Enhanced Breadth-First Ray Tracing

Koji Nakamaru Yoshio Ohno

Abstract

Breadth-first ray tracing that utilizes uniform spatial subdivision can render a large number of objects without breakdown. The original algorithm however has two major drawbacks: redundant data processing and limited grid resolution. We present several refinements for these drawbacks and realize fast and robust external ray tracing. We achieved speedups of roughly up to 4x for SPD scenes with up to 50 million objects, and up to 14x for pathological cases with 1 billion objects, all rendered on a PC with 256MB memory.

1 Introduction

Ray tracing is known to be a powerful, versatile rendering technique. Unfortunately, the ordinal algorithms are designed for in-memory scene databases, and does not work well for large data. This is a challenging problem that was pointed out by Cook, et al. in their historic paper about REYES.

Recently, Pharr, et al. [Pharr97] and we [Nakam97] are concentrating on this topic on standard workstations. Pharr introduced a cache-based algorithm, while we introduced a stream-based algorithm – in other words, Pharr directly enhanced traditional, retained mode ray tracing with a sophisticated cache mechanism, while we constructed immediate mode ray tracing by switching the roles of rays and scene data. Pharr's algorithm stores both scene data and rays on disk, while minimizing the amount of reading.[1] This approach is similar to those of many algorithms developed for parallel computers. Our algorithm, on the other hand, keeps rays in memory and reads scene data sequentially. The access to scene data is minimized by processing rays in breadth-first order, thus maximizing the amount of rays processed at a time. Müller, et al. originally developed this *breadth-first ray tracing* for a large number of objects and their approach for accelerating ray-object intersection tests can be classified as a directional technique [Mülle92]. Our contribution was a more efficient algorithm that utilizes uniform spatial subdivision. Law, et al. also described a similar algorithm for thrashless ray casting on parallel computers [Law96].

[1] To be precise, tessellated geometry may also be added to/discarded from the cache only in memory. Consult the paper [Pharr97] for further details.

The main advantage of our original algorithm is robustness. It never causes a sudden breakdown that may occur for cache-based algorithms. This is the most important property of immediate mode rendering algorithms including Z-buffer and REYES. The disadvantage, on the other hand, is redundant data processing that cache-based algorithms intrinsically avoid. All data is processed even if it is not necessary at all, both in [Mülle92, Nakam97]. Another drawback of the algorithm, which is not directly related to cache-based algorithms, is that the grid resolution is limited by the size of memory.[2]

In this paper, we give our refinements for reducing these problems. The refinements are so simple and general that the original algorithm is enhanced smoothly, and furthermore, total speedup is significant in spite of their simplicity. We achieved speedups of roughly up to 4x for SPD scenes with up to 50 million objects (6GB), and up to 14x for pathological cases with 1 billion objects (87GB), all rendered on a PC with 256MB memory. Similar techniques are found in other non-ray tracing algorithms, especially enhanced REYES [Apoda00] and several culling algorithms [Mölle99]. We actually consider this paper as an example that shows how such techniques can be naturally imported into our original algorithm. Introducing these and other techniques, we realize fast and robust *external* ray tracing.

The rest of this paper is organized as follows. Section 2 describes the original algorithm and its drawbacks more specifically. Section 3 describes refinements for them. Section 4 shows experimental results and we conclude in Section 5.

2 Original Algorithm and Drawbacks

We describe our original algorithm briefly and its drawbacks more specifically here. Consult the paper [Nakam97] for a full understanding because the actual algorithm has more complicated details.

The basic idea of breadth-first ray tracing is *keeping rays instead of objects in memory and processing objects sequentially*. In our original algorithm, rays are stored in voxels of uniform spatial subdivision. For non-uniform environments, the user may specify multiple groups of objects, each of which corresponds to one grid. The whole process is outlined as follows:

Preprocess Voxels overlapping each bounding box are marked while reading data for each grid sequentially.

