Breadth-First Ray Tracing Utilizing Uniform Spatial Subdivision

Koji Nakamaru and Yoshio Ohno, Member, IEEE Computer Society

Abstract—Breadth-first ray tracing is based on the idea of exchanging the roles of rays and objects. For scenes with a large number of objects, it may be profitable to form a set of rays and compare each object in turn against this set. By doing so, thrashing, due to disk access, can be minimized. In this paper, we present ways to combine breadth-first methods with traditional efficient algorithms, along with new schemes to minimize accessing objects stored on disk. Experimental analysis, including comparisons with depth-first ray tracing, shows that large databases can be handled efficiently with this approach.

1 Introduction

Ray tracing is known to be a powerful technique, and a major bottleneck of this technique—its computation time—has been greatly reduced by previous authors [1]. However, Cook et al. have pointed out one serious problem with traditional ray tracing [2]: The cost to access the scene database is not considered. This may cause trouble with a scene containing a huge amount of data because accessing it causes thrashing. This problem contrasts clearly with an advantage of the z-buffer algorithm. In the z-buffer algorithm, objects are sequentially processed and can be stored in the secondary memory. Although we have to maintain the main memory for z-buffer, its size depends only on the number of pixels on the screen. This predictable bound of required memory makes the z-buffer algorithm particularly convenient for hardware implementation [3].

This problem has not been deeply studied yet for the following reasons:

- If we do not use many large textures, each object uses a small amount of memory. One sphere requires only four floating point numbers, for example.
- The complexity of the scene database can be increased by hierarchical instancing [4], [5]. This is a powerful method and is practical to use in a very high quality rendering system [6]. Other methods based on object properties also reduce the amount of required memory [7], [8].
- Simple and effective methods are available for improving the accessing of scene databases, such as certain caching methods [9], [10], the use of coherence-space-filling curves [11], etc.

Once we exceed the capacity of virtual memory, however, ray tracing will be useless. Users may want to render data that has details of the real world without optimizing/reducing their scene databases. We want to avoid this weakness in ray tracing, and provide robustness to users.

The cause of this problem is accessing the scene database in undetermined order. Müller et al. proposed a new strategy of ray tracing to avoid the undetermined accessing [12], [3]. They call this strategy breadth-first ray tracing, a name derived from the fact that ray trees are traversed in breadth-first order. That is, first, we determine the nearest intersection points for all view rays. Then, we treat all shadow rays and determine shadows. Subsequently, we treat all rays of reflection and refraction, and so on. We hold rays, instead of objects, in the main memory and can access objects sequentially with this strategy. Breadth-first traverse for ray trees has already been used for some acceleration methods [13], [14], for vectorized ray tracing to maximize the performance of vector processors [15], and for parallel ray tracing, where each processor’s local memory is very small [16]. The work of Müller et al. was, however, the first use directly intended for handling large scene databases even on ordinal workstations.

Müller et al. made the ray-z-buffer a concrete algorithm, utilizing some special acceleration structure for this strategy. Unfortunately, the time behavior of the ray-z-buffer in a concrete implementation showed poor behavior compared with that of efficient implementations of depth-first ray tracing. A distributed version of ray-z-buffer has also been implemented to cope with this “absolute time” problem [3].

Breadth-first ray tracing is basically slower than depth-first ray tracing when the scene database is small because of the overhead for holding rays. We can get, however, a robust ray tracer by switching the type of ray tracing according to the size of the scene database (Fig. 1). The capabilities described above—the predictable bound of memory or sequential data accessing—may be important, also, on a machine with much memory space, because they increase the cache coherency. Troubles with the design of efficient algorithms might have been the reason that breadth-first ray tracing did not receive much attention in the past, in spite of those interesting features. The ray-z-buffer may be improved to some degree, but its current form is not essential to breadth-first
ray tracing. The key is to “exchange the roles of rays and the scene database,” and then, we can compose efficient algorithms based on today’s major acceleration methods.

The algorithm proposed in this paper utilizes uniform spatial subdivision [17] and has several improvements to minimize accessing objects. The whole process is outlined as follows:

- **Preprocess**: All scene data are converted into binary form including bounding boxes. We then scan all data to prepare the voxel structure.
- **Intersection tests**: Intersection tests are done reading the scene data sequentially to produce results for all rays in the current depth, including shadow, reflection, and refraction rays.
- **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 nonshadow rays, again reading the scene data sequentially.

