Multi-View Visibility Orderings for Real-Time Point-Sampled Rendering

S. Parilov
LucasArts

W. Stuerzlinger
Department of Computer Science,
York University, Toronto

Abstract

Occlusion-compatible traversals and z-buffering are often regarded as the only choices for resolving visibility in the image- and point-based rendering community. These algorithms do either per-frame or per-pixel computations. This paper first discusses visibility orderings for point samples in general and then discusses orderings that are valid for multiple views. Then we present a novel, highly efficient, visibility algorithm for point-sampled models that has much smaller per-frame cost than previous approaches. We also discuss a high-performance, cache friendly implementation of this visibility method and present results. Finally, we speculate on possible hardware implementations.

1. Introduction

Real world environments are usually very complex. While computer graphics has made great progress, the generation of images of "real" scenes in real-time (>20Hz) is still a challenge. There are two fundamental problems in creating an image – identifying the pixels covered by any given object and determining which parts are visible to the viewer. These two problems are usually called rasterization and visibility.

Naive rasterization algorithms perform a number of operations proportional to the number of covered pixels. Recently, output-sensitive algorithms have been introduced, with rasterization time proportional to the number of pixels covered. For this progressively simpler versions are pre-computed. These simpler versions are used for more distant parts of the scene (as they appear in less detail on-screen). For visibility computations, there are two fundamental approaches. Visibility can be computed at linear cost in the number of primitives or on a per-pixel basis. The former assumes that visibility is computed on the primitives directly and is in the worst case proportional to the number of primitives. The latter takes time proportional to the number of pixels covered. An alternative that can compute visibility in sub-linear time is to order the primitives a priori in such a way that rendering them in that order yields the correct result. This ordering can be done at run-time with logarithmic cost if primitives are organized in a hierarchical data structure.

For highest efficiency there are two more requirements. It is important that primitives are read in a cache-coherent manner, otherwise memory latencies play a large factor, and ideally, a visibility algorithm should not require any frame-buffer reads. Finally, any rendering algorithm needs to compute a correct result. Due to the limits of sampling this is usually hard to guarantee, but the maximum error that can be tolerated is one pixel wide (usually around the boundary of an object). Consequently, a high-performance rendering algorithm should have the following properties:

- store a complete object representation
- rasterization cost proportional to the number pixels covered on the screen
- sublinear time for visibility computation (assuming some pre-processing)
- correct and consistent visibility (limited by sampling)
- cache-coherent data reads and writes
- only frame-buffer writes (no reads)
- the ability to efficiently utilize existing hardware
- the potential for hardware implementation

Our approach fulfills all of the above criteria. In the following, we first discuss relevant previous work. Then, we describe the theory behind our algorithm and consider its correctness under some assumptions. Later, we discuss practical considerations related to sampled images and pixel grids. We then describe the first implementation of the algorithm and present results.

1.1. Previous work

Over the last few decades, numerous point-sampled rendering approaches have been proposed. Among others, those include 3D image-based rendering by warping (IBRW) and rendering point clouds.

McMillan et al. derived warping equations that describe the relationship between the projections of a point on two images [10]. The pixels of the source image are then warped in an order towards or away from the epipole of an image. However, with this algorithm samples are written into the output in an unpredictable pattern, which complicates hardware implementations. Mark et al. modified the image traversals to be more cache-friendly [9], but concluded that occlusion-compatible traversals are not viable on modern cache and memory architectures due to bad cache utilization. Shade et al. resolved intra-scene occlusions by using Layered Depth Images (LDIs), which store multiple samples per source pixel [20]. Due to the nature of the data structure, implementations of LDIs suffer from bad cache-coherency and are hard to port to hardware [20]. Oliveira et al. decomposed the warping function into two steps, pre-warping and texture mapping, where the last step can be performed via graphics hardware. The simplicity of this method enables a very cache-coherent implementation. However, this method cannot represent images with more that one layer of samples. In summary, most of the image-based rendering works employ occlusion-compatible image traversals [10, 20, 4, 13, 16].