Intersection Intersection tests are done while reading data for each grid sequentially, and produce intersections for all rays in the current depth, including

[2] Although we explicitly show differences between two approaches for conciseness, many similar components are found in both algorithms and perhaps we can make a rendering system that seamlessly switches or combines them. The key for such hybrid systems is to develop methods for detecting or precursing thrashings in the cache-based part.

```
preprocess()
{
  for (each object on disk) {
    read an object;
    determine voxels overlapping its bounding box;
    mark overlapping voxels;
  }
}

intersection()
{
  while (there are unfinished rays in memory or on disk) {
    while (memory is not filled) {
      read a ray;
      traverse voxels for storing the ray
       into several candidate voxels;
      if (no candidate voxel for the ray is found)
        write the intersection to disk;
    }
    for (each object on disk) {
      read an object;
      determine voxels overlapping its bounding box;
      for (each overlapping voxel)
        for (each ray in the voxel)
          do intersection calculation
           for updating the intersection of the ray;
    }
    for (each ray in memory) {
      traverse voxels for storing the ray
       into several candidate voxels;
      if (the intersection of the ray is complete)
        write the intersection to disk;
    }
  }
}

shading()
{
  while (there are unprocessed intersections on disk) {
    read intersections to fill memory;
    sort intersections by object identifier;
    for (each object on disk) {
      read an object;
      do shading calculation and generate next rays;
    }
  }
}
```

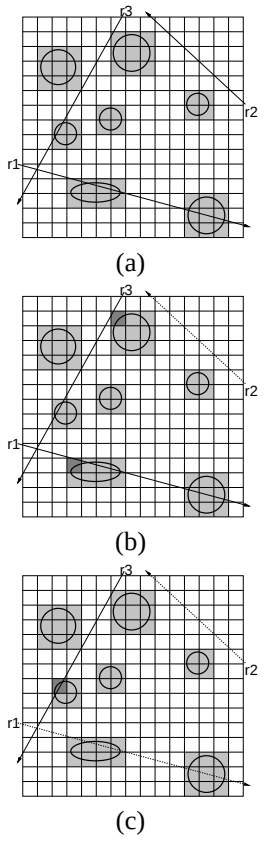


Figure 1: Pseudocode of the original algorithm and a graphical example of the intersection phase. In the graphical example, ‘several’ in the pseudocode is fixed to ‘one’. (a) Voxels overlapping each bounding box are marked in the preprocess phase. There are three rays: r1, r2, and r3. (b) In the intersection phase, rays are stored into first candidate voxels. The process for r2 is immediately finished because there is no candidate voxel on the path. The intersection point of r1 is found, but r1 has not reached that point (i.e., the intersection is not ‘complete’) and its process is continued. (c) The process of r1 is finished in the middle of traversing to the next candidate voxel. r3 is stored into the next candidate voxel and then its intersection is found.

shadow, reflection, and refraction rays. This procedure results in partial intersections. After all grids are processed, these intersections are merged to get final intersections.

Shading Local color contributions for the previous depth are determined using the results for shadow rays, and new rays are calculated with intersection points for non-shadow rays, again reading the scene data sequentially.

The latter two phases are repeated until either there is no new ray or the maximum tracing depth is reached.

The procedure for each grid in the intersection phase is most important and complicated. We firstly traverse voxels for storing each ray into voxels. For both limiting the amount of memory space needed and avoiding redundant intersection tests, we initially store each ray in only the first few, non-empty 'candidate' voxels it penetrates from its origin.[3] We then read each object and determine which voxels overlap its bounding box. If any voxel stores rays, we check the intersection between each of these rays and the object, and update each ray's intersection. After all objects are processed, we find the next set of non-empty voxels for each ray and process all objects again. We repeat this process as needed. Figure 1 shows a rough pseudocode of the whole algorithm and a graphical example of the intersection phase.

This algorithm keeps sequential access for data on disk and never breaks down. There are however two major drawbacks:

1. Redundant Data Processing:
   All objects are processed even if they are not necessary. In the intersection phase, we store rays only in the partial candidate voxels in each pass and therefore many voxels are empty. Reading an object overlapping such empty voxels is wasteful. Both determining which voxels overlap each bounding box and finding them empty are simple operations, but the total cost becomes high if there are a large number of objects. Reading all objects is also redundant in the shading phase where only objects intersected with rays are necessary.

2. Limited Grid Resolution:
   The grid resolution is limited by the size of memory. Although the algorithm works with some limited resolution, its computation time increases linearly for the number of objects. The computation time increases at a slower rate if the resolution is increased.