The latter two steps are repeated either until there is no new ray or the maximum tracing depth is reached. Note that the scene data are accessed sequentially in any parts, thus causing no thrashing.

Concepts of the algorithm may be applied with other acceleration techniques, especially octree spatial subdivision [18] and adaptive/nested grids [19], [4]. However, constructing efficient nonuniform structures still remains a productive research area even in ordinal on-memory ray tracing [20] and is a difficult problem for large databases. We have currently adopted a very simple solution for a nonuniform environment. More details and results are shown in later sections.

The rest of paper is organized as follows: Section 2 describes intersection tests and shading in detail. Section 3 shows several results including comparisons with a depth-first ray tracer. We conclude and indicate several directions for future work in Section 4.

## 2 Algorithm

Our algorithm for intersection calculations is a bit complicated. We will describe the basic idea to utilize uniform spatial subdivision (called USS, below) in Section 2.1. This algorithm presents two problems, and the solution for the serious one is shown in Section 2.2. The solution for the other problem is shown in Section 2.3. Based on this algorithm for a single grid, Section 2.4 describes solutions for a nonuniform environment. Section 2.5 describes the shading part, which is also important when we handle the large scene database.

### 2.1 Basics

If we exchange the roles of rays and the scene database directly for USS, we treat the voxels as containing rays instead of objects. We then read each object and determine which voxels overlap it. If any voxel contains rays, we check the intersection between each of those rays and the object, and update each ray’s intersection information. The pseudocode is shown in Pseudocode 1.

```plaintext
main()
{
    Object o;
    Ray r;

    initialize voxels;

    while (there are unprocessed rays on the disk) {
        // storing rays into voxels
        read rays until main memory is full;
        for (each r in the main memory) {
            initialize 3DDBA of r;
            traverse voxels and store &r in each voxel;
        }
        // reading objects and checking intersections
        for (each o on the disk) {
            read o;
            determine which voxels overlap o;
            for (each voxel overlaps o)
                for (each r in this voxel)
                    if (r.mailbox != o.id) {
                        check the intersection between r and o;
                        update r.intersection_information;
                        r.mailbox = o.id;
                    }
        }
        // writing intersection information
        for (each r in the main memory)
            write r.intersection_information to the disk;
    }
}
```

### Pseudocode 1

r.mailbox corresponds to the mailbox [21] in USS, except it stores the object number. This algorithm will work, but there are some problems, one being that we cannot perform intersection tests in the order of objects along each ray. In normal USS, on the other hand, we can perform intersection tests by starting from objects nearest to the ray’s origin and terminating processing when the ray enters a voxel which is beyond the current closest intersection point. Another problem is that the algorithm consumes much memory space because of the pointers from voxels to each ray. This is a more serious problem because necessary memory space increases in proportion to the resolution of voxels.
Obviously, we do not need to store rays in voxels that no object overlaps. Voxels in which rays are stored are reduced by previously checking which voxels overlap any object. This is not enough, though (consider the situation in which all voxels are filled with some objects). To limit the amount of memory space needed, we initially store each ray in only the first few, nonempty voxels it penetrates from its origin, and process all objects as before. Then, we find the next set of nonempty voxels for each ray and process these, repeating this process as needed.

Another modification to the algorithm is replacing voxel/object overlap testing with voxel/axis-aligned bounding box overlap testing. Instead of determining precisely each time whether a voxel overlaps an object, as in Pseudocode 1, we substitute testing voxels which simply overlap the object’s bounding box. This simplifies object/voxel testing at the cost of a bit of unnecessary ray-object intersection tests. However, object/voxel testing is crucial for handling large databases, and, later, we will use the fact that each volume composed with these voxels is convex to avoid many other unnecessary tests in Section 2.3. We also point out that the bounding box for each object can be stored as six floating point values, thus minimizing disk storage and making the scene database independent of voxel resolutions, and that Craig Kolb’s RayShade ray tracing package successfully uses this definition of voxels for each object.