Pfister et al. represented objects as sets of surfels [15], i.e. point primitives without explicit connectivity, organized them into an octree, and rendered them with a warping algorithm. Holes in the generated images are filled with interpolation and an algorithm similar to z-buffering. Grossman et al. represented objects as collections of points in 3D [6]. They used a hierarchy of z-buffers and employed an algorithm that detects and fills holes in the images. Rusinkiewicz et al. present a system for rendering large scenes using a point rendering system (QSplat) that targets large point-sampled datasets [17, 7]. The object is stored as a hierarchy of bounding spheres, which allows for level-of-detail control and efficient frustum culling, and provides a compact dataset. Visibility is resolved using z-buffering. The randomized Z-buffer approach by Wand et al.'s approach randomly selects a set of samples on the primivites of the scene, with the number of samples proportional to projected area. The samples are then projected using z-buffering to resolve visibility [22]. Most point-based rendering systems use z-buffering to resolve visibility [8, 6, 15, 17, 22, 3, 14]. The main disadvantage of this apprach is the necessity to store and *update* depth information per pixel.

The discussed approaches are summarized in table 1, which considers the cost of rasterization, the cost of computing visibility in terms of the number of primitives. For cache-coherency, 4WR indicates that linear access patterns can be achieved by replicating the data and storing it in four different orders. Furthermore, the table lists is the algorithm suffers from visibility artifacts. The rest of the table provides data for a comparison of performance. McMillan's work was mostly theoretical, therefore performance data is not available.

	3D IBRW	LDI/LDI Trees	Relief Textures	3D HIC	Rand Z-buffer	QSplat	Surfels	Z-buf	Our approach
Complete scene representation	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Cost of rasterization	O(n)	O(n)/O(plogn)	O(n)	O(plogn)	O(plogn)	O(plogn)	O(plogn)	O(n)	O(plogn)
Per-frame visibility computation cost	Const	Const/O(logn)	Const	O(logn+p)	O(logn+p)	O(logn+p)	O(logn+p)	O(n)	O(logn+p)
Streaming source data reads	4WR	No	4WR	Yes	No	No	No	Yes	Yes
Cache-coherent frame buffer access	No	No	Yes	No	No	No	No	No	Yes
Guaranteed correct images	Almost	Almost	Almost	Almost	Almost	Yes	Yes	Yes	Can be guaranteed
Potential hardware implementation	Hard	Hard	Easy	Hard	Easy	Hard	Moderate	Done	Easy
Largest model size (samples)	N/A	$5.0*10^6$	$6.5*10^4$	$1.0*10^5$ tris	4.1*108 tris	$1.0*10^9$	$5.4*10^5$	N/A	$2.7*10^{11}$
Performance (Msamples/sec)	N/A	1.4	0.7	N/A	1.5	2.5	1.3	40,000	58
Hardware platform	N/A	300/250 MHz CPUs	PII 400 MHz	R3000	PIII 800 MHz	SGI Onyx2	PIII 700 MHz	GeForce4	12xR12000 300 MHz

Table 1: Comparison of image-based and point-sampled rendering approaches. *p* is the number of rendered samples, *n* is the number of samples in the scene.

1.2. Contributions

The fundamental idea of the new visibility algorithm presented here is the same as Newell, Newell, and Sancha's [5] and Seitz and Dyer's [19], only applied to point samples. The samples are first ordered in decreasing depth and are then rendered back-to-front. The samples drawn later then cover those drawn earlier [5].

In this work we present a new multi-view visibility sorting algorithm, which orders samples in a particular way. Once established, that ordering stays correct for nearby camera positions. Because samples are pre-ordered, they can be rendered by sequential traversal, which results in spatially and temporally coherent memory access patterns, and thus improves cache-hit ratios and makes simple and efficient hardware implementations possible. Whenever the camera moves too far, the algorithm efficiently re-sorts the samples in linear time to re-establish the ordering.

2. Visibility determination

We first discuss a theoretical, ideal, case, and prove the existence of correct multi-view orderings for point samples. We then analyze the difficulties that arise in practice, and propose a method to solve them.

2.1. Multi-view visibility orderings for point samples

For simplicity, we consider only the case of a planar perspective pinhole camera separated by a plane from the set of point samples. For the purpose of determining visibility, we define a point sample as an indication of the presence of a surface within a small neighbourhood of the point sample. In the ideal case, this neighbourhood is negligibly small, and the samples are thus infinitesimal. In this section, we also assume a continuous image, as opposed to a pixel grid. Note that even though the samples are infinitesimal and the image is continuous this situation can still give rise to occlusion problems (see Figure 1b).

