The Visual Computer manuscript No.
(will be inserted by the editor)

David Chisnall · Min Chen · Charles Hansen

Ray-Driven Dynamic Working Set Rendering

For Complex Volume Scene Graphs Involving Large Point Clouds

Abstract Ray tracing a volume scene graph composed of multiple point-based volume objects (PBVO) can produce high quality images with effects such as shadows and constructive operations. A naive approach, however, would demand an overwhelming amount of memory to accommodate all point datasets and their associated control structures such as octrees. This paper describes an out-of-core system for rendering such a scene graph in a scalable manner. In order to address the difficulty in pre-determining the order of data caching, we introduce a technique based on a dynamic, in-core working set. We present a ray-driven algorithm for predicting the working set automatically. This allows both the data and the control structures required for ray tracing to be dynamically pre-fetched according to access patterns determined based on captured knowledge of ray-data intersection. We have conducted a series of experiments on the scalability of the technique using working sets and datasets of different sizes. With the aid of both qualitative and quantitative analysis, we demonstrate that this approach allows the rendering of multiple large PBVOs in a volume scene graph be performed on desktop computers.

Keywords out-of-core · very large dataset visualization · octree · point-based modeling · point-based rendering · ray tracing · volume scene graph

D. Chisnall and M. Chen
Department of Computer Science
University of Wales Swansea
Swansea SA2 8PP, UK
Tel.: +44-1792-295393
Fax: +44-1792-295708
E-mail: {csdavec, m.chen}@swansea.ac.uk

C. Hansen
School of Computing
University of Utah
Salt Lake City, Utah 84112, USA
Tel.: +1-801-581-3154
Fax: +1-801-581-5843
E-mail: hansen@cs.utah.edu

1 Introduction

Point-based modeling and rendering is a collection of techniques that enable direct processing of complex geometric objects represented by large discretely sampled point clouds [28,30]. Point clouds are usually rendered directly, using forward projection and image-space composition of point splats [41,10,25]. In terms of computational costs, this approach is highly attractive, facilitating the use of graphics hardware and stream-based data processing. However, by minimizing the interaction between points in the object space, it does not easily permit the generation of some visual effects, such as shadows, reflection and refraction, especially when considering a complex and arbitrary scene composed of multiple point-based objects.

An alternative direct rendering approach is to organize point-based objects in a volume screen graph and synthesize images using discrete ray tracing [4]. Not only does this approach address some of the shortcomings of splatting, but it also facilitates combinational and comparative visualization of volume and point datasets by allowing both to co-exist in the same volume scene graph. For example, in Fig. 1, a translucent bunny, built from a digitized point set,

Fig. 1 Combining the Stanford Bunny point set with a conventional volume dataset (San Diego rabbit heart) in a volume scene graph.

Fig. 2 A volume scene graph consisting of four dragons modeled using a point-set (Stanford Dragon) and artificial clouds represented by a volume dataset (Erlangen Clouds).

is combined with a heart captured as a volume data set. In Fig. 2, four dragons, also built from a digitized point set, is immersed in artificial clouds that are modeled as a volume dataset. Despite these benefits, the approach is yet to deliver a real-time solution. Nevertheless, it can no doubt bring benefits to some visualization and graphics applications; and with the rapid advances in both hardware technology and distributed computing, its usability will be further enhanced in the coming years.

The fundamental bottleneck of ray tracing a volume scene graph with multiple point clouds is that it demands an overwhelming amount of memory to accommodate all point datasets and their associate control structures such as octrees. Let $\{P_1, P_2, \ldots, P_n\}$ be a set of point clouds contained in a volume scene graph, and $\{C_1, C_2, \ldots, C_n\}$ be their corresponding control structures. To address the *scalability* of this approach, one needs to consider the following issues:

- *The size of each individual point cloud*, $|P_i|$ — With modern digitization technology, P_i may easily contain millions, or even billions of points.
- *The size of the control structure for each point cloud*, $|C_i|$ — A discrete ray tracer benefits particularly from a spatial partitioning scheme, such as an octree. However, to achieve optimal speed efficiency, it is not uncommon that the control structure for partitioning a point cloud may consume more space than the corresponding raw dataset, i.e., $|C_i| > |P_i|$.
- *The number of point clouds in a volume scene graph*, n — The growth of n has a profound impact on the total space requirement for both point datasets and their control structures.
- *The complexity of ray path predication* — For a simple ray-casting algorithm, it is possible to preprocess a volume scene graph and predetermine a static data prefetching strategy. The more visual effects the ray tracer incorporates, the more difficult the ray path predication will be. With some complex visual effects, it is likely that static preprocessing would not yield much benefit.

This paper describes an out-of-core system for ray tracing volume scene graphs in a scalable manner. We introduce a technique based on a dynamic, in-core working sets, which addresses the combined difficulties in pre-determining data mixing patterns and ray paths, and hence data access patterns. We assume that the number of points in each individual point cloud is much larger than the number of point datasets in a scene, i.e., $|P_i| >> n$. Hence we utilize octrees to partition individual point datasets, and use the bounding boxes of scene graph nodes to partition the scene. During ray tracing, the point datasets and their octrees are stored out-of-core, and the required octree nodes and point data are pre-fetched automatically according to access patterns predicted based on captured knowledge of ray-data intersection. Our testing results have shown that this technique is scalable, and enables volume scene graphs composed multiple point clouds to be rendered directly on desktop computers.

The remainder of this paper is organized as follows. We first give a brief review of point-based modeling and rendering techniques, and out-of-core techniques, in Section 2. In Section 3, we outline the basic in-core method for modeling and rendering volume scene graphs with multiple point clouds, and highlight the correlation between modeling quality and memory consumption. In Section 4, we present a ray-driven technique for predicting the working set automatically, and outline the dynamic algorithms for predicting octree access and for pre-caching octree nodes and the corresponding point data. In Section 5, we describe our experiments on the scalability of the technique using working sets and datasets of different sizes, and present our qualitative and quantitative analysis. This is followed by our concluding remarks in Section 6.

2 Related Work

There is a close relationship among volume visualization, implicit modeling and point-based techniques. *Points*, as modeling primitives, are extensively featured in all three classes of techniques.

The advances in *volume visualization* have produced a collection of methods for rendering *volume datasets*, including isosurfacing [24], ray casting [20] and forward projection [41]. Though in most cases, point primitives, i.e., voxels, are organized into a grid or a mesh, in some cases, volume modeling with scattered data is necessary, (e.g., [26]),

though existing approaches mostly involve the construction of a mesh structure connecting these points together.

Implicit modeling [2,27,42] facilitates the composition of complicated objects from elemental field functions, each of which is often defined on a point primitive. The octree method has been used for polygonizing implicit surfaces [3] and computing ray-surface intersections [16]. Due to the computational costs of polygonization and ray tracing, the emphasis has always been placed on the use of a small set of point primitives or elemental field functions.