The new pseudocode is shown in Pseudocode 2 (see also Fig. 2).
main()
{
    Object o;
    Ray r;

    initialize voxels;
    // checking whether voxels overlap axis-
    // aligned bounding boxes
    for (each o on the disk) {
        read o;
        determine which voxels overlap the axis-
        aligned bounding box of o;
        for (each voxel overlaps the axis-aligned
            bounding box of o)
            mark the flag in the voxel;
    }
    while (there are unprocessed rays on the
    disk) {
        // preprocessing rays
        read rays until main memory is full;
        for (each r in the main memory)
            initialize 3DDDA of r;
        while (there is r whose intersection
            information is not complete) {
            // storing rays into partial voxels
            for (each r in the main memory)
                if (r.intersection_information is not
                    complete) {
                    traverse voxels and store &r into
                    several voxels whose flags are
                    marked;
                }
        // reading objects and checking inter-
        // sections
        for (each o on the disk) {
            read o;
            determine which voxels overlap the
            axis-aligned bounding box of o;
            for (each voxel overlaps the axis-
                aligned bounding box of o)
                for (each r in this voxel)
                    if (r.mailbox := o.id) {
                        check the intersection between
                        r and o;
                        update r.intersection_information;
                        r.mailbox = o.id;
                    }
        }
    // writing intersection information
    for (each r in the main memory)
        write r.intersection_information to the
        disk;
}

Pseudocode 2

r.intersection_information is complete if
1) r.intersection_information has the intersection
   point in voxels already traversed or
2) r goes outside the whole bounding box of the scene.

Pseudocode 2 shows an algorithm that can select objects along
each ray beginning from its origin and can fix the amount of
main memory space. The scene database is, however, repeat-
edly accessed until all r.intersection_information are
complete. This may cause trouble for the large scene data-
bases that we want to handle, and solving this problem is a
key point in making this algorithm practicable. We show
the solution in Section 2.2.

There is another problem that is not easily clarified—
redundant intersection tests. This is caused by both the algo-

rithm shown in Pseudocode 2 and the solution shown in
Section 2.2. We describe this problem specifically and show
the solution in Section 2.3.

We here define three terms for convenience of explana-
tion. The “pass” means the body of the inner “while” loop
in Pseudocode 2. The “store number” indicates the number
of voxels in which each ray is stored in each pass. This
number controls the amount of required memory for stor-
ing rays as described above. The “voxel boundary” indi-
cates the boundary of the volume which consists of voxels
overlapping the axis-aligned bounding box of the object.

2.2 Reducing the Number of Accesses to the Scene
Database

Assuming that the store number is a fixed number, how
many passes are needed? We experimentally found that
the number of rays which do not receive complete intersec-
tion information is reduced exponentially. Fig. 3a
shows an example for default “rings” [22], where the store
number is always one. Note that the higher grid resolution
also increases the number of passes. When we render a
small scene database, it is not necessary to pay attention to
many passes, because the cost of accessing the scene data-
bases is very small. When we render a large scene database,
however, we cannot ignore the cost. To reduce the num-
ber of passes, we increase the store number with the fol-

lowing methods:

- Using the memory space of the “finished” rays for the
  remaining rays that do not get complete intersection
  information. For example, if the store number is 1 at
  the start, we increase this number to 2 when the re-
  maining rays decrease by half. This method can keep
down the exponential reduction.

- Increasing the store number in proportion to the grid
  resolution res. For example, if we set the store number
  at one when the res is 20, we will set it at five when
  the res is 100. While this method increases the neces-
  sary memory space with O(res), it is not a serious
problem because the coefficient of the increase is small.
This method can keep the number of passes constant when the grid resolution is increased.

Fig. 3b shows the effects of above methods, where the
initial store number is one for the 20 × 20 × 20 grid and two
for the 40 × 40 × 40 grid.
The key point is making each ray go through voxels as fast as possible. There is another problem that slows down rays. Suppose that the store number is always one, as in Fig. 4a. Three passes are necessary for the ray to go through voxels that overlap both the ray and the object. Note that each object has to be checked only in the first voxel where the ray “enters” voxels for that object. We use the following method to reduce this redundancy:

1) In addition to the flag that shows the object occupation, we define six flags that show which faces of each voxel are included in any object’s voxel boundary. These flags are marked in the initial bounding box/voxel overlap testing.

2) Then, as each ray is moved through the voxel structure, the ray has to be stored only if the voxel has any objects overlapping it and
   • either the ray passes through a voxel boundary facing the ray, flagged as used by one or more objects (Fig. 4b),
   • or the ray originates in the voxel and crosses any voxel boundary (Fig. 4c).

