Efficiency issues on Ray Tracing Machine

Kirill A. Dmitriev
Keldysh Institute of Applied Mathematics RAS
Moscow, Russia

Abstract

Recently computer world was amazed by the explosive growth of the hardware efficiency. Average computer now has integrated hardware ability of displaying thousands triangles per second. Hi-end graphical accelerators (e.g. Sony Playstation 2) render up to 20 million triangles per second. However, importance of old good ray tracing, as the most accurate method for realistic image synthesis became not lower than, say, ten years ago. Here we consider two commonly used ray-tracing methods: regular grid and octogrid traversing. Ray-tracing speed, memory requirements and preprocessing speed are compared.

Keywords: *Ray tracing, Voxel grids, Efficiency, Benchmarks.*

1. INTRODUCTION

A number of interactive rendering techniques have evolved recently. Most of them are based on hardware-accelerated polygonal renderers. However, such renderers have a lot of limitations due to both the algorithms used and the tight coupling to the hardware. While software ray tracing methods were always more attractive in the terms of quality and supported features, speed was never their advantage. As the computers become more and more powerful, with larger number of processors, one can suggest that the speed issue will become not so important soon and precise software algorithms of image synthesis become predominant. For example [1] describes a system of 60 processors, which is able to perform interactive ray tracing (15 frames per second) of 35 million spheres. Such exceptional speed places the system higher than any hi-end polygonal renderer existing up to date.

Of course, spheres rendering has more scientific than practical interest, but the above example shows that some day ray tracing can become a most used methods in every area of computer graphics, including real-time visualization. In this paper we describe our exploration of different ray tracing techniques directed on the development of most speedy algorithm.

In general, the relative efficiency of different ray tracing algorithms may vary depending on features of the processed scene and global illumination algorithms. In this article we focus on selection of the best method for practical applications.

The environment we use for comparison of ray tracing techniques is the physically accurate global illumination algorithms (backward rendering, forward Monte-Carlo ray tracing) applied for complex realistic architectural scenes.

2. GENERAL EFFECIENCY ISSUES.

The purpose of Ray Tracing Machine (RTM) is quick finding intersection of a ray with the scene geometry. There are two different types of queries RTM should handle:

- find closest intersection point;
- find all intersection points;

Closest intersection is required for primary, reflected and transmitted rays. All intersections are required for calculation of light weakening as it goes from light to the point of interest. For efficiency reason it is important to keep those queries separate. E.g. when finding closest point, RTM can ignore all geometry hits which occurs at a distance larger than the most close of all previous hits. Thus, first query can be sufficiently optimized if, after every hit, the part of scene behind the hit point is culled.

Another important issue is effective use of processor caching. Let us suppose your algorithm needs the following information about triangle: indices of triangles vertices, triangle plane index, some flags, and bounding box. The most natural would be to allocate four arrays for each of above values. Nevertheless, it turns out that processor works better with data, which is stored in the nearby memory blocks. Most effective will be to create a single array of the following structs:

```
struct TrgInfo
  {
  int vert_ind[3];
  int pln_ind;
  UINT flags;
  float box[2][3];
  };
```

There are also other methods of low-level optimizations. General guideline on such type of optimizations can be found in [2]. In this article we would like to focus more on the higher-level optimizations, in particular on the voxel grid creation/traversing algorithms.

3. VOXELIZATION TECHNIQUES

The most time consuming operation during ray tracing is calculation of ray intersection with triangles. Voxelization is the best-known and widely used method to reduce number of those operations. The idea is to place nearby triangles in axis aligned bounding boxes (usually cubic). Voxels are located in a regular fashion to make use of Brosenhaim-like algorithms for their traversing.

Above only the general idea is given, but up to date there exists a huge amount of methods to make space traversing more efficient and reduce the number of required operation to minimum. We do not consider here more sophisticated approaches such as described in [6], [7] as they have nontrivial settings for which optimum are scene dependent and their automatic finding is a difficult problem.

We investigated three voxelization techniques:

3.1 Uniform space subdivision

The uniform subdivision is the classic approach well known in literature on Ray Tracing originated by Akira Fujimoto more than 10 years ago. The idea is to subdivide scene bounding box on to equal cubes, each scene dimension is divided on the number of cubes proportional to its length. After that a pure Brosenhaim algorithm is used for scene traversing.