[3] The 'candidate' voxels are those into which rays should be stored. For example, a ray should be stored where it enters the region of voxels corresponding to a bounding box. Consult the paper [Nakam97] for further details.

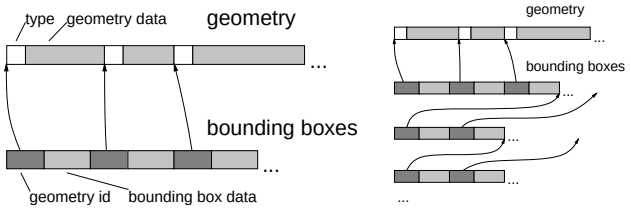


Figure 2: Data formats on disk. The sketches show only things about geometry and bounding boxes. (a) Input data is converted into two separated files: for geometry and for bounding boxes. The geometry file contains records each of which is organized with a type identifier and geometry data itself. The bounding boxes file contains records each of which is organized with an identifier of the corresponding data – actually, a byte position on file – and bounding box data itself. (b) Bounding boxes are then sequentially merged for creating a bounding box hierarchy. Each record contains the end position of children.

3 Refinements

There are two major problems described in the previous section: redundant data processing and limited grid resolution. In the following sections, we first solve the redundant data processing problem. The solution introduces several modifications, which are also utilized in solving the limited grid resolution problem.

3.1 Solution for Redundant Data Processing

Separation of Geometry and Bounding Boxes Most operations rarely require both geometry and its bounding box. The preprocess phase requires only bounding boxes, while the shading phase requires only geometry. The intersection phase requires both types of data, but geometry is required only if its bounding box is intersected with some ray. We separate bounding boxes from geometry considering these points. The data formats on disk are shown in Figure 2 (a).

Although seeking in the non-separated stream may seem to be enough, it requires another cost. Each seek operation requires an operating system call taking only a short time, but millions of calls cannot be neglected. In fact, we have experienced performance gain in building codes for large data storage by avoiding seek operations as many as possible.

Lazy Processing We introduce the *forward-only* seek operation and delay processing data as much as possible, instead of always reading and processing them. This is

a basic lazy evaluation, but was not emphasized in previous papers, though some of them might support it. It should be emphasized more because it has a great impact on performance if we process large data.

There are two types of lazy processing: for each object's geometry and with a bounding box hierarchy. The following sections describe them.

Lazy Processing for Each Object's Geometry As we described before, the intersection phase requires geometry only if the bounding box is intersected with some ray. The pseudocode of this part is thus modified as follows:

```
for (each object on disk) {
  read the bounding box;
  determine voxels overlapping the bounding box;
  for (each overlapping voxel)
    for (each ray in the voxel)
      if (the ray intersects the bounding box) {
        if (the geometry has not been read) {
          seek in the geometry data stream;
          read the geometry;
        }
        do intersection calculation
         for updating the intersection of the ray;
      }
}
```

The shading phase just follows a similar procedure in the inner '`if`' clause above. This makes the shading phase depend mostly on the number of rays, not on the number of objects.

```
sort intersections by object identifier;
for (each intersection) {
  if (the geometry has not been read)
    seek in the geometry data stream;
    read the geometry;
  }
  do shading calculation and generate next rays;
}
```

Lazy Processing with a Bounding Box Hierarchy Even if bounding boxes separated from geometry are processed, the cost of processing bounding boxes themselves grows in proportion to the number of objects. To reduce this cost, we introduce a bounding box hierarchy that is created by merging original bounding boxes sequentially. If a parent bounding box can be skipped in some operation, the operation for all children – including geometry – can be skipped. If it cannot be skipped, the check for the parent bounding box becomes wasteful. Parent bounding boxes are however few compared with the whole data, so that their overheads are also small. The new data formats on disk are shown in Figure 2b.

Actual merging for one hierarchy level is the following:

1. The 'area ratio threshold' T is set to $\left(s/N^{1/3}\right)^2$, where s is an user-specified constant, and N is the number of bounding boxes at the current level. If we

compute the area ratio for a cube whose side is s times as long as a side of a voxel, it is equal to T.