One of the major advances in recent years is point-based modeling and rendering. The most significant examples of this development include Surfels [28] and QSplat [30]. Other important developments include [14,1,33,43]. In addition to the splatting approach commonly adopted in point-based rendering, ray tracing point clouds through intersection has been examined [31,40,39].

Recently a number of researchers addressed the convergence of these techniques, for example, approximating volume datasets with implicit models [15], building implicit surfaces upon point clouds, using the point-based approach for isosurfacing volume datasets [43,38,23], and combining point clouds and volume datasets in volume scene graphs [4].

Many visualization processes involve datasets that are much too large to fit into the internal memory of a computer, and have to rely on external disk storage, usually under the virtual memory management of an operating system. The external disk access can become a serious bottleneck in terms of rendering speed. *Out-of-core algorithms* (also known as *external memory algorithms*) [37] are designed to solve a variety of batch and interactive computational problems by minimizing disk I/O overhead.

Various out-of-core graphics and visualization algorithms have been proposed to handle large structured and unstructured 3D data-sets, for instance, in the context of (i) isosurface extraction [9,7,8,6,34], (ii) terrain rendering [22], (iii) streamline visualization [36], (iv) mesh simplification [21], (v) rendering time-varying volume data [32], (vi) rendering unstructured volumetric grids [19,12,6], (vii) ray tracing [29], and (viii) radiosity [35]. While some algorithms rely little on internal memory (e.g., [7,12]), others utilize preprocessed data structures, such as octree [36] and indexing [32] to optimize disk I/O operations. Use of *Active Data Repository* for visualizing large volume datasets was also reported [17].

While point datasets are usually excessively large, there has been little existing work on out-of-core methods for handling point datasets [13]. This motivates us to investigate the feasibility of multiple large point sets in the context of discrete ray tracing.

3 Modeling and Rendering Multiple Point Sets

This section describes the background problem in volume modeling and rendering for the out-of-core algorithm to be described in Section 4. While the latter is the main focus of this paper, it is useful to highlight the technical merits and issues associated with a generic approach for handling large point clouds in volume scene graphs. In this section, we also discuss the decision of using an octree over a kD-tree.

3.1 Point-based Volume Object (PBVO)

Volume objects can be defined procedurally as well as built from discretely sampled datasets such as CT and MRI scans. In particular, they can be defined on point clouds using appropriate *radial basis functions* [4], and can therefore be integrated into a volume scene graph as elemental volume objects at terminal nodes. For example, consider a discretely sampled point cloud $P = \{p_1, p_2, \ldots, p_m\}$, where each p_i is associated with a confidence value and an intensity value. We can map the confidence value to a radius r_i, and the intensity value to an opacity value, o_i. The former defines the *radius of influence* of p_i, and the latter contributes to the visibility of points within its radius of influence.

Consider a radial basis function, $\omega(q, p_i, r_i)$, such that,

$$\omega(q, p_i, r_i) = 0, \forall q, \parallel q - p_i \parallel > r_i$$

where $\parallel q - p_i \parallel$ denotes the Euclidean distance between q and p_i. For a collection of opacity values, $o_1, o_2, \ldots, o_m$, associated with $p_1, p_2, \ldots, p_m$ respectively, a scalar field O is therefore defined using a *blending function* as:

$$O(q) = \sum_{1 \le i \le m} \omega(q, p_i, r_i) v_i. \quad (1)$$

Several blending functions were considered in [4]. Images in this paper were rendered using either the function proposed by Wyvill *et al.* [42] or that proposed by Chen [4].

$O(q)$ in Eq. (1) in effect defines the opacity of every point in a 3D volumetric domain which is the union of the spherical bounding volumes of all points in P. Hence, $O(q)$ defines the essential component of a volume object and can be rendered using discrete ray tracing. Such a volume object is called a *point-based volume object* (PBVO). We can specify luminance properties of a PBVO using transfer functions, or by building the relevant scalar fields in a similar manner to $O(q)$. Fig. 3 shows a PBVO defined on a very large point cloud of over 14 million points. Its opacity field was constructed using a radial basis function with $r_i = 2, o_i = 1, i = 1, 2, \ldots, m$.

3.2 Volume Scene Graphs

In the theoretic framework of *Constructive Volume Geometry* (CVG) [5], a *volume scene graph* is an algebraic expression, called a *CVG term*, which involves a class of spatial objects and a family of constructive operations. In practice, a CVG term can be represented by a tree, where constructive operations are defined at non-terminal nodes, and elemental volume objects are defined at terminal nodes of the tree. Each subtree defines a composite volume object, while

Fig. 3 A point-based volume object defined on the the Stanford Lucy dataset of 14,027,872 points.

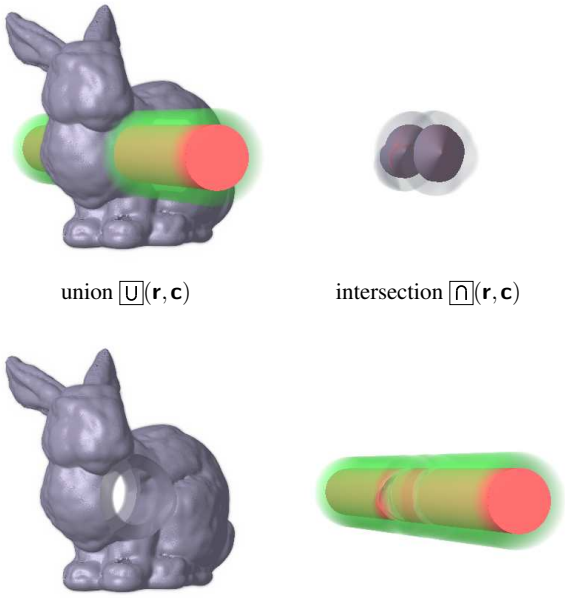


Fig. 4 Applying three basic CVG operations to a PBVO **r** and a procedurally defined cylindrical object **c**.

the root represents the final composite volume object, or the *scene*. To facilitate the sharing of low level object data, we allow a CVG term to be realized using a directed acyclic graph with a single root, hence resulting in a volume scene graph. Geometrical transformations and transfer functions can be applied at each graph node.

Fig. 4 shows the results of applying CVG operations to a PBVO, **r**, built from the Stanford bunny point set, and a procedurally defined cylindrical object, **c**. The example shown in Fig. 1 involves the use of a union operation and a difference operation. The latter is used, in conjunction with a cylindrical object, to remove part of bunny for exposing the heart. The example in Fig. 2 demonstrates that multiple PBVOs in a volume scene graph can share the same point set.