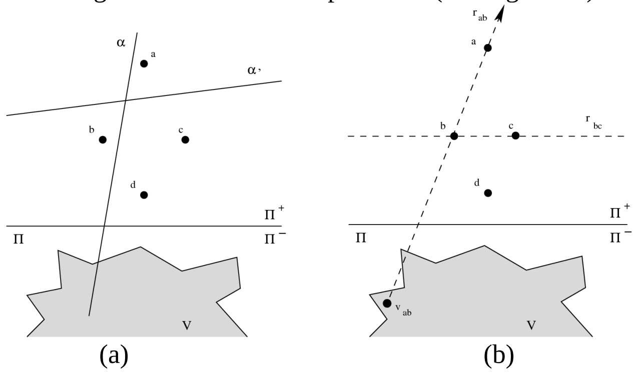

Figure 1: (a) Illustration of separating plane α
(b) Visibility ordering constraints. Lines passing through any pair of points may or may not impose restrictions on the visibility ordering. Ray r_{ab} intersects the viewing volume, forcing *a* to be rendered before *b*. Ray r_{bc} does not intersect the viewing volume, and hence places no restriction on the order in which *b* and *c* can be rendered.

It is then possible to render the samples in a particular order, such that the correct visibility is resolved by painter's algorithm [5]. Moreover, we claim that there exist orderings such that the same order is valid for a whole set of camera positions. Such orderings are called multi-view visibility orderings.

Proposition 1: Assume a set $S=\{s_i\}$ of point samples, a volume V, in which the camera used to render the scene is located, and a separating plane Π such that $V \subset \Pi^-$ and $S \subset \Pi^+$, where Π^- is the negative half-space, and Π^+ is the positive half-space, with respect to Π. It is possible to order S in a way such that the primitives correctly occlude each other when rendered as seen from any position in V (Figure 1a).

Proof of proposition 1: Consider a plane α, which separates two points, a and b. Similar to BSP-trees, the order in which the points should be drawn depends on the half-space in which the observer is located [1]. For points that are a finite distance from each other, it is always possible to select such a separating plane α', such that α' does not intersect the volume V. In this case, no matter where in V the camera is, the order of the points a and b w.r.t. visibility stays the same. QED.

One way to establish correct orderings, is to consider a ray $r_{ab} = v_{ab} + t_{ab}(a\text{-}b)$, passing through a and b such that $v_{ab} \in V$ and $t \neq 0$. For $t > 0$, the correct order is $<a,b>$. For $t < 0$, the order is $<b,a>$. Moreover, if no such ray exists (samples b and c in Figure 1b), the order in which the points are rendered is not important, since points can occlude each other only along a ray passing through both of them. Thus, we can say that a ray imposes an ordering constraint (see Figure 1b). Note that in the absence of ordering constraints many orderings are possible (see below). In particular, points sorted in order of non-increasing distance from the plane Π are always in a correct order.

The above idea does not apply for non-infinitesimal primitives, such as clouds or surfels, as it may then be impossible to pick a separating plane that does not intersect V. Figure 2 illustrates a case, where all potential separating planes intersect V. Then, it is impossible to construct an ordering that is correct for the all of V.

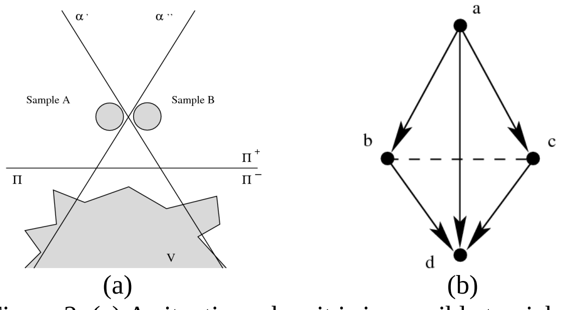

Figure 2: (a) A situation when it is impossible to pick a suitable separating plane for non-infinitesimal samples. (b) Occlusion graph. A directed edge $a \rightarrow b$ exists iff point b can occlude point a.

The above assumes that the camera does not go to infinity, that is V is bounded, and that samples do not coincide. If the camera is allowed to go to infinity on a plane parallel to Π, then there may not exist a separating plane between the points lying on a line parallel to Π. These assumptions are not too strict. When the camera crosses the separating plane, a new order needs to be established. This corresponds to the case of picking another image or LDI, which samples the scene better [20].

Another assumption in the above is infinitesimal samples. In practice, samples have non-infinitesimal size and the size of a neighbourhood determines the size of the footprint the rendered sample will cover in the image. However, if we assume that each sample produces only a footprint of one pixel or less, the above still holds (see section 2.3).

