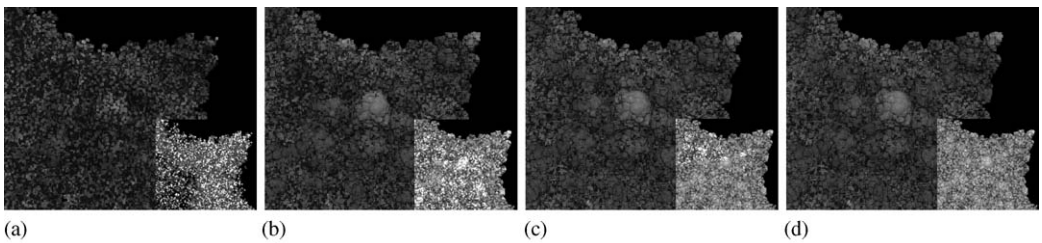

Fig. 9. Streaming. Progressive refinement of the mixing interface isosurface (234 M samples) on a ADSL connection at 1.25 M bps. The main images show the model as presented to the user, while the inset images illustrate progressive refinement by hilighting areas where refinement stops because of missing data. (a) 1s, (b) 10s, (c) 20s, and (d) 30s.

task, the application remains usable even for very large models on consumer-level network connections. The first images in the progressive refinement sequence also illustrate that a heavy subsampling on coarser scales can lead to strong aliasing artifacts for very complex models, as the average sampling distance is significantly below Nyquist frequency. Our static sample randomization replaces missing data with random information which is stable over time, thus the visual effect of aliasing is less noticeable as it would be for a regular sampling at similarly coarse resolutions. Nevertheless, for some models the occurring aliasing could notably diminish visual quality. Improving this aspect is an important avenue for future work.

7. Conclusions

We have presented a simple point-based multiresolution structure for interactive out-of-core visualization of very large point models on consumer graphics platforms. The system is comparable in both implementation complexity and image quality to (streaming) QSplat. Despite its simplicity, it is able to handle models of much higher depth complexity and is at least one order of magnitude faster in terms of rendering speed. The current major limitation is in image quality. Since we are focusing on high speed visualization, we simply use OpenGL hardware supported points for point cloud rendering, and do not use a per-sample splat orientation and size, which limits our ability to correctly treat texture and transparency. The integration with more advanced filtering techniques implemented on the GPU would resolve these problems, hopefully without compromising too much rendering speed.

As demonstrated in our tests, state-of-the-art triangle based solutions, such as adaptive tetrapuzzles, are currently slightly faster and produce higher quality images, mainly because they can fully exploit current graphics board designs, which are optimized for rendering meshes that share information at vertices. These solutions are, however, much harder to implement and require higher preprocessing times.

Given its simplicity, we consider the current method of immediate practical interest for most interactive display applications.

Acknowledgements

We are grateful to the Stanford Graphics Group and the Lawrence Livermore National Laboratories for making benchmark datasets available. Special thanks also go to Valerio Pascucci (LLNL) and Marco Agus (CRS4).

References

[1] Gobbetti E, Marton F. Layered point clouds. In: Proceedings of Eurographics Symposium on Point Based Graphics. 2004. p. 113–20, 227.

[2] Rusinkiewicz S, Levoy M. QSplat: a multiresolution point rendering system for large meshes. In: Computer Graphics Proceedings of Annual Conference Series (SIGGRAPH 00). ACM Press; 2000. p. 343–52.

[3] Cignoni P, Ganovelli F, Gobbetti E, Marton F, Ponchio F, Scopigno R. BDAM—batched dynamic adaptive meshes for high performance terrain visualization. Computer Graphics Forum 2003;22(3):505–14.

[4] Cignoni P, Ganovelli F, Gobbetti E, Marton F, Ponchio F, Scopigno R. Planet-sized batched dynamic adaptive meshes (P-BDAM). In: Proceedings IEEE Visualization. IEEE Computer Society Press Conference held in Seattle, WA, USA: 2003. p. 147–55.

[5] Cignoni P, Ganovelli F, Gobbetti E, Marton F, Ponchio F, Scopigno R. Adaptive tetrapuzzles—efficient out-of-core construction and visualization of gigantic polygonal models. ACM Transactions on Graphics 23(3), Proceedings of SIGGRAPH 2004, to appear.