3.3 Discrete Ray Tracing

So far, discrete ray tracing is still the most appropriate means for directly rendering a volume scene graph which features multiple volume objects, solid or translucent. The basic ray tracing mechanism is to sample at regular intervals along each ray cast from the view position. At each sampling position s, we recursively determine if s is inside the bounding box of the current CVG subtree, until we reach a terminal node. If s is inside the bounding box of the terminal node which contains an elemental volume object, we evaluate its opacity field $O(s)$ and possible other luminance attributes.

When an opacity field is defined on a point cloud $P = \{p_1, p_2, \ldots, p_m\}$, it is necessary to identify a subset of points, $P' \subseteq P$, such that

$$P' = \{p'_i \,|\, p'_i \in P \text{ and } \| s - p'_i \| \leq r_i\}.$$

Given such a subset, we can sample the radial basis function of each $p'_i \in P'$, and obtain a scalar value by using the above-mentioned blending function.

3.4 The Benefits and Costs of Using Octrees

For a large point cloud, the most expensive cost in rendering a point cloud P is the identification of the subset P', as it involves a distance calculation against every point $p_i \in P$, thereby limiting the scalability when $|P|$ increases.

For each large point cloud P in the volume scene graph, we therefore utilize an *octree* structure for partitioning the points in the local data coordinate system of P. In each level of the hierarchy, a subtree contains only those points, which have some influence in the bounding box of the subtree. It is important to note that due to the non-zero radius of influence of each point, and the likely overlaps among the 'volumes of influence' of different points, a point element can belong to more than one leaf nodes. We therefore store only indices, rather than the records of points, in the leaf nodes of an octree.

In comparison with a brute force ray tracer, an octree-based ray tracer can have almost linear speedup in relation to the sizes of point clouds, if there is sufficient space for an octree that provides a sufficiently fine partition of a point cloud. Table 1 shows the speedup pattern in rendering three point sets different sizes, where points are placed randomly on a spherical surface.

However, Table 1 also shows that, for large point clouds, the amount of space consumed by an octree can be quite

height ▼	# points ►	1000	10000	100000
H = 3	speedup	23.75	36.62	26.92
	space	20 MB	155 MB	1499 MB
	η_{max}	45	411	3895
H = 5	speedup	49.91	318.84	446.81
	space	196 MB	741 MB	5708 MB
	η_{max}	15	89	739
H = 7	speedup	46.61	429.01	1122.27
	space	901 MB	12189 MB	63634 MB
	η_{max}	9	48	348

Table 1 Testing results for ray casting three randomly generated point clouds. The octree height is limited to 3, 5 and 7 respectively. η_{max} is the highest number of points that occupy a leaf node in the octree.

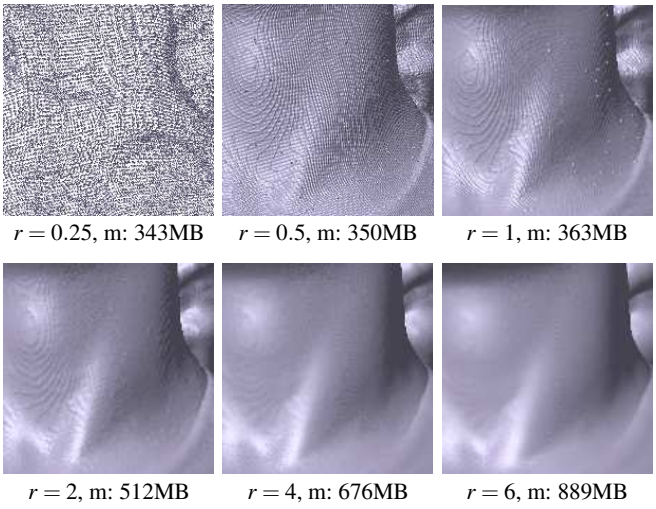


Fig. 5 Rendering of Lucy's neck with varying radii for the radial basis function, from 0.25 to 6. One can observe the improvement of image fidelity proportionally to the increase of the radius (r), which also leads to the additional consumption of memory (m).

noticeable, especially when points are densely placed, the radius of influence is set to a relatively large value, or the limit for octree height is generously set.

Note that the size of the octree can grow significantly faster than the point set. This is due to the fact that each point must be present in every octree leave node in which there exists a point where its radial basis function evaluates to a number greater than zero. Depending on the radial basis function, the point value, and threshold value used, a single point may have a radius of effect spanning a large number of octree leaf nodes. This differs from the traditional use of octrees for partitioning a volume dataset or a polygonal mesh, where each voxel or vertex resides only in one leaf node as the neighborhood connection is defined implicitly (in the case of a volume dataset) or explicitly (in the case of a polygonal mesh).

Fig. 5 illustrates the effect of varying the radius of the radial basis function. It shows a zoomed-in section from the neck of the statue shown in Fig. 3. As the radius is increased, the points blend together more smoothly to form the appearance of an iso-surface, and at the same time, the octree structure consumes more space for dealing with the increasing 'volume of influence' of each point. In fact, when we ran our in-core ray tracer on a desktop computer with 1GB memory, any setting with $r > 6$ encountered some difficulties, resulting in excessive virtual memory swapping and an unreasonable amount of system time overhead. Additionally, the octrees used in this set of examples are limited to only six levels, no where near the optimal depth for such a large dataset. Often some leaf nodes of the octree contained over 100,000 indices to points.

It has often been suggested that a kD-tree [18] could be deployed instead of an octree in this application as it has been successfully used in conjunction with many rendering algorithms such as ray tracing (e.g., [39,40]). We found that several features of the kD-tree make it unsuitable for this application. Firstly a kD-tree is most effective when points are considered to be unrelated or have the same small radius. Neither condition holds in our application, as we assume that points may have different radii (see Section 3.1 and [4]). Secondly, a kD-tree, like a BSP tree, focuses the precise order of primitives in relation to a given viewing (or ray) direction. It relies on search to identify neighboring primitives, for instance, in rendering point-based implicit surfaces [40]. Hence, the fast detection of an opaque surface closest to the viewing position minimizes the need for the search, the cost of which depends on the radius of points. In volume rendering, translucent objects are a common feature, which do not benefit from the precise ordering as much as opaque surfaces. On the other hand, minimal search requirement at each sampling point is better suited for rendering point-based volume objects.

Without considering the above constraints of a kD-tree, it is possible to design a kD-tree structure to accommodate overlapping radial basis functions in order to identify neighboring points without search. In this case it would encounter the same space issue as an octree. Hence, our study of out-of-core methods is also applicable to other spacial partitioning strategies. Nevertheless, consider a set of points and a sampling position in an octant and a kD-tree cell. Since the radial basis functions are close to a regular cube than an arbitrary cuboid, an octant represents a more cost-effective shape for storing points that are relevant to the sampling position.

4 Out-of-core Rendering

4.1 Algorithm Overview

The concept of *working set* was first introduced in the context of memory management in operating systems [11]. We adaptively use the concept in this work as our approach to the out-of-core management in some way resembles many typical methods found in operating systems, such as anticipatory paging.