2.2. Visibility orderings in the ideal case

Consider a graph (called an occlusion graph), with a vertex for each sample in the original dataset. In the graph, a directed edge $a \rightarrow b$ exists, iff there is an ordering constraint between a and b with $t > 0$. An undirected edge a—b exists, if a and b never occlude each other when the camera is within V. The transitive closure of this graph reflects all visibility dependencies.

Any topological sort of such a graph then yields a correct occlusion compatible ordering of point samples. The fact that a topological sort is not unique in general implies that there are many possible orderings. Figure 2b presents the graph corresponding to the occlusion constraints for the points in Figure 1b. For this particular arrangement, correct orderings are $<a,b,c,d>$ or $<a,c,b,d>$.

As there may be multiple solutions to a given instance of the topological sort problem, there may be many correct orderings. In practice, any sorting algorithm will work, as long as the obtained solution is a correct solution of the topological sorting problem. For instance, it is sufficient to sort samples according to their distance from the observer. As there are many correct orderings, one can select the best one according to some criteria. For example, consider an image tiled into a set of rectangular tiles so that each one fits into one cacheline. One could assign weights to the edges of the occlusion graph proportional to the probability of two corresponding samples being projected into the same tile (for a camera within the viewing volume). Then a solution of the Traveling Salesperson Problem (TSP) on this graph would yield a correct ordering for which the cache-hit rate is close to the maximum. However, as we will see below, under certain assumptions it is possible to structure the data so that linear-time sorting methods can be used (see section 3.2).

2.3. Visibility ordering computation in practice

The above discussion is based on the assumptions of a continuous image, and infinitely small samples. However, real implementations need to quantize the image into pixels. Therefore, we assume the size of the neighbourhood of a sample to correspond to one pixel in the image or less. This can easily be guaranteed with level-of-detail algorithms, see section 3. With pixel grids, two samples occlude each other when they are projected into the same pixel. This constraint is harder to deal with than the ideal case, as the points can occlude each other even though there is no line passing through both points and the image plane of the camera.

The ideal case solution (section 2.2) did not place any constraints on the order of the samples on a plane parallel to the separating plane. Any ray through such samples is parallel to the separating plane and hence places no constraints on their visibility ordering. Thus, the ordering for such samples in the image plane is effectively random. As the ordering cannot identify the foremost sample the result may be arbitrarily bad in the sense that any sample may end up as the foremost in the pixel. The situation is illustrated in the Figure 3. Both camera *A* and camera *B* are located within Π^{-}, and the order in which points *a* and *b* are handled is pre-determined by the ordering. However, as can be seen from the figure, for camera *A* the correct ordering is <*b*,*a*>, and for camera *B* the correct ordering is <*a*,*b*>.

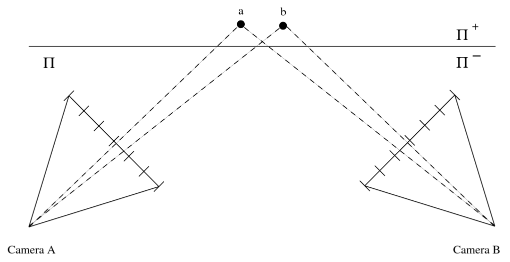

Figure 3: Occlusion errors due to image quantization.

A way to mitigate this undesired effect is to introduce additional separating planes (see Figure 4). The idea is then to pick the plane most perpendicular to the current viewing direction and order samples with respect to this plane each time the image is generated. While the samples are then re-sorted more often than theoretically needed, the chances of encountering an incorrect ordering are lower. This is because we have decreased the chances of two points lying on a ray parallel to the separating plane being projected into the same pixel of the image. This means that one can control visibility artifacts by setting a threshold on how much the viewing direction is allowed to deviate from the direction in which the samples are sorted.

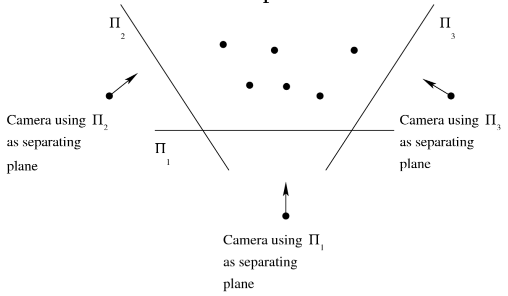
Π 2
Π 3
Camera using Π 2 as separating plane
Camera using Π 3 as separating plane
Π 1
Camera using Π 1 as separating plane

Figure 4: Using multiple separating planes. Depending on the camera position and viewing direction, either Π_1, Π_2, or Π_3 is a better choice for the current separating plane.