2. Each bounding box is read and merged with the current accumulated bounding box if $A/A_{all} \leq T$, where A is the surface area of the resulting bounding box, and A_{all} is that of the scene-wide bounding box.

Repeating the above procedure creates a bounding box hierarchy. We also limit the number of hierarchy levels and the number of top-level bounding boxes as follows:

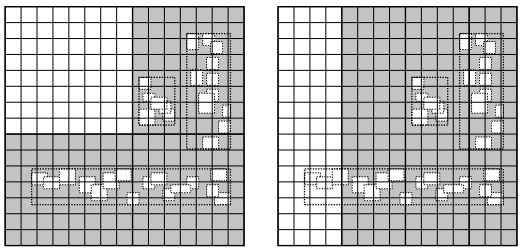
- If $N \leq n$ or $L \geq l$, we invoke the above procedure once using t instead of T and stop repeating the procedure, where n, l, and t are user-specified constants, and L is the next (resulting) level.

The above simple method groups small bounding boxes into clusters, while isolating large bounding boxes. It also automatically create top-level bounding boxes, which were manually specified in the original algorithm. Although the hierarchy is not optimal because we simply merge bounding boxes in a sequential order, it is common that objects in the stream (the output of modeling programs) locally distribute especially when there are a large number of objects. We keep the algorithm as an immediate mode one by adopting this method. Note also that this refinement itself does not forbid reordering objects, though such reordering for large data may require lengthy time and large temporary space. Some preprocess may reorganize objects to gain locality and to get a better bounding box hierarchy. For example, organizing objects as in the cache database described in [Pharr97] and/or utilizing *R*-tree are good choices. Major in-memory algorithms may also be utilized by sequentially splitting objects into several groups each of which fits in memory.

3.2 Solution for Limited Grid Resolution

The efficiency of uniform spatial subdivision depends on the grid resolution. The total amount of memory for efficient subdivision grows in proportion to the number of objects – this is a typical space-time tradeoff. If we handle one billion objects, each axis resolution becomes 1,000 resulting in 10^9 voxels. Non-uniform subdivision may reduce this problem, however, simple in-memory subdivision is not enough to solve the problem: how can we handle billions of objects that are uniformly distributed in space?

We favor uniform spatial subdivision because of its simplicity and efficiency, so that we first attempted to solve the problem for uniform spatial subdivision. Non-uniform environments are handled with multiple grids, as in the original algorithm. Other good solutions may include the use of aggregate objects, though we have not investigated it yet.



(a) (b)

Figure 3: Partition into subspaces in the preprocess phase. (a) Cubic partition (displayed with thick lines) maximizes the possibility of culling non-cubic parent bounding boxes (displayed with thick dotted lines). (b) Layered partition, for example, can never cull the parent bounding box at the bottom.

In the following sections, we show solutions for the preprocess phase and the intersection phase, where the intersection phase is further divided into two parts: traversal and ray-object intersection.

Preprocess In the preprocess phase, we process each subspace one by one instead of the whole space. For example, $1,000 \times 1,000 \times 1,000$ voxels can be handled as eight subspaces each of which contains $500 \times 500 \times 500$ voxels. Results are then merged to get the whole set of voxels.

Although this procedure needs multiple passes for the scene data, we can reduce the cost of each pass by utilizing a bounding box hierarchy. If the subspace is small, many bounding boxes do not overlap it; we can skip such bounding boxes at once if their parent bounding box can be skipped. Note that a parent bounding box may have an edge almost equals to that of the whole bounding box in length; we partition space so that each of the subspaces becomes cubic for maximizing the possibility of culling (see Figure 3).

Traversal The original algorithm reads a ray and immediately traverses voxels for the ray. This is easy to implement, but is not allowed for a high-resolution grid, because it causes random access to voxels stored on disk. To solve this problem, we first reduce the number of traversals and thus the amount of disk access by modifying '`intersection()`' in Figure 1 as follows, where a 'pass' means one cycle of the main loop:

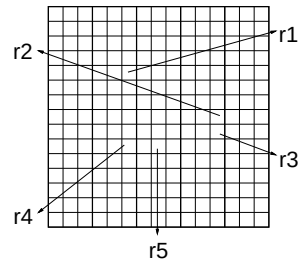


Figure 4: Traversal with layered partition. There are five rays (r1–r5). Each ray is initially stored into the block (displayed with thick lines) corresponding to the current voxel coordinate. These blocks are then processed one by one in both horizontal directions. r1 and r3 are processed in the pass from the left to the right, while r2 and r4 are processed in the pass from the right to the left. r5 may be processed in either pass.

```
intersection()
{
  // initialization
  while (rays are on disk) {
    read rays to fill memory;
    traverse voxels until each ray's 'first candidate voxel' is found;
    if (the first candidate voxel is found)
      its voxel coordinate is saved with the ray identifier on disk;
  }
  // main loop
  while (there are unfinished rays in memory or on disk) {
    read rays that have the first candidate voxels to fill memory;
    traverse voxels to get each ray's other candidate voxels in this pass
     until each ray's 'first candidate voxel' in the next pass is found;
    for (each object on disk) {
      ...
    }
    for (each ray in memory)
      if (the intersection of the ray is complete)
        write the intersection to disk;
  }
}
```

Note that the first candidate voxel in the next pass is *previously* determined. We can determine which ray survives in the next pass without traversing voxels again, so that the last '`for`' loop has no traversal. Although this modification may seem to be complicated, it is a simple application of *sentinel* (the first candidate voxel in the next pass is the sentinel).

Second, we adopt a method like scanline algorithms for preventing random access to voxels on disk; using layered partition, voxels are partitioned into several blocks each of which fits in memory. We then read voxels in each block and traverse voxels for each ray in the block. The actual procedure for '`traverse voxels...`' in the above pseudocode is the following (see Figure 4):

```
// initialization
store rays into blocks on x-axis;
// main loop
for (each block in ascending order) {
  read voxels for the current block;
  for (each ray in the current block) {
    traverse voxels;
    if (the ray exit the current block)
      store the ray into the next block;
  }
}
for (each block in descending order) {
  do the same procedure in the above 'for' loop;
}
```

The above method needs to access the whole set of voxels only twice at most and the access can be done in a sequential order. We may partition other axes for further reducing memory requirement, though it will require more access to disk.

Ray-Object Intersection After the traversal part is finished, how should we hold voxels for storing rays? An easy solution is a similar one in the previous sections: holding only partial voxels and processing all objects for these voxels. Repeating this process produces correct results, however, it increases passes for objects. Preventing the increase of passes for objects is important, so that we alternatively adopt a two-level grid. For example, $1,000 \times 1,000 \times 1,000$ voxels are replaced with $100 \times 100 \times 100$ upper level voxels each of which may store a lower level block containing $10 \times 10 \times 10$ voxels. This is similar to the method described by Jevans, et al. [Jevan89], except we store rays instead of objects.

Holding voxels in two levels, as such, is not enough for reducing memory requirement. Suppose that there are some rays in every upper level voxel. Simply allocating lower level blocks in this worst case requires a large amount of memory again. We adopt the following method for solving this problem, because in general few upper level voxels have many rays (see Figure 5):

1. The fixed number of lower level blocks are allocated.
2. Rays that will be stored are counted in each upper level voxel.
3. Lower level blocks are assigned to upper level voxels, in the order of counts: a voxel with more rays has priority. There may not be enough blocks, because the number of blocks is limited.
4. Rays are stored. Each ray is stored into voxels in lower level blocks if possible; otherwise, it is stored into upper level voxels directly, with its voxel coordinates.

For rays directly stored into upper level voxels, we must check whether the ray actually belongs to voxels that overlap each bounding box. Storing rays into upper level voxels thus causes an overhead in general, but it is also better for a small number of