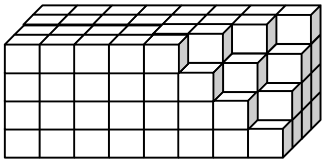

Figure 1: Uniform voxelization

The key advantage of this method is that absolutely regular space subdivision allows fast traversal of the voxels grid. Time required for search of intersected voxel is usually negligible compared to other operations.

One disadvantage of the method is non-efficiency and/or tremendous memory requirements for highly non-uniform shapes. Really, suppose that you have a scene with 100,000 triangles and having dimensions [10x10x10] meters. Suppose that 99,000 triangles are located in a small volume in the center of the scene and that you use backward ray tracing to receive an image of exactly this small volume. In this case described uniform voxelization will not give any benefit because all triangles will fall into one or two voxels, located in the center of the grid. For more-less efficient voxelization you will need to create a very dense subdivision, which is quite memory consuming.

Another disadvantage of the method is lack of adaptation and need in the external setting of space subdivision density. On practice most efficient space subdivision is a function of not only scene bounding box dimensions but also of number of triangles and their distribution in space. It is difficult for algorithm to determine how much voxels are actually required for fastest ray tracing.

3.2 Regular recursive grids

The shape of recursive regular grid is depicted on Figure 2. On the top level we have ordinary uniform subdivision. Then every voxel of uniform grid can be recursively subdivided into a fixed (same for all voxels) number of cubic subvoxels. Subdivision depth is not limited.

Approach is the special version of general EN-TREE approach, which features are:

- efficient support of arbitrary non-uniform shapes;
- lack of externally tuned parameters;

This scheme inherits the main advantage of the uniform subdivision, namely fast Brosenhaim-like voxels traversal. In the same time the scheme has high adaptivity for the scene non-uniformities.

The key feature of the approach is that the uniform voxels subdivision, present at the top level, enables fast Brosenhaim-like algorithm for the grids traversal similar to the uniform subdivision. If a traced ray goes from one supervoxel to the adjacent similarly subdivided supervoxel than almost all Brosenhaim coefficients remain valid.

It allows implementing ray transfer between adjacent supervoxels almost as fast as for uniform grids. The ray transfer between differently subdivided voxels is slightly more costly, but even in this case majority of previous Brosenhaim coefficients can be efficiently reused.

Automatic voxelization builder should solve the following tasks: decide number of voxels on top level and criteria for recursive adaptive subdivision voxels into subvoxels.

The number of voxels in the top-level uniform grid is selected automatically taking into account the scene uniformity. For highly non-uniform scenes with large empty areas it is better to have few top-level voxels. Dense top-level subdivision would decelerate rays traversal through empty spaces. In opposite for 'close to uniform' scenes it is better to create large number of top level uniform voxels and minimum subvoxels. In this way the superfluous switchings between supervoxels/subvoxels during ray tracing are avoided.

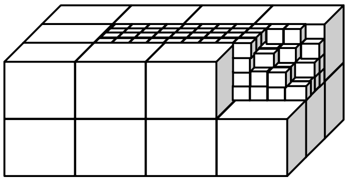

Figure 2: Regular recursive greed

Subdivision of a voxel into subvoxels is performed if the number of polygons intersected with it is larger than threshold. There are two internal parameters: N_SUB_VOXELS - number of subvoxels to which every voxel dimension is split (the same for each dimension and for all voxels) and VOX_NTRG_THR - number of triangles threshold. Optimal values for both parameters depend mainly on the respective performance of the grid traversal code and triangle intersection code. The optimum almost does not depend on particular scene. This feature allows finding the reasonable values ones by means of benchmarks and then to hardware them into source code.

It should be noted that not only ray tracing speed is valuable in above method. Varying N_SUB_VOXELS and VOX_NTRG_THR parameters it is possible to choose a rational balance between ray tracing speed, preprocessing time and memory load.

3.3 Octree grid

The method of regular grids described above is sufficiently heuristic and uses different assumptions to create a most efficient voxelization. Experience tells that that human intuition is often very wrong about what changes will make the code faster. Many factors play here, in particular features of processor operation and caching can influence speed significantly.

That is why we also implemented a classical algorithm of octree traversal. This algorithm uses an octree structure to store hierarchical voxels grid, which shape is depicted on Figure 3. The top cubic voxel has size equal to maximal of scene dimensions.