Consider the running of an algorithm as a series of algorithmic steps $\{\Lambda_1, \Lambda_2, \ldots, \Lambda_i, \ldots\}$, and each step is merely a functional group of an arbitrary number of instructions. A *working set* of an algorithm in execution is the subset of the associated data structures being accessed during an algorithmic step Λ_i. Note that unlike the definition commonly used in the context of operating systems, here the duration of a working set is not a constant time window. In general, different algorithmic steps may require different execution time.

In the case of our discrete ray tracer, the primary algorithmic steps are sampling individual volume objects in a volume scene graph. For an octree defined on a point-based volume object (PBVO), the working set of a sampling operation is basically the leaf node of the octree that contains the sampling point, and the points referenced by the leaf node. Assuming that it is not possible to have all parts of the octree and the entire point set in-core at all times, the aim of our data management strategy is therefore to ensure that the working set for an algorithmic step Λ_i is located in-core, before and during Λ_i. Without such a data management strategy, the renderer will quickly encounter a situation that the working set for an incoming step Λ_j is out-of-core, stalling the renderer until it can be swapped in.

Fig. 6 gives an overview of the data environment of a volume scene graph to be rendered by our out-of-core ray tracer. For each individual point cloud, there is a complete out-of-core copy of the entire point dataset and the corresponding octree. Each out-of-core point cloud has an in-core memory cache. The amount of in-core memory available for each point cloud is set when an out-of-core copy is created, and can be modified by the user. This allows the algorithm to be easily scaled down in highly constrained memory situations.

Since we assume that the number of points in each individual point cloud is much larger than the number of point datasets in a scene, comparatively the actual memory requirement for storing a volume scene graph (without the actual data for its elemental objects) is negligible. We thereby maintain the data structure for the volume scene graph in-core. Note that it is possible for different PBVOs to share the same point clouds.

Our implementation consists of three main functional components, namely the *discrete ray tracer*, an *out-of-core octree controller*, and an *out-of-core point set controller*, connected as shown in Fig. 7. The ray tracer sends requests for discrete sampling points to the octree controller. This then determines whether the node containing the requested point is currently cached in-core. If it is, then it simply returns it, and attempts to predict the next one to be accessed. The prediction strategy will be detailed in 4.3.

Once the octree controller has determined the next node or subtree most likely to be accessed, it attempts to preemptively fetch it from disk. At the same time, it informs the point set controller of the point list from the accessed leaf node. On receipt of the point list, the point set controller

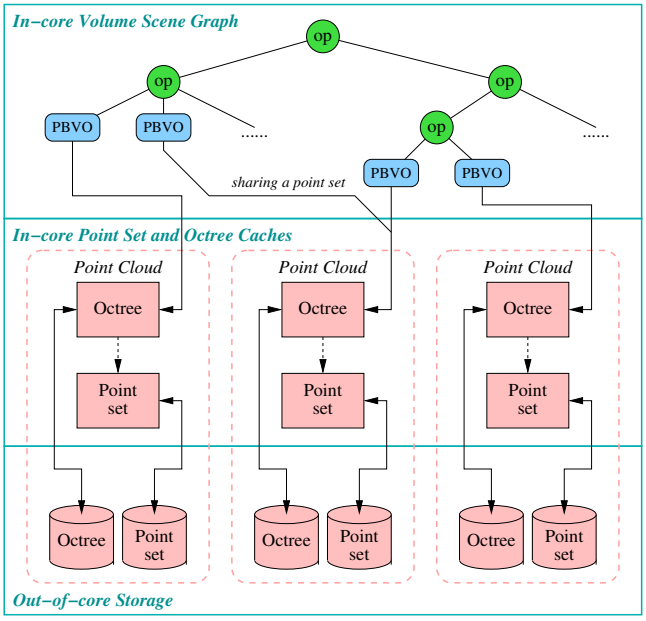


Fig. 6 The data environment of a volume scene graph.

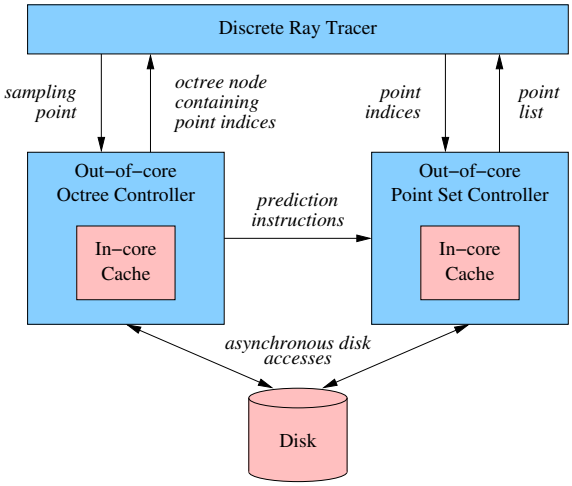


Fig. 7 The main components of the out-of-core renderer.

checks that these are cached in-core, and if not attempts to load them asynchronously.

The ray tracer then attempts to retrieve points identified by the octree node from the point set. By this time, they should already be cached in-core. Hence, it is the octree controller and point set controller that try to make sure the working set related to each sampling point is cached in-core before the ray tracer progresses to the sampling point.

4.2 I/O Management during Rendering

Most modern hard disks have a sufficiently high sustained transfer rate to keep the ray tracer fed with data, if it could

be read in a linear fashion. Consider a single point cloud. In our implementation, each octree node takes up to 244 bytes on disk[1] The maximum transfer rates for hard disks usually rely on reads of consecutive blocks of 512 or more bytes, making the required reads fairly inefficient.

The layout of the octree on disk is such that the child nodes of a single node are contiguous, allowing them all to be read with a single disk read. This has a negligible cost, since the cost of a slightly longer read is insignificant compared to cost of moving the disk head to the correct position to read one node. This layout does not benefit very much the navigation to a single leaf node, however it does increase the probability that a neighboring leaf node will be cached, giving a benefit when seeking the next leaf node.

In order to permit the fast traversal of the octree, the access of any non-leaf node causes all of that node's children to be pre-emptively swapped in. Additionally, the path between a currently accessed node and the root node is locked in-core, allowing movement up the tree to occur without requiring any disk accesses. The disadvantage of this approach is that it locks more nodes into in-core memory than are strictly required, however for discrete ray tracing applications, where adjacent octree nodes are frequently required, the benefit is worthwhile.

Individual nodes in the octree are accessed using a retain-release mechanism. Retained nodes are locked in-core. Once a node is released, and its reference count becomes zero, it is not immediately swapped out. Instead, its priority is decayed. No node is evicted until the allocated in-core space is exhausted. Since the point set and octree data are not modified during rendering, and hence does not need to be written back to disk, it costs little to free in-core nodes individually.