For faster rendering, we are interested in performing as few re-sorts as possible. The minimal number of re-sorts corresponds to the maximal value of the visibility re-sorting threshold. To determine the maximal value of the visibility re-sorting threshold that produces no artifacts, we performed a series of worst-case tests. For these tests we assumed that the density of the projected samples onto the image is at least as high as the image resolution. To facilitate computation of how closely samples can project we further assumed that samples are placed in an (arbitrarily dense) regular grid. If these constraints cannot be met, we assume that level-of-detail techniques are used (see section 3). The reader is also advised that sampling artifacts are outside the scope of this section (as opposed to visibility artifacts).

For these worst-case tests, we generated a 3D-grid with samples at each grid point. A full cube of samples is the worst case, as (1) the samples are as close to each other as possible thus maximizing the potential for visibility errors, and (2) removing a sample results in no visual artifacts in the case when the existence of this sample would produce an artifact. For a regular grid of samples, it is easy to show that once a correct sorting order is established, visibility artifacts can occur only if the viewing angle deviates more than 45° from the normal of a face.

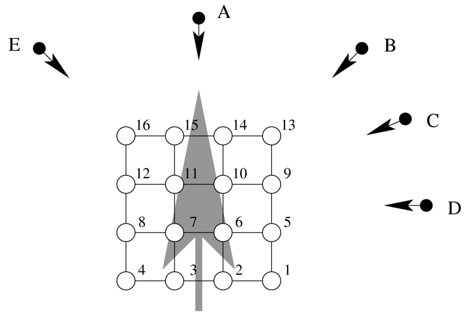

Figure 5: Different views of a set of pre-ordered samples in a grid (shown in 2D for illustration). The grey arrow illustrates the sorting direction; numbers next to point samples show the order of samples in the sequence. *B*, *C*, *D* are worst case view points. See text for details.

We fixed the ordering of samples to be according to the distance to one face of the cube (the sort-face). We then can view this cube from various viewing positions (Figure 5) and consider the results:

- Looking directly at the sort-face of the cube (camera *A* in the Figure 5) is the best case for the visibility sorting algorithm and no visibility artifacts can occur under the assumptions stated in the text until the camera reaches position *B* (or moves beyond *E*). Although some sampling artifacts are possible due to projection of samples from the front face of the cube onto a discrete grid of pixels in the image, these do not influence visibility.
- Looking directly at an edge of the cube between the sort-face and one of the adjacent to it faces (camera *B*). Here it is possible that visibility artifacts start to appear and the closer the camera gets to positions *C* and *D*, the higher the probability for artifacts.

In an implementation of this test we assigned each face of the cube a color, and the inside of the volume another to make artifacts easily noticeable. A correct image without visibility artifacts shows no pixels with the color of the inner volume of the cube. And indeed, as soon as the camera passes point *B*, the images start to show artifacts.

Summarizing we can say that as long as the viewing direction does not deviate more than 45° from the sorting direction for a grid of samples there will be no visibility artifacts. For real, non-worst-case scenes, the threshold value can be somewhat increased, which may reduce the number of re-sorts necessary while maintaining a low probability of producing artifacts. We experimentally determined that setting the threshold to 70° for non-synthetic datasets results in practically no visible artifacts. Another view of the problem is to observe that the ideal algorithm fails beyond camera *B* because of the finite size of the neighbourhood in which the surface sampled by the point sample is present. This was discussed at the beginning of this section.