[6] Levoy M, Whitted T. The use of points as a display primitive. Technical Report TR 85-022, University of North Carolina at Chapel Hill, 1985.

[7] Grossman JP. Point sample rendering. Master's thesis, Department of Electrical Engineering and Computer Science, MIT 1998.

[8] Rusinkiewicz S, Levoy M. Streaming QSplat: a viewer for networked visualization of large, dense models. In: Symposium for Interactive 3D Graphics Proceedings, 2001. p. 63–8.

[9] Kalaiah A, Varshney A. Statistical point geometry. In: Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing. Eurographics Association, 2003. p. 107–15.

[10] Sainz M, Pajarola R, Lario R. Points reloaded: point-based rendering revisited. In: Proceedings of Eurographics Symposium on Point Based Graphics. 2004. p. 121–8.

[11] Wand M, Fischer M, Peter I, auf der Heide FM, Straßer W. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. In: SIGGRAPH 2001 Proceedings. 2001. p. 361–70.

[12] Stamminger M, Drettakis G. Interactive sampling and rendering for complex and procedural geometry. In: Proceedings of the 12th Eurographics Workshop on Rendering Techniques. Berlin: Springer; 2001. p. 151–62.

[13] Dachsbacher C, Vogelgsang C, Stamminger M. Sequential point trees. ACM Transactions on Graphics 2003;22(3):657–62.

[14] Pfister H, Zwicker M, van Baar J, Gross M. Surfels: surface elements as rendering primitives. In: Akeley K, editor. SIGGRAPH 2000. Computer Graphics Proceedings, Annual Conference Series. ACM Press—Addison-Wesley, Longman: 2000. p. 335–42.

[15] Zwicker M, Pfister H, van Baar J, Gross M. Surface splatting. In: SIGGRAPH 2001 Proceedings. 2001. p. 371–78.

[16] Alexa M, Behr J, Cohen-Or D, Fleishman S, Levin D, Silva CT. Point set surfaces. In: Proceedings of the Conference on Visualization'01. IEEE Computer Society; 2001. p. 21–8.

[17] Ren L, Pfister H, Zwicker M. Object space EWA surface splatting: a hardware accelerated approach to high quality point rendering. Computer Graphics Forum 2002;21(3): 461–70.

[18] Botsch M, Kobbelt L. High-quality point-based rendering on modern GPUs. In: Proceedings Pacific Graphics. 2003. p. 335–343.

[19] Samet H. The design and analysis of spatial data structures. Reading, MA: Addison-Wesley; 1990.

[20] Gregorski B, Duchaineau M, Lindstrom P, Pascucci V, Joy KI. Interactive view-dependent rendering of large IsoSurfaces. In: Moorhead R, Gross M, Joy KI editors. Proceedings of the 13th IEEE Visualization 2002 Conference (VIS-02). Piscataway, NJ: IEEE Computer Society; 2002. p. 475–84.

[21] Mirin AA, Cohen RH, Curtis BC, Dannevik WP, Dimits AM, Duchaineau MA, Eliason DE, Schikore DR, Anderson SE, Porter DH, Woodward PR, Shieh LJ, White SW. Very high resolution simulation of compressible turbulence on the IBM-SP system. In: Supercomputing'99. ACM Press and IEEE Computer Society Press; 1999. CDROM Article No. 40.

[22] Yoon S-E, Salomon B, Manocha D. Interactive view-dependent rendering with conservative occlusion culling in complex environments. In: Proceedings IEEE Visualization. 2003. p. 163–70.

[23] Pauly M, Gross M, Kobbelt LP. Efficient simplification of point-sampled surfaces. In: Moorhead R, Gross M, Joy KI editors. Proceedings of the 13th IEEE Visualization 2002 Conference (VIS-02). Piscataway, NJ: IEEE Computer Society; 2002. p. 163–70.

[24] Fischer K, Gärtner B, Kutz M. Fast smallest-enclosing-ball computation in high dimensions. In: Proceedings 11th Annual European Symposium on Algorithms (ESA). 2003. p. 630–41.