Unfortunately, it is not possible to group the points together on disk for linear reads. The reason for this is that each individual point can be referenced by several octree nodes, the number varying with the radius of the radial basis function associated with each point.

Various parts of the discrete ray tracer, such as opacity sampling and shadow sampling, trigger the pre-caching of out-of-core data. Each of these assigns a priority value between 0 and 255 to the record, representing the confidence of its prediction algorithm. When the allocated in-core space has been exhausted, records are swapped out based on priority, with the least recently used record of the lowest priority being accessed first.

The method provides two major benefits over the built-in paging strategy in a conventional operating system:

– *Fine grained access* — The operating system relies on the granularity of the paged memory system, and will only swap entire pages in and out of main memory. This means that, in order to load a single byte, an entire page (typically around 4KB) must be swapped out to make room for it. We are able to swap out only the relevant working sets actually needed.
– *Lower latency* — The operating system will typically wait until a page is accessed which is currently out-of-core before acting. It will then issue a page fault, evict the least recently used page, and then load the required page. The problems with this are that the evicted page may be the one needed next, and that the process accessing the page is stalled while it waits for the page to be loaded.

[1] Each node stores the x, y, z extents of the node for fast bounding volume tests, an address of the parent, and a type/status flag. In addition, each non-leaf node stores eight pointers to children but no references to points. Each leaf node stores references to a set of points, which have their volumes of influence overlap with the octant. As the size of this subset is variable, we adapted the UNIX mechanism for storing the block addresses in an i-node by having a fixed list of direct references, and an secondary reference when there are more points. The coordinates and attributes of points are not stored in the octree.

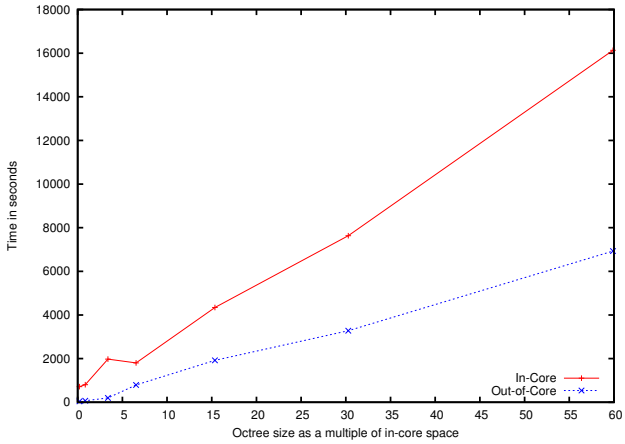


Fig. 8 A performance comparison between the in-core only rendering and the out-of-core approach with prefetching.

Figure 8 shows a comparison of the time taken to render a randomly generated point set using both in-core and out-of-core approaches. The octree size is expressed as a multiple of the amount of RAM space available to the rendering process. For example, from Table 1, we can observe that with 100,000 points, a 7-level octree tree would require 60 times more space than typical RAM space available on a PC.

In the pure in-core case, once this value of multiples exceeded one, the demand-paging subsystem in the operating system is automatically invoked to handle swapping. In fact, most everyday systems are configured with the size of the swapping space set to between 50% and 100% of the RAM space. This means, when the value of multiples reaches about $1.5 \sim 2$, the allocatable physical address space of the operating system will be exceeded and rendering will be aborted. Thus relying on the demand-paging subsystem is not scalable.

In order to observe the effectiveness of the prefetching strategy described in this paper in comparison with the demand-paging subsystem, we reconfigured an operating system by allowing significantly more swapping space than a typical configuration. It can be clearly seen that the out-of-core approach performs better in terms of raw speed. The in-core approach, using the operating system for paging takes more than twice as long to complete in all cases.

4.3 Prediction Scheme

The *prediction algorithm* developed for the ray tracer requires two out-of-core data structures, one representing the octree and the other representing an array of points. The algorithm detects octree access patterns based solely on its captured knowledge of previous accesses. The array of points has an internal predictor which predicts regular accesses, and additionally accepts hints from an external source. The first predictor is used when performing pre-processing — the point set is streamed into in-core memory and each point is processed in order. The second predictor is used during rendering, with the hints being generated from the octree.

Algorithm 1 Predict Octree Access

```
Require: s: the new sampling point
Require: s1 and s2: the last two sampling points accessed
Require: c: a Boolean value indicating if the next cell for s′ is cached
  N ← the leaf node containing s
  if s1 and s2 are set then
    if s, s1 and s2 are not on the same line then
      s′ ← the next sampling point on the ray (s, s2) but not in N
      M ← the node containing s′ {navigating from N to M}
      c ← conditional test if M is now in-core?
    end if
  else
    s1 ← s2
    s2 ← s
    c ← NO
  end if
```

Algorithm 1 shows the basic outline of the prediction algorithm. Accesses to nodes in the octree from a discrete ray tracing algorithm are along the paths of rays. The algorithm makes use of this fact, along with the fact that only two points are required to uniquely identify a line.

Every time a point is accessed, the algorithm first determines whether it is on the same line as a previous access. If this is the case, then it checks whether the last attempt to pre-cache the node was successful. If not, then it tries again. The attempted pre-caching is performed by Algorithm 2.

If the new point is not on the same line as previous ones, then it attempts to predict a new line based on the last accesses. This works for a secondary ray from a point, but fails for a new initial ray after the previous one terminates.

It is not difficult to extend this algorithm to record the start positions of rays and generate new predictions based on rays originating at these points. However doing so with current hardware results in performance degradation. The computation is relatively expensive, and the special case covered is infrequent. If processing speeds continue to increase at a rate faster than disk access times, then this may become a more attractive proposition within a few years.

Algorithm 2 handles the pre-caching. It attempts to navigate to the required node, without accessing any nodes stored out-of-core. First, it navigates up the tree until it finds a node which contains the point. This is guaranteed not to require accessing any out-of-core nodes, since the path between the root node and a currently retained leaf node is always locked in-core.

Algorithm 2 Pre-cache an Octree Node

```
Require: s: the new sample point
Require: N: the starting node
  while s is not in N do
    N ← the parent of N
  end while
  while N is not a leaf node do
    M ← the child of N containing s
    if M is in-core then
      N ← M
    else
      pre-cache M
      return NO
    end if
  end while
  return YES
```

The second phase navigates down the tree towards the node as far as it can without accessing a node that is not stored in-core. If it reaches an out-of-core node, then it stops, and pre-caches the node.

Fig. 9 illustrates this pre-caching mechanism. The node marked as A is the original node, containing s, and the node marked as B is the one containing the point $s\prime$. In other words, the last search within the octree returned A, and the next search along the same line which does not return A, will return B.