2.4. Convex-cell space partitioning and LOD

The results of the previous section are only applicable for a viewer constrained to one side of a plane. To avoid this restriction on viewer motion, we resort to space partitioning methods (e.g. BSPs trees, octrees, etc.) Then, for an observer outside the current partition, the planes separating the cell from the rest of the volume can serve as separating planes. When the camera moves, it may cross such a separating plane, thus invalidating it. In this case, we have to pick another separating plane. With practically any spatial datastructure, we can use back-to-front traversals to resolve the visibility between the nodes. Inside a node, the visibility is resolved by sorting, as described above.
As we need to control the level of detail, we choose an octree to partition space. Before rendering an octree node, we check if the current ordering of the samples within the node is valid, given the current camera position. If the order is valid, we render the samples, simply traversing the stream of samples in one forward loop. If the order is not valid, the samples need to be re-sorted. When re-sorting, we choose the new separating plane to be the side of the octree node cube with the normal vector most orthogonal to the viewing direction. The pseudocode for rendering the scene is presented in Figure 6.

```
render_the_scene(a node)
  traverse data structure back-to-front;
  if current node has sufficient LOD
    render_the_node(node);
  else
    recurse to children;

render_the_node(node)
  get normal to current separating plane;
  compute view_vector to center of node;
  if angle between normal and view_vector
                             >= threshold
    pick separating plane most orthogonal
               to current view direction;
    topological_sort(node.samples);
  for all node.samples
    project and draw sample
```

Figure 6: Pseudocode for the rendering algorithm.

3. Implementation and Optimization

We implemented our system on a 16-processor SGI Onyx2 with 10GB of memory. Time-critical inner loops were fine-tuned manually.

3.1. Pre-processing

For generality, we assume that the source data is an unorganized point cloud, where its coordinates and color specify each point. We store these points in an axis-aligned octree. Within each node, we arrange the samples into a regular 16x16x16 grid, which allows nodes to fit into the cache. The regular sampling inside each node allows us to (1) store data in a compact form, and (2) efficiently process the data. The octree construction algorithm has a complexity of *O(nlogn)* on the number of samples, for non-degenerate cases.

3.2. Octree rendering and visibility sorting

To render the scene, we visit the nodes in back-to-front order. Due to level of detail control, the number of the nodes rendered is proportional to the resolution of the image being generated, but only logarithmically proportional to the complexity of the scene. Therefore, our algorithm is output-sensitive [21].
The visibility between the samples for a given octree node is established by sorting as discussed in section 2.3. However, based on the fact that samples are stored in a grid in each node, we can use a counting sort, which orders samples in *linear* (!) time. Splitting the scene into octree nodes also allows us to amortize the cost of visibility sorting over a number of frames, by re-sorting only parts of the entire scene at each frame, as opposed to re-sorting the entire scene every few frames.

3.3. Rendering a single node

An octree node is implemented as structure containing the number of samples stored in the node, the array of samples (*x,y,z* and *r,g,b* for each sample), and up to eight pointers to children of the node. The samples are located on a 16x16x16 regular grid and coordinates within each node are encoded in 4 bits. World-space coordinates are computed from the node size and its position in the world during the octree traversal.
Due to the visibility sort, the samples in a node are traversed in one forward loop when rendering. This achieves the highest possible performance on cached memory architectures. Note, that this cache-coherency is achieved because once a correct visibility order is established, that sequence can be cached and re-used over a number of frames. All renderings using this order are then a simple traversal of an array.

3.4. Image tiling, parallelization, and hole-filling

For better cache utilization on pixel writes, we tile our output images in a way similar to texture swizzling used in modern graphics hardware [12]. We also run 12 parallel rendering processes to achieve real-time frame rates even on large data sets. For simplicity, each CPU in our system has its own copy of the entire scene. The output image is partitioned into 12 regions (1 per CPU), which are rendered in parallel and integrated into the whole image before displaying. The image rendered from point samples may break up into disjoint pixels at close zoom-ins. To avoid this, when we reach a leaf node with no children, we temporarily create 8 children by replicating samples from the current node. For details, refer to [11].

4. Results

We tested our algorithm with several scenes, which are characterized in tables 2 and 3, and shown in figures 7 and 9. The octree construction during preprocessing took less than 30 minutes on a 2GHz PC for each of the mentioned datasets.

Scene	Octree depth	Total nodes	% non-empty	% leaf nodes
Cylinders	9	$1.7*10^6$	77.9	68.7
Checkerboard	10	$5.5*10^6$	100.0	75.0
Hand	7	$3.4*10^4$	79.7	62.6
Stanford Toys	10	$3.5*10^6$	75.9	51.4
Replicated Toys	16	$1.4*10^{10}$	95.3	77.8

Table 2: Octree statistics for test scenes (see Figure 7).

Scene	Samples in octree	Samples in leaf nodes	% samples in leaves
Cylinders	$1.7*10^6$	$9.9*10^6$	20.6
Checkerboard	$2.2*10^7$	$4.1*10^6$	18.8
Hand	$4.5*10^6$	$3.6*10^6$	64.1
Stanford Toys	$6.8*10^7$	$1.9*10^7$	27.6
Replicated Toys	$2.7*10^{11}$	$1.7*10^{11}$	62.8

Table 3: Sample counts for the test scenes.