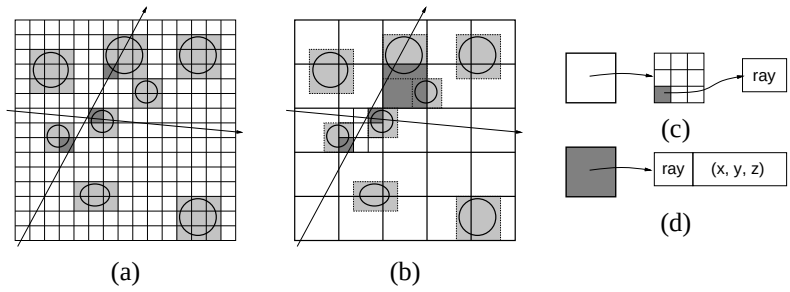


Figure 5: How to store rays into voxels in two levels. (a) Traversal results. There are three voxels (small dark ones) in which rays should be stored. (b) Voxels in two levels. Thick lines show upper level voxels. The lower level voxels (small dark ones) and the upper level voxel (large dark one) store rays. (c) If a lower level block is available, lower level voxels in the block hold the ray. (d) If a lower level block is not available, the upper level voxel holds the ray with the voxel coordinate. The voxel coordinate is used to determine whether the ray belongs to voxels that overlap each bounding box.

rays because accessing lower level voxels requires another cost. The above method reduces memory requirements, limits the amount of memory required in the worst case, and its speed loss is small.

Although it is difficult to find optimal resolutions for upper/lower level voxels, we currently adopt the following method and maximize the upper level resolution; this reduces rays per upper level voxel:

1. The user initializes upper and lower resolutions for an axis – R_u and R_l – with their maximum values.

2. The grid resolution is determined by Klimaszewski's method [Klima97]. Its maximum resolution for an axis is defined as R.

3. If $R \leq R_u$, we employ a single-level grid.

4. If $R > R_u$, we employ a two-level grid where R_u and R_l are modified as follows: $R_l = min(R_l, ceil(R/R_u))$, and then $R_u = min(R_u, ceil(R/R_l))$.

4 Results

We have run the implementation under Linux on a 450MHz PentiumII PC with 256MB memory. All test data was produced in NFF with the SPD package [Haine87]

and was converted into binary form. No instancing was used, and other required testing procedures were also followed. In the rest of this section, we show several results in brief. Consult the JGT web page for more detailed descriptions of experimental configurations and results.

We firstly tested relatively small data (up to 50 million objects/6GB) for investigating various tendencies of refinements in Section 3.1 – *separation of geometry and bounding boxes* (S), *lazy processing for each object's geometry* (G), and *lazy processing with a bounding box hierarchy* (B). Each grid resolution was determined by the method described in Section 3.2. For all scenes, refinements become more effective as data grows. For each scene of its largest size factor, the 'S+G' results in speedups as was expected, and its gain factor is 1.38–2.30. On the other hand, the effectiveness of 'B' greatly depends on the depth complexity (how many objects a ray passes through) and/or the occlusion complexity (how many objects are occluded) as in other culling techniques. The 'B' also has an overhead for creating and processing a bounding box hierarchy. The 'S+G+B' therefore results in various speedups; the gain factor is 1.39–4.48, and the 'S+G+B to S+G' gain factor is 0.99–2.63. The 'B' however does not cause large overheads in worst cases, because there are only a small number of parent bounding boxes. We also tested two-level grids for these scenes by limiting the grid resolution explicitly. Although two-level grids have 10–20% loss, it can be rewarded with the efficiency of high-resolution grids.

We secondly tested refinements for huge data. Table 1 summarizes statistics. These statistics show that applying refinements for huge data further improves rendering (preprocess and tracing) time. The gain factors range from 5 to 14 in rendering time. The gain factor in tracing time is quite large – 70.78 – for 'rings368m' ('rings368' with different view data), because only a small portion of objects contributed to the image and many objects were culled by 'B'. Note that it is not an ordinary view-frustum culling; there are reflection/shadow rays going outside the frustum. Note also that high-resolution grids were used by applying refinements in Section 3.2 even for 'base' in order to avoid lengthy computation time. The gain factors would be much larger if the grid resolution for 'base' was limited.

For huge data, creation time is extremely long, and the preprocess phase accounts for a large part of rendering time. This results from strictly following the testing procedure of SPD. In actual systems, however, it is important to cooperate with modeling systems for efficient rendering.

5 Conclusions

We have shown several refinements for breadth-first ray tracing utilizing uniform spatial subdivision. Total speedup is significant. The new algorithm is much faster than the original one if there are many objects having no contribution to the rendered image, while it still works nicely in other situations. It also allows the use of high-

Table 1: Statistics for huge data

statistics of scene data and rendering (preprocess+tracing) time

	#objects	size (MB)	creation	base	S+G	S+G+B	
mount14	536,870,916	49,664	19:16:15	19:26:15	2.04	5.33	(2.62)
rings368	1,000,787,041	88,857	26:37:42	50:54:40	1.84	12.43	(6.76)
rings368m	1,000,787,041	88,857	26:37:42	30:40:26	1.65	13.65	(8.26)

preprocess/tracing time

	preprocess				tracing			
	base	S+G	S+G+B		base	S+G	S+G+B	
mount14	6:31:37	2.10	8.87	(4.23)	12:54:37	2.01	4.44	(2.21)
rings368	11:03:30	1.82	5.61	(3.09)	39:51:10	1.84	18.76	(10.17)
rings368m	11:03:30	1.82	5.61	(3.09)	19:36:56	1.57	70.78	(45.01)

where 'nameN' means the scene 'name' of the size factor 'N'; 'creation' shows time for both executing an SPD command and converting its NFF output into binary form; 'base' shows rendering time without refinements (except high-resolution grids); 'S+G' and 'S+G+B' show gain factors; and the values in parentheses show the 'S+G+B to S+G' gain factors. Time is shown in hours:minutes:seconds. The view parameter is the only difference between 'rings368m' and 'rings368': The view frustum of 'rings368m' contains approximately 1 million objects.

resolution grids that may not fit in memory, achieving further speedup.

Another good point is that these refinements smoothly enhance the original algorithm and keep the whole algorithm's generality; the input data, for example, is still handled as a sequential stream of objects. This enables us to import other refinements easily in the future. For example, reordering objects spatially and combining other acceleration/cache-based schemes are interesting.

Our rules in refinements are based on $N^{1/3}$ and we use rough settings for them, but there should be better rules/settings (see the nice discussion by Havran and Sixta [Havra99]). It is interesting to investigate such rules/settings for huge data, because too much analysis of a scene leads to a too lengthy preprocess. It is also valuable to consider general and portable scene database management systems. If there is a system providing detailed information of the scene, data conversion and preprocess can be greatly shortened. Such information can be utilized also in the actual ray tracing phase.

Acknowledgments

We thank the reviewer and Ronen Barzel for polishing this paper. We also thank Eric Haines for encouraging to submit this work to JGT. This research is partially supported by The Japan Society for the Promotion of Science "Research for the Future" Program No. JSPS-RFTF97100103.

References

[Apoda00] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan*. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, 2000.

[Haine87] Eric A. Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, 7(11):3–5, November 1987. `http://www.acm.org/pubs/tog/resources/SPD/overview.html`.

[Havra99] Vlastimil Havran and Filip Sixta. Comparison of hierarchical grids. *Ray Tracing News*, 12(1), June 1999. `http://www.raytracingnews.org/rtnv12n1.html#art3`.

[Jevan89] David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. *Graphics Interface '89*, pages 164–172, June 1989.

[Klima97] Krzysztof S. Klimaszewski and Thomas W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, January 1997.

[Law96] Asish Law and Roni Yagel. Multi-frame thrashless ray casting with advancing ray-front. *Graphics Interface '96*, pages 70–77, May 1996. ISBN 0-9695338-5-3.

[Mölle99] Tomas Möller and Eric Haines. *Real-Time Rendering*. A K Peters, Ltd., 63 South Avenue Natick, MA 01760, 1999.

[Mülle92] Heinrich Müller and Jörg Winckler. Distributed image synthesis with breadth-first ray tracing and the ray-z-buffer. In B. Monien and T. Ottmann, editors, *Data Structures and Efficient Algorithms. Final Report on the DFG Special Initiative (v. 594 of Lecture Notes in Computer Science)*, pages 124–147. Springer-Verlag, 1992.

[Nakam97] Koji Nakamaru and Yoshio Ohno. Breadth-first ray tracing utilizing uniform spatial subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):316–328, October - December 1997. ISSN 1077-2626.

[Pharr97] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Proceedings of SIGGRAPH 97*, pages 101–108, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

Web Information

The images in this paper, as well as the complete numerical results of our experiments, are available online at

`http://www.acm.org/jgt/papers/NakamaruOhno01`