If no pre-caching were accomplished, this access would require three disk reads. To make matters worse, there is no way of determining where each node is on disk until its parent node has been read, and so none of these reads could be initiated until the previous one has been completed. Given common hard drive seek times of 8ms, this would stall the algorithm for 24ms — 24,000,000 clock cycles on a 1GHz CPU — when leaving an octree node in this way. Even adjacent octree nodes would incur an 8,000,000 cycle penalty.

The first time a point inside A is accessed, the prefetching algorithm will navigate along the branches indicated by the number ‘1’. Once it reaches the first node that is not cached in-core, it will issue a request that this node be pre-cached, and then return. The rendering algorithm can then

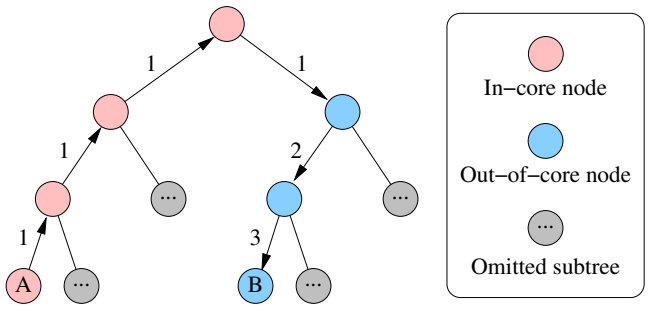


Fig. 9 Pre-caching nodes in an octree

perform processing on the points in A while the node is loaded. The second time a point in A is accessed, this node should be in-core, and so it will get two levels down the tree, to the end of the arrow indicated by a '2', before encountering an out-of-core node. Again, this node will be marked for fetching and the predictor will return. The third time, indicated by a '3', it will get to node B and cache this.

4.4 Prefetching Scheme

Prefetching data efficiently is very important to the overall performance of the rendering. Moving from a pure in-core implementation to an out-of-core approach caused a significant, yet unavoidable, performance hit. Previously, moving to the next node in an octree was as simple as de-referencing a pointer, something which can be accomplished in a very small amount of time. In order to perform the same operation in an out-of-core setting, the following steps are required:

1. Translate the record index to a disk location.
2. Calculate the hash of the location.
3. Determine whether the node is in-core.
4. Fetch it, if not.

The hashing technique is adapted from hardware cache implementations — the lowest n bits of the record index (not the disk address) are taken and used as the hash. This has the advantage that it is very computationally cheap to perform, and allows sequential accesses to fill the in-core hash table without collisions.

With a fast and efficient hashing algorithm, the time taken to determine whether a record is in-core is not high, however the access process still potentially takes several function calls — each of which has a cost associated with it — making it far slower than a simple pointer access.

If the prediction algorithm fails, then the cost is very high, since the data must be synchronously accessed from disk. This may stall the process for several hundred clock cycles while it waits for the disk access to return. In the case of the pure in-core implementation, this is the situation encountered whenever the required memory exceeds the available physical memory — each access to out-of-core virtual memory creates a page fault which stalls the process until a page is read from disk.

The prefetching process makes use of the operating system's asynchronous I/O capabilities. When a record is identified as requiring loading in-core, an asynchronous read request is dispatched, and the rendering process continues. When the record is accessed, or when the asynchronous I/O buffers have all been used, the request is completed. In most cases, the prefetch requests are dispatched sufficiently far in advance that the asynchronous read has completed by the time it is required.

In order to alleviate some of the system call overhead incurred from large number of small reads, the POSIX lio_list system call is used, which allows up to 16 asynchronous reads[2] to be initiated with a single system call. In many cases, this allows all of the points contained within a single octree leaf node to be loaded with a single system call. An additional benefit of this is that it allows the kernel to re-order the disk reads to reduce seek time on the disk.

Both the point set and octree make use of the same underlying code for shifting data in and out of core. This code receives the pre-caching requests and priority information from the high level code, and evicts low-priority records when their space is needed.

5 Experimental Results

5.1 Scene Graph Complexity

We have run a number of tests on different desktop computers, including a legacy Alhlon 1.4 PC (1.4GHz, 70MB), a Pentium 4 PC (3GHz, 1GB), a Pentium M 770 laptop (2.13GHz, 0.5GB), an PowerMac G5 (2×2.5GHz, 2GB) and an Apple G4 laptop (1.5GHz, 0.5GB).

Fig. 10 demonstrates that this technique can be used to synthesize images from complex volume scene graphs on a desktop computer. The volume scene graph is composed of six point-based volume objects, built from two point sets (Stanford Lucy of 14,027,872 points, and Stanford Bunny of 35,947 points). They are partially immersed in artificial clouds represented by a volume dataset (Erlangen clouds of 512×512×32 voxels). The scene is lit by three point light sources, casting shadows in different directions. From Fig. 10, we can observe hard shadows cast by the bunnies and the Lucy statue, soft shadows by the clouds, and self-shadows by Lucy's arm. As the radial basis function used for the Lucy point set has a much smaller radius, it requires much finer sampling intervals.

5.2 Memory Usage

The memory usage of the algorithm is entirely configurable, with the only constraint being that there must be enough in-core space allocated to store the maximum number of nodes which need to be processed at once. As discussed in 4.2, the path between a current access node and the root node of each octree is locked, this maximum number is thus related to height of the octree h and the number of individual point sets n. In terms of space complexity, this is of $O_{space}(n \cdot h) = O_{space}(n \cdot \log m)$, where m is the average number of leaf nodes in each octree, which is related the average size of each point set.

For example, in a single PBVO test with the Bunny point set, the total memory used by the out-of-core ray tracer — including the in-core data cache and all other memory allocated by the process — was under 20MB. In contrast, the in-core implementation used around 300MB. For larger data

[2] This number is implementation dependent, however 16 is common.

Fig. 10 A volume scene graph composed of six PBVOs (built from two point sets, Stanford Bunny and Lucy), a volume dataset (Artificial Clouds from Erlangen) and a procedurally defined floor.

sets, it is possible to keep the memory usage at a similar level, however this comes at the expense of speed. In the following section, the trade off between memory usage and performance is examined in more detail.

5.3 Performance

The performance of the algorithm is dependent on the amount of memory allocated to it. This can be controlled in two ways — the amount allocated to each point set and the amount allocated to each octree can be varied independently. Table 2 shows how the performance varies as each of these is changed.

The timing data in Table 2 gives the time, in seconds, taken to render a scene containing a single PBVO (Stanford Bunny of 35,947 points). It includes both preprocessing time and rendering. The preprocessing times vary between 55-65 seconds. The results were taken from a PowerMac G5 with 2×2.5GHz CPUs. Our current implementation currently only makes use of a single CPU directly, however the second CPU is used to process asynchronous I/O requests.

The results in Table 2 indicate that increasing the size of the point cache has a more noticeable impact on performance than increasing the size of the octree cache. This is due to the fact that the points required change dramatically over short octree traversals, meaning that a small increase in the point cache size can dramatically cut down the total amount of disk I/O required.