Then it is recursively subdivided each time on eight subvoxels, creating octree. Comparison with pure octree method should give an answer whether above given argumentation in favor of regular recursive grids is valid.

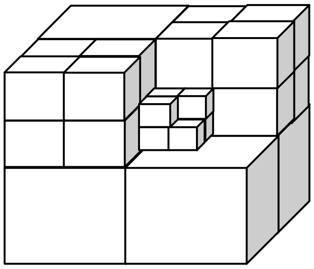

Figure 3: Octree grid

Note that octree grid is a special case of regular recursive grid. Provided that we do not use top-level uniform subdivision and set N_SUB_VOXELS = 2, we receive exactly octree grid. However, restricting ourselves to N_SUB_VOXELS = 2, we can optimize the code basing on this constant. For example, let’s consider algorithm for ray descending from supervoxel to subvoxel in the general case and algorithm optimized for the case N_SUB_VOXELS = 2.

```
double remain[3]; // distance to border of voxel by each dimension
double tot_remain[3]; // distance between adjacent voxel borders
int sub_voxel; // index of sub voxel [0, N_SUB_VOXELS ^ 3 – 1]
UINT ray_mask; // 3 bits are used - bit is 1 if coordinate of ray dir > 0
int vox_incr[3] =
  {1, N_SUB_VOXELS, N_SUB_VOXELS * N_SUB_VOXELS};

// general case
for (int ic = 0; ic < 3; ++ic)
  {
  tot_remain[ic] /= N_SUB_VOXELS;
  int t = (int)(remain[ic] / tot_remain[ic]);
  t = Min(t, N_SUB_VOXELS – 1);
  remain[ic] -= t * tot_remain[ic]
  if (ray_dir[ic] > 0)
    sub_voxel += (N_SUB_VOXELS – 1 - t) * vox_incr[ic];
  else
    sub_voxel += t * vox_incr[ic];
  }

// case with N_SUB_VOXELS = 2
sub_vox = dir_mask;
for (ic = 0; ic < 3; ++ic)
  {
   tot_remain[ic] *= 0.5;
   if (remain[ic] > tot_remain[ic])
     {
      remain[ic] -= tot_remain[ic];
      sub_vox ^= (1 << ic);
     }
  }
```

For the sake of simplicity, case when ray direction is parallel to one (or several) coordinate planes is not considered. Restricting N_SUB_VOXELS to 2 also allows to simplify code for ray switching from super voxel to adjacent similarly subdivided super voxel without any recalculation of Brosenhaim coefficients (remain[3] and tot_remain[3] in the code above).

Octree subdivision we want to create should satisfy the following conditions:

- every voxel should have most optimal amount of triangles;
- we do not want to create excessive subdivision. Obviously, that it is not good if each of eight created voxels contain the same number of triangles as the parent voxel contained;

Ray tracing algorithm with both regular recursive grid and octree grid make use of information about triangles complanarity. Thus, for triangles amount to be optimal, we use this information during voxelization construction also. Grid is built as follows:

```
create large voxel, enclosing whole scene;
for (every voxel)
  {
  threshold = VX_THRESHOLD;
  plane_index = index of the very first voxel triangle;
  for (it = 0; it < number of triangles in voxel; )
    {
    if (plane of this triangle != plane_index)
      {
      threshold -= PLANE_WEIGHT;
      plane_index = plane of this triangle;
      }
    if (++it > threshold)
      {
      subdivide this voxel on eight subvoxels;
      break;
      }
    }
  }
```

Note that triangles must be sorted by planes before applying of above algorithm. Two constants, as well as in the case of regular grids determine subdivision process. VX_THRESHOLD is equal to maximal number of triangles, belonging to different planes which voxel can contain. (PLANE_WEIGHT + 1) * VX_THRESHOLD defines number of triangles, which can happen in voxel, provided that they all belong to single plane.

4. RESULTS

We used two scenes for tuning of key method parameters. First one is relatively large interior scene, consisting of 97036 triangles lighten by 39 light sources. Second is simple rectangular room with table and chairs in the center: number of triangles is 3368, 5 lights are located by the walls. For testing ray tracing algorithm, images of resolution 800x600 were calculated PPP(Pixel Per