Note that the last case must be handled because the ray never enters those voxels.

2.3 Reducing Redundant Intersection Tests

Redundant intersection tests are initiated for two reasons. One is the partial traversal method shown in Pseudocode 2. The ray in Fig. 5a has to be checked for object 1 in each pass, even if we use the mailbox, because the mailbox cannot work across each pass. Other objects beside object 1 overwrite the mailbox of the ray in Fig. 5a, for example.

Remember that we have defined voxels for each object as those overlapping the axis-aligned bounding box of the object. The volume consisting of those voxels is convex, and, so, each ray goes through this volume only once. We utilize this property to solve the problem as follows:

• Holding the last (farthest from the ray origin) voxel position where each ray was stored in the previous pass. If the voxel at this position is one of voxels for some object, we do not need to invoke the real intersection test between the ray and the object because it was done in previous passes (Fig. 5b).

Another reason for redundant tests is the increase of the store number mentioned in Section 2.2. If the store number is more than one, we have cases where we test rays against objects in some voxels which may get obscured by successful ray-object intersections in voxels closer to the ray’s origin. Redundant tests caused by the second method in Section 2.2 are necessary for keeping the number of passes constant. Those caused by the first method are also unavoidable if there is no unprocessed ray on the disk and we want to reduce the number of passes. Suppose, however, that there are many rays that we cannot hold in the main memory at once. Pseudocode 2 shows an easy scheme in which rays are divided into several sets, and each set is processed one by one, causing redundant tests due to the first method. We can reduce these tests by the following method:

• At the end of each pass, writing the intersection information for finished rays, reading unprocessed rays on the disk into the memory space of finished rays, and initializing their 3DDDAs (Fig. 5c). The store number is determined from the number of rays remaining in the main memory, with methods described in Section 2.2.

When the main memory is filled up with rays, the store number is kept as small as possible and we can reduce redundant intersection tests. The drawback of this method is that it is necessary to sort intersection information because the order of writing is different from reading the rays. The sorting is, however, much easier compared to other main operations, such as intersection calculations, shading calculations, etc.

The final pseudocode for intersection calculations is shown in Pseudocode 3.
2.4 Solutions for Nonuniform Environments

We have described the algorithm for intersection tests which utilizes a single grid. This can work efficiently for a uniform environment, but not for a nonuniform environment. As mentioned before, this may be solved with some nonuniform structures, though constructing efficient nonuniform structures for large databases is a difficult problem. For this reason, we have adopted the following simple solution:

1) Objects are grouped into several sets each of which is associated with a single grid.
2) Then, intersections for each grid are calculated as before, and the results for all grids are merged.

This solution is very easy and works fine if there are not very many grids. We show several results in Section 3.

Another solution which has not been tested yet utilizes aggregate objects [5]. The whole scene is handled as one.
grid, but primitives located roughly at the same position are associated with an aggregate object, the acceleration structure which is handled as one object. We can solve the problem and also combine other acceleration/modeling techniques with the current algorithm in this way, though this method needs more space for keeping at least one aggregate object in the main memory.

2.5 Shading

Shading is invoked after intersection tests for one tracing depth are completed. As we are already familiar with the ray-filling method described in Section 2.3, it seems possible and efficient to calculate new rays whenever an intersection is found, and add these rays into a queue in the main memory. In fact, this is possible if the database access in shading is not bottlenecked. However, shading needs the calculation of a normal vector at the intersection point, which, in turn, needs to access geometry data and might also need to calculate the local coordinate and to access large texture data. These data should be accessed sequentially, as well.

Concerning these points, we separate shading from intersection tests and delay calculations of normal vectors, etc., until shading. Intersection tests produce intersection information for each ray which is expressed with three components: the ray number, the object number, and the distance from the ray origin to the intersection point. The question is how to access three kinds of data: rays, the scene database, and intersection information.
main()
{
    Object o;
    Ray r;

    initialize voxels;
    // checking whether voxels overlap axis-aligned bounding boxes and
    // whether faces of each voxel are included in any object’s voxel
    // boundary.
    for (each o on the disk) {
        read o;
        determine which voxels overlap the axis-aligned bounding box of o;
        for (each voxel overlaps the axis-aligned bounding box of o) {
            mark the flag in the voxel;
            if (some faces of the voxel are included in the voxel boundary)
                mark the corresponding flags in the voxel;
        }
    }