Table 2 Performance (in seconds) as memory is constrained.

number of octree nodes in core ▼	number of points in core 32768	16384	8192	4096
131072	94.59	98.67	100.31	107.15
65536	90.67	96.25	96.62	101.63
32768	88.28	95.20	94.89	100.91
16384	88.78	96.58	95.84	103.38
8192	92.39	98.56	100.31	108.25

Fig. 11 A volume scene graph composed twenty point-based volume objects (Stanford bunny), ray traced with three light sources.

Increasing the amount of memory allocated to the octree cache does not always give a performance benefit. The pre-fetching algorithm requires very little space to ensure that all of the required nodes are in-core before they are accessed. Once the cache reaches this size, increasing it delivers no benefit — the extra space is not required. Increasing this amount further increases the length of time required to find a single cache record, incurring a performance penalty. For larger cache sizes, this penalty is sufficient that increasing the allocated memory results in a performance penalty.

An in-core ray tracer would have a great difficulty in dealing with multiple point sets, such as the volume scene graph in Fig. 10. The out-of-core ray tracer can easily handle such a volume scene graph in terms of memory allocation, without experiencing inefficient disk I/O managed by the operating system. The rendering process in this case involves the full data environment as shown in Fig. 6.

To evaluate the performance under the condition of multiple point sets, we rendered the volume scene graph in Fig. 11 by associate the twenty bunnies to different numbers of point sets. Though we are in fact using the same point set, for the purpose of scalability test, we treat the repeated uses as independent point sets. For example, in the first test, one point set is used for all bunnies. In the second, half use one point set, and half use another. When we make ten repeated uses of the Bunny point set, we let each point set to be shared by two bunnies in the scene.

Table 3 Performance for multiple point sets.

number of point sets ▶	1	2	4	8	10
Preprocessing	57.0	114.8	230.9	470.7	589.0
Rendering	13.8	13.9	14.5	16.8	16.8

Table 3 gives both preprocessing (i.e., octree building) and rendering times in seconds. The preprocessing time scales linearly as new point sets are added, but the actual rendering time changes little and remains almost constant in relation to the increasing number of point sets from 1 to 10. The results are also shown graphically in Fig. 12. This clearly indicates the effectiveness and efficiency of the out-of-core management implemented in this work.

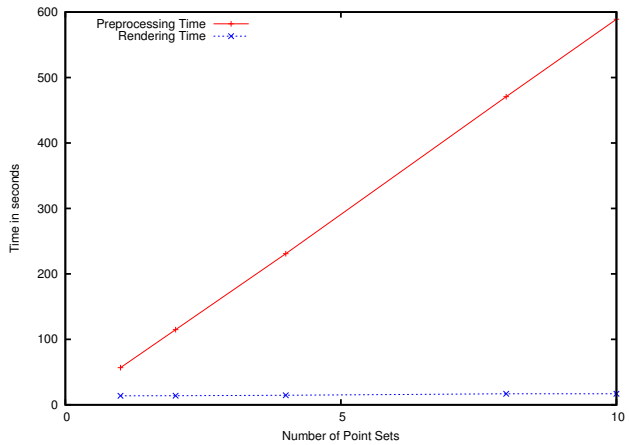


Fig. 12 Graph showing performance for multiple point sets.

6 Conclusions

We have presented an out-of-core solution to a difficult problem for rendering multiple point-based volume objects using discrete ray tracing. Our I/O management involves a dynamic, in-core working set, and we use a ray-driven algorithm for predicting the working set automatically. Our results have shown that the algorithm scales well to very low memory conditions. Performance increases can be gained by increasing the size of the point cache up to the size of the point data set, and by increasing the size of the octree cache up to a limit dependent on the structure of the data and the number of in-core nodes required at any given time. We have demonstrated that this approach allows the rendering of multiple large PBVOs in a volume scene graph on common commodity desktop computers.

Our future work will focus on the development of a parallel out-of-core ray tracer. We are in the process of extending this approach to a distributed-memory architecture. Our initial tests with limited algorithmic changes have shown a great potential to achieve an interactive rate for some less complicated scenes on a large cluster.

Acknowledgements This work was conducted under the framework of the eViz project, which is jointly funded by EPSRC ICT programme and e-Science Programme (GR/S46574). The collaboration was supported by an EPSRC travel grant (EP/D059674). The authors also wish to acknowledge the sources of datasets used in the paper, which include the *Bunny* and the *Lucy* from Stanford University, and the *Artificial Clouds* from the Universität Erlangen-Nürnberg.

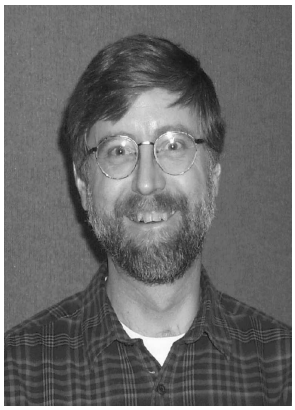
References

1. Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Silva, C.: Point set surfaces. In: Proc. IEEE Visualization, pp. 29–36 (2001)
2. Blinn, J.: A generalized algebraic surface drawing. ACM Transactions on Graphics **1**(3), 235–256 (1982)
3. Bloomenthal, J.: Polygonization of implicit surfaces. Computer Aided Geometry Design **5**(4), 341–355 (1988)
4. Chen, M.: Combining point clouds and volume objects in volume scene graphs. In: to appear in Proc. Volume Graphics. New York (2005)
5. Chen, M., Tucker, J.: Constructive volume geometry. Computer Graphics Forum **19**(4), 281–293 (2000)
6. Chiang, Y.J., Farias, R., Silva, C.T., Wei, B.: A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In: Proc. IEEE Symposium on Parallel and Visualization and Graphics (2003)
7. Chiang, Y.J., Silva, C.T.: I/o optimal isosurface extraction. In: Proc. IEEE Visualization '97, pp. 293–300 (1997)
8. Chiang, Y.J., Silva, C.T.: Interactive isosurface extraction. In: Proc. IEEE Visualization '98, pp. 167–174 (1998)
9. Cignoni, P., Montani, C., Puppo, E., Scopigno, R.: Optimal isosurface extraction from irregular volume data. In: Proc. IEEE Symposium on Volume Visualization, pp. 31–38 (1996)
10. Crawfis, R., Max, N.: Texture splats for 3d scalar and vector field visualization. In: Proc. IEEE Visualization, pp. 261–266 (1993)
11. Denning, P.J.: The working set model for program behavior. Communications of the ACM **11**(5), 323–333 (1968)
12. Farias, R., Silva, C.T.: Out-of-core rendering of large, unstructured grids. IEEE Computer Graphics and Applications **21**(4), 42–50 (2001)
13. Gross, M.: The utility of points as primitives for visualization. In: Keynote Speech in IEEE Visualization (2005)
14. Grossman, J.P., Dally, W.J.: Point sample rendering. In: Proc. Eurographics Workshop on Rendering, pp. 181–192 (1998)
15. Hua, J., Qin, H.: Haptics-based dynamic implicit solid modeling. IEEE Transactions on Visualization and Computer Graphics **10**(5), 574–586 (2004)
16. Kalra, D., Barr, A.: Guaranteed ray intersections with implicit surfaces. Computer Graphics (Proc. SIGGRAPH 89) **23**(3), 297–306 (1989)
17. Kure, T., Çatalyürek, Ü., Chang, C., Sussman, A., Saltz, J.: Visualization of large data sets with the active data repository. IEEE Computer Graphics and Applications **21**(4), 24–33 (2001)
18. L.Bentley, J.: Multidimensional binary trees used for associative searching. Communications of ACM **18**(9), 509–517 (1975)
19. Leutenegger, P., Ma, K.L.: Fast retrieval of disk-resident unstructured volume data for visualization. In: External Memory Algorithms and Visualization (DIMACS Book Series, American Mathematical Society), vol. 50 (1999)
20. Levoy, M.: Display of surfaces from volume data. IEEE Computer Graphics and Applications **8**(3), 29–37 (1988)
21. Lindstrom, P.: Out-of-core simplification of large polygonal models. In: Proc. ACM SIGGRAPH 2000, pp. 259–262 (2000)
22. Lindstrom, P., Pascucci, V.: Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. IEEE Transactions on Visualization and Computer Graphics **8**(3), 239–254 (2002)

23. Livnat, Y., Tricoche, X.: Interactive point-based isosurface extraction. In: Proc. IEEE Visualization, pp. 457–464 (2004)
24. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3D surface construction algorithm. Computer Graphics (Proc. SIGGRAPH 87) **21**(4), 163–169 (1987)
25. Mueller, K., Yagel, R.: Fast perspective volume rendering with splatting by using a ray-driven approach. In: Proc. Visuzlization '96, pp. 65–72 (1996)
26. Nielson, G.M.: Scattered data modeling. IEEE Computer Graphics and Applications **13**(1), 60–70 (1993)
27. Nishimura, H., Hirai, A., Kawai, T., Shirakawa, I., Omura, K.: Object modeling by distribution function and a method of image generation. Computer Graphics, Jounral of Papers given at the Electronics Communication Conference '85 (in Japanese) **J68-D**(4) (1985)
28. Pfister, H., Zwicker, M., van Baar, J., Gross, M.: Surfels: surface elements as rendering primitives. In: Computer Graphics (Proc. SIGGRAPH 2000), pp. 335–342 (2000)
29. Pharr, M., Kolb, C., Gershbein, R., Hanrahan, P.: Rendering complex scenes with memory-coherent ray tracing. In: Computer Graphics (Proc. SIGGRAPH 97), pp. 101–108 (1997)
30. Rusinkiewicz, S., Levoy, M.: QSplat: a multiresolution point rendering system for large meshes. In: Computer Graphics (Proc. SIGGRAPH 2000), pp. 343–252 (2000)
31. Schaufler, G., Jensen, H.W.: Ray tracing point sampled geometry. In: Proc. Eurographics Workshop on Rendering Techniques, pp. 319–328 (2000)
32. Shen, H.W., Chiang, L.J., Ma, K.L.: A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. In: Proc. IEEE Visualization '99, pp. 371–378 (1999)
33. Stamminger, M., Drettakis, G.: Interactive sampling and rendering for complex and procedural geometry. In: Proc. Eurographics Workshop on Rendering Techniques, pp. 151–162 (2001)
34. Sutton, P.M., Hansen, C.D.: Accelerated isosurface extraction in time-varying fields. IEEE Transactions on Visualization and Computer Graphics **6**(2), 98–107 (2000)
35. Teller, S., Fowler, C., Funkhouse, T., Hanrahan, P.: Partitioning and ordering large radiosity computations. In: Computer Graphics (Proc. SIGGRAPH 94), pp. 443–450 (1994)
36. Ueng, S.K., Sikorski, C., Ma, K.L.: Out-of-core streamline visualization on large unstructured meshes. IEEE Transactions on Visualization and Computer Graphics **3**(4), 370–380 (1997)
37. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Computer Survey **33**(2), 209–271 (2001)
38. von Rymon-Lipinski, B., Hanssen, N., Jansen, T., Ritter, L., Keeve, E.: Efficient point-based isosurface exploration using the span-triangle. In: Proc. IEEE Visualization, pp. 441–448 (2004)
39. Wald, I., Friedrich, H., Marmitt, G., Slusallek, P., Seidel, H.P.: Faster isosurface ray tracing using implicit kd-trees. IEEE Transactions on Visualization and Computer Graphics **11**(5), 562–572 (2005)
40. Wald, I., Seidel, H.P.: Interactive ray tracing of point based models. In: Proc. Symposium on Point-based Graphics (2005)
41. Westover, L.: Footprint evaluation for volume rendering. Computer Graphics (Proc. SIGGRAPH 90) **24**(4), 367–376 (1990)
42. Wyvill, G., McPheeters, C., Wyvill, B.: Data structures for soft objects. The Visual Computer **2**(4), 227–234 (1986)
43. Zwicker, M., Pfister, H., Baar, J., Gross, M.: EWA volume splatting. In: Proc. IEEE Visualization, pp. 29–36 (2001)

David Chisnall received his B.Sc. degree in computer science from University of Wale Swansea in 2003, and is currently pursuing his PhD study in the same university. In 2003, he worked on a short-term industrial project as a research assistant. His research interests include out-of-core visualization, knowledge capture and processing, system-level autonomic management algorithms, and distributed computation.

Min Chen received his B.Sc. degree in Computer Science from Fudan University in 1982, and his Ph.D. degree from University of Wales in 1991. He is currently a professor in Department of Computer Science, University of Wales Swansea. In 1990, he took up a lectureship in Swansea. He became a senior lecturer in 1998, and was awarded a personal chair (professorship) in 2001. His main research interests include visualization, computer graphics, interactive systems and multimedia communications. He is a fellow of British Computer Society, a member of Eurographics and ACM SIGGRAPH.



Charles (Chuck) Hansen received the BS degree in computer science from Memphis State University in 1981 and the PhD degree in computer science from the University of Utah in 1987. He is a professor of computer science in the School of Computing at the University of Utah. From Fall 2004 to Spring 2005, he was a Visiting Professor at ARTIS/GRAVIR IMAG/INRIA. From 1989 to 1997, he was a technical staff member in the Advanced Computing Laboratory (ACL) located at Los Alamos National Laboratory, where he formed and directed the visualization efforts in the ACL. He was a Bourse de Chateaubriand Postdoctoral Fellow at INRIA, Rocquencourt in 1987 to 1988. His research interests include large-scale scientific visualization and computer graphics.