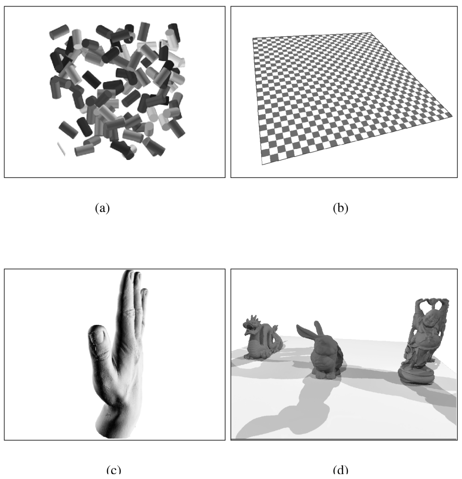

Figure 7: Test scenes. (a) Cylinders; (b) Checkerboard; (c) Hand; (d) Stanford Toys.

The Cylinders scene is targeted at creating a large number of visibility event. It consists of 100 randomly placed interpenetrating cylinders and samples were generated by raytracing three orthogonal views at 1500x1500 pixel resolution. The Checkerboard scene is a classic computer graphics example for testing algorithms involving re-sampling for severity of aliasing. The Hand scene is a laser-scan of a human hand model at 0.25 mm resolution. The Stanford Toys scene presents three complex objects, reconstructed from laser-scans of real objects at Stanford University. We ray traced the meshes for the Stanford Toys scene at high resolution from a number of viewpoints around it and then merged the results. The resulting point-sampled representation consists of approximately 69 million points. For large-scale tests we replicated this scene.

4.1. Performance of visibility algorithm

The visibility algorithm (1) traverses the octree, and (2) to re-sort the samples within a node, if needed.

The octree traversals in our system consume less that 2% of the time, and do not influence the performance in a significant way. Since the visibility orderings are valid for a number of views, the samples within nodes do not have to be re-sorted at each frame. We set the value of the re-sorting threshold to 70° in our experiments. This forces a re-sort only if the direction from the current viewpoint to the nodes is too different from the direction in which the samples within the node are sorted. The threshold was determined experimentally and can cause visibility artifacts that are at most 1 pixel wide. To guarantee no occlusion artifacts, the threshold value should be set to 45°. With these settings, we measured the average number of node re-sorts per frame versus the number of visualized nodes. The measurements were done on a 360-frame fly-around a scene, where the viewing direction was changing by 2° at each frame. Such settings approximately correspond to those expected during a high-frame-rate human-controlled navigation through a scene [2]. The results are summarized in Table 4.

	Avg. nodes rendered	Avg. nodes re-sorted	% re-sorted
Cylinders	3790	92.9	2.5
Hand	2728	108.0	4.0
Stanford Toys	22650	1469.6	6.5
Replicated Toys	70215	3510.0	5.0

Table 4: Visibility sorting statistics (per frame)

4.2. Performance of a single rendering process

We timed the performance of the render code using the Stanford Toys scene in a fly-around sequence. We measured the time required to re-sort a node and the time required to render a node with an octree containing a single node with 256 samples, and ran the re-sorting and rendering 10^6 times on it. The re-sorts took 38 seconds, the rendering 25 seconds. Thus, re-sorting a node is approximately 50% slower than rendering it. However, with the re-sorts occurring for only ~5% of the rendered nodes per frame this incurs only approximately 12.5% overhead. This is a modest loss of time, compared to other alternatives. For example, occlusion-compatible traversals, producing incoherent memory-accesss patterns on writes, can potentially spend more time fetching data from memory than actually processing it [9]!

We used the SGI performance analyzing tool perfex to collect more statistics. For $2.6*10^8$ samples, the implementation took 24.7 seconds, for a rate of $1.0*10^7$ samples/second or $9.6*10^{-9}$ seconds/sample, which corresponds to 29 CPU cycles/sample. The L1 data cache hit rate was over 98% and the cache line reuse was 58.31.