    // initialize rays in the main memory and the store number
    read rays until main memory is full;
    for (each r in the main memory)
        initialize 3DDDA of r;
    determine the store number from the number of rays in the main memory;
    while (there is r whose intersection_information is not complete) {
        // storing rays into partial voxels
        for (each r in the main memory)
            if (r.intersection_information is not complete) {
                traverse voxels and store &r into the store number of voxels
                where r enters those of some objects;
            }

    // reading objects and checking intersections
    for (each o on the disk) {
        read o;
        determine which voxels overlap the axis-aligned bounding box of o;
        for (each voxel overlaps the axis-aligned bounding box of o)
            for (each r in this voxel)
                if (r.mailbox := o.id
                    && r.last_voxel is out of voxels for o) {
                    check the intersection between r and o;
                    update r.intersection_information;
                    r.mailbox = o.id;
                    }
        }

        // writing complete intersection information, preprocessing new rays,
        // and determining the store number
        for (each r in the main memory)
            if (r.intersection_information is complete) {
                write r.intersection_information to the disk;
                if (there are unprocessed rays on the disk) {
                    read one unprocessed ray on the disk and store it into r;
                    initialize 3DDDA of r;
                }
                }

            determine the store number from the number of rays in the main memory;
    }
}

Pseudocode 3
While the structure of objects differs from one another, the structure of each ray or the intersection information is a single structure. We adopt the method shown in Pseudo-code 4 in consideration of both this fact and the handling of very large scene databases.

```c
main()
{
  Object o;
  Ray r;

  while (there are unprocessed rays on the disk) {
    read rays until main memory is full;
    read intersection information corresponding to rays in the main memory;
    sorting rays and intersection information by the object numbers;
    rewind the file pointer for objects;
    for (each r in the main memory) {
      while (o.id < r.intersection_information.object_id)
        read o;
      do shading calculations;
      write color contributions and next generation rays to the disk;
    }
  }
}
```

**Pseudocode 4**

In short, we

1) divide rays and intersection information into several sets, and
2) join each set to objects.

Note that the line “write color contributions...” generates incompletely ordered elements and we have to sort them, though this is also an easy operation.

### 3 Results

We show several results in this section. Our implementation consists of small programs which are integrated by one shell script “sray.” Each program is written in C, running under Linux on a Pentium PC with 32 MB memory. The screen resolution is 512 × 512 and the number of view rays is 263,169 (513 × 513, see README in SPD package).

We made comparisons between sray and RayShade to obtain accurate absolute time and to make any extra costs clear. RayShade is a well-known fast ray tracer that is based on US$ and implements other nice techniques. In order to get the same conditions, sray’s primitive intersection testers are based on those of RayShade, and RayShade is adjusted to trace the same rays of sray using command-line options and applying some patches. In the following statistics, “rayshade-ss” denotes RayShade, which uses the same view rays and the same shading model, while “rayshade-ss-r” denotes “rayshade-ss,” which cannot use shadow caching [23] and the ray box cull in each voxel [4]. “rayshade-ss” is almost equivalent to RayShade on the “depth 0” condition, where only view rays and first shadow rays are traced. Most of scenes are entirely enclosed with one grid, where each axis’s resolution is defined as $N^{1/3} + 0.5$, where $N$ is the number of objects. The grid for balls/trees encloses all objects except the basement plane.

Table 1 shows statistics for the default scenes of the SPD package. The store number is always one in order to make the number of intersection tests the same. In terms of total time, sray time is three to five times longer than rayshade-ss time, and two to three times longer than rayshade-ss-r time. We can describe the reasons for slower results as follows:

- **Preprocess:** The contents of preprocesses differ from each other in two ray tracers. The main reason for longer time is, however, just writing onto the disk.
- **Intersection tests:** Our implementation aims to hold as many rays as possible in order to handle large scene databases efficiently, and has no ray box cull. This is one of the reasons for slower results in intersection tests. Shadow caching also makes another difference. In the part for rayshade-ss-r, however, each ray shade time is still 1.2 to 1.8 times longer. The extra costs of sray are:
  - Accessing rays stored in each voxel and related intersection information.
  - The last voxel checking described in Section 2.3.

![Table 1](image)

while (there are unprocessed rays on the disk) {
  read rays until main memory is full;
  read intersection information corresponding to rays in the main memory;
  sorting rays and intersection information by the object numbers;
  rewind the file pointer for objects;
  for (each r in the main memory) {
    while (o.id < r.intersection_information.object_id)
      read o;
    do shading calculations;
    write color contributions and next generation rays to the disk;
  }
}
Filling up rays described in Section 2.3.

Accessing rays is important because the amount of rays is much larger than that of objects in default SPD scenes.

- **Shading:** sray is three to nine times slower than RayShade. This is caused by the sorting of rays and other data, described in Section 2, and by the file I/O. Note that the costs of these operations depend not on the size of the scene database but on the number of rays.

Fig. 6 to Fig. 8 show changes in time when the number of objects increases, and Table 2 shows the size of memory allocated by rayshade-ss for each scene. For sray, the store number is defined as $\left\lfloor res/10 + 0.5 \right\rfloor$ initially, and its maximum value is defined as $res$, where $res$ is the grid resolution defined as before for both ray tracers. These graphs basically show expected results: rayshade-ss is faster where there is sufficient available memory and sray becomes faster where rayshade-ss thrashes. rayshade-ss time grows more rapidly once the swap space starts to be used. rayshade-ss is, however, also faster in tracing (total – preprocess) time for several scenes even if it is slower in the preprocess step. These interesting behaviors depend on the nature of each scene:

- **balls/tree:** There are many tiny objects concentrating at small space. These objects have to be accessed as a ray enters such space and cause many memory faults.
- **gears:** There are many obscured and never accessed objects. This makes the working set small and makes the program work even for the relatively large database.
- **mount/tetra:** There is no concentration of objects, like balls/tree, but many objects finally cause memory faults. We think that mount will cause results similar to those of tetra, though we have not made an experiment for the next size factor because it exceeds the swap space.
- **rings:** Each object decreases its size slowly as the size factor increases and occupies more voxels than other scenes. This causes the very lengthy preprocess time even for 32 MB database. The tracing time, on the other hand, shows the behavior similar to that for gears.
The next one is an interesting demonstration. Because the access to objects is totally sequential, we can easily handle them as compressed data. Fig. 9 shows the comparison between the change in time for data compressed with `gzip` and that for normal data. The costs to expand data grow in proportion to the data size, but we can handle the data even over the disk size.² Figs. 10, 11, and 12 show examples rendered from compressed scene databases and Table 3 shows statistics for these pictures.

TABLE 2

<table>
<thead>
<tr>
<th>size factor</th>
<th>number of objects</th>
<th>allocated memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>balls</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7,382</td>
<td>1.3</td>
</tr>
<tr>
<td>5</td>
<td>66,431</td>
<td>10.9</td>
</tr>
<tr>
<td>6</td>
<td>597,872</td>
<td>97.2</td>
</tr>
<tr>
<td>gears</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9,345</td>
<td>5.5</td>
</tr>
<tr>
<td>5</td>
<td>18,251</td>
<td>9.8</td>
</tr>
<tr>
<td>6</td>
<td>31,537</td>
<td>16.3</td>
</tr>
<tr>
<td>7</td>
<td>50,079</td>
<td>25.2</td>
</tr>
<tr>
<td>8</td>
<td>74,753</td>
<td>37.1</td>
</tr>
<tr>
<td>9</td>
<td>106,435</td>
<td>52.4</td>
</tr>
<tr>
<td>10</td>
<td>146,001</td>
<td>71.4</td>
</tr>
<tr>
<td>11</td>
<td>194,327</td>
<td>94.8</td>
</tr>
<tr>
<td>mount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8,196</td>
<td>5.2</td>
</tr>
<tr>
<td>7</td>
<td>32,772</td>
<td>13.0</td>
</tr>
<tr>
<td>8</td>
<td>131,076</td>
<td>44.1</td>
</tr>
<tr>
<td>rings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8,401</td>
<td>2.6</td>
</tr>
<tr>
<td>10</td>
<td>23,101</td>
<td>6.9</td>
</tr>
<tr>
<td>13</td>
<td>49,141</td>
<td>14.7</td>
</tr>
<tr>
<td>16</td>
<td>89,761</td>
<td>27.0</td>
</tr>
<tr>
<td>17</td>
<td>107,101</td>
<td>32.0</td>
</tr>
<tr>
<td>tetra</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4,096</td>
<td>1.5</td>
</tr>
<tr>
<td>7</td>
<td>16,384</td>
<td>5.3</td>
</tr>
<tr>
<td>8</td>
<td>65,536</td>
<td>21.1</td>
</tr>
<tr>
<td>9</td>
<td>262,144</td>
<td>86.2</td>
</tr>
<tr>
<td>tree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>8,191</td>
<td>2.1</td>
</tr>
<tr>
<td>12</td>
<td>16,383</td>
<td>4.1</td>
</tr>
<tr>
<td>13</td>
<td>32,767</td>
<td>8.2</td>
</tr>
<tr>
<td>14</td>
<td>65,535</td>
<td>16.3</td>
</tr>
<tr>
<td>15</td>
<td>131,071</td>
<td>32.4</td>
</tr>
<tr>
<td>16</td>
<td>262,143</td>
<td>64.7</td>
</tr>
</tbody>
</table>

Fig. 8. Changes in time for the size of the scene database. Scenes are (a) “tetra” and (b) “tree.”

Fig. 10. “mount” (compiled with `-DNEW_HASH`) of the size factor 11.
This paper has introduced an efficient algorithm for breadth-first ray tracing. This algorithm inherits features of USS and can efficiently handle very large scene databases, which may be compressed data. We can get a robust and efficient ray tracer by using both depth-first ray tracing, utilizing major acceleration methods, and breadth-first ray tracing, utilizing our algorithm.

We emphasize the point that we can combine breadth-first ray tracing with today’s major acceleration methods. Combining breadth-first ray tracing with other acceleration methods is an interesting area of research. The determination of parameters for acceleration structures and the construction of nonuniform structures which target large databases, should also be investigated in the future.

Other interesting issues include the following: Many caching algorithms for parallel ray tracing are available, but it is not clear whether those are also effective on a single processor. Actually, caching algorithms targeting very large databases have not yet been studied in detail. It is important to make good caching algorithms for delaying the breakdown, because breadth-first ray tracing has the basic overhead for holding rays. Once we get some good caching algorithms and those properties, it will also become easy to determine the switching point between depth-first and breadth-first ray tracing.

It is also interesting to implement breadth-first ray tracing on a machine with much memory space. The overhead for holding rays is especially large on a machine such as the one we used. On a machine with much memory space, we can reduce this overhead, and, then, the capabilities that increase the cache coherency—the sequential data accessing and the compressed data handling—become effective. Each object can be an aggregate object on such a machine, and we can combine breadth-first ray tracing with normal acceleration/modeling methods.

We would like to thank Kelvin Sung for notifying us Müller’s work in the early stage of this research. We also thank TVCG reviewers for their helpful comments on this work. This research is partially supported by The Japan Society for the Promotion of Science “Research for the Future” Program No. JSPS-RFTF97100103.

**TABLE 3**

<table>
<thead>
<tr>
<th>number of objects</th>
<th>data size (MB)</th>
<th>compressed data size (MB)</th>
<th>time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mount11</td>
<td>8,388,612</td>
<td>904</td>
<td>8.35</td>
</tr>
<tr>
<td>rings80</td>
<td>10,432,801</td>
<td>1,522</td>
<td>384</td>
</tr>
<tr>
<td>tetra12</td>
<td>16,777,216</td>
<td>1,808</td>
<td>255</td>
</tr>
</tbody>
</table>

“Time” does not include data generation/conversion time.

**REFERENCES**


Koji Nakamaru received a BS in mathematical science and an MS in computer science from Keio University in 1990 and 1992, respectively. Following receipt of his MS, he worked at Hitachi, Ltd. He was a research student from 1993 to 1996. Nakamaru is currently a doctoral student at the Graduate School of Computer Science at Keio University. His research interests include high quality rendering systems and management of complex scene databases.

Yoshio Ohno received a BS and an MS in administration engineering from Keio University in 1968 and 1970, respectively. Following receipt of his MS, he did research at the Institute of Information Science at Keio University from 1970 to 1988. He received a PhD in administration engineering from Keio University in 1986. He was an assistant professor from 1987 to 1994 and has been a professor at the Graduate School of Computer Science at Keio University since 1995. His research interests include computer graphics, CAGD, and DTP.