To test the performance on large scenes, we replicated the Stanford Toys scene 4096 times. This resulted in an octree of depth 16, with $2.7*10^{11}$ samples in $1.4*10^{10}$ nodes (see Figure 8). We rendered this scene using the same fly-trough sequence. The average performance was about 10 frames per second. This is a small decrease in the frame-rate compared to the significant increase in the number of samples in the octree. This test demonstrates the ability of our system to render large point-sampled datasets in real-time.

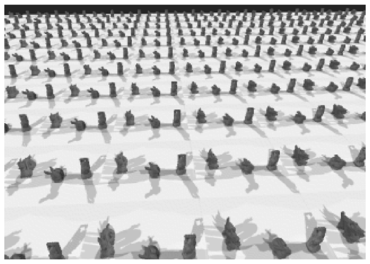
Figure 8: Part of the replicated Stanford Toys scene

5. Conclusions

We demonstrated a system to render large point-sampled scenes in real-time. It utilizes a combination of methods, level of detail control, a new multiple-view visibility algorithm, and efficient and fine-tuned parallel implementation. The hierarchical representation of the scene resulted in an output-sensitive algorithm. Our visibility algorithm is an alternative to the existing algorithms. The main advantages of our algorithm are its simplicity and high performance. The system achieves very cache-coherent data access patterns and avoids frame buffer reads completely, which opens the possibility for hardware implementations. We used parallelization to achieve real-time framerates in practice. Hence, this work is a step towards high-performance rendering of extremely detailed scenes.

Our approach can also be used to speed-up other forms of rendering. For example, an LDI can be partitioned into small regions, and the visibility order for each region re-determined whenever necessary. Some recent work, including [3] and [14], could also directly benefit from using our approach with minimal changes.

References

[1] M. Abrash. Bsp trees. Dr. Dobbs Sourcebook, 1995.

[2] R. Azuma. Tracking requirements for augmented reality. CACM, pages 50–51, July 1993.

[3] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry. Eurographics Workshop on Rendering, 2002.

[4] C.-F. Chang, G. Bishop, and A. Lastra. Ldi tree: a hierarchical representation for image-based rendering. In SIGGRAPH, 291–298, 1999.

[5] J. D. Foley and A. van Dam. Computer Graphics: Principles and Practice. Addison-Wesley, 1996.

[6] J. P. Grossman and W. J. Dally. Point sample rendering. Eurographics Workshop on Rendering, 181–192, 1998.

[7] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, et al. The digital michelangelo project: 3d scanning of large statues. SIGGRAPH, 131–144, 2000.

[8] M. Levoy and T. Whitted. The use of points as a display primitive. Technical report, University of North Carolina, Chapel Hill, 1985.

[9] W. R. Mark and G. Bishop. Memory access patterns of occlusion-compatible 3d image warping. SIGGRAPH/Eurographics Workshop on Graphics Hardware, 35–44, 1997.

[10] L. McMillan. An Image-based Approach to Three-Dimensional Computer Graphics. PhD thesis, University of North Carolina, Chapel Hill, 1997.

[11] S. Parilov, Master's thesis, 2002.

[12] nVidia. Geforce4 product overview. 2002.

[13] M. M. Oliveira, G. Bishop, and D. McAllister. Relief texture mapping. SIGGRAPH, 359–368, 2000.

[14] R. Pajarola. Efficient level-of-details for point based rendering. Proc. IASTED Computer Graphics and Imaging, 2003.

[15] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: surface elements as rendering primitives. SIGGRAPH, 335–342, 2000.

[16] V. Popescu and A. Lastra. High quality 3d image warping by separting visibility from reconstruction. Technical report, Univ. of North Carolina, Chapel Hill, 1995.

[17] S. Rusinkiewicz and M. Levoy. Qsplat: a multiresolution point rendering system for large meshes. SIGGRAPH, 343–352, 2000.

[18] G. Schaufler and W. Stuerzlinger. Three dimensional image cache for virtual reality. Computer Graphics Forum, 227–235, August 1996.

[19] S. M. Seitz and C. R. Dyer. Photorealistic scene reconstruction by voxel coloring. Computer Vision and Pattern Recognition Conference, 1067–1073, 1997.

[20] J. Shade, S. Gortler, L. He, and R. Szeliski. Layered depth images. SIGGRAPH, 231–242, 1998.

[21] O. Sudarsky and C. Gotsman. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality. Computer Graphics Forum, 249–258, 1996.

[22] M. Wand, M. Fischer, I. Peter, F. M. auf der Heide, and W. Strasser. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. SIGGRAPH, 361–370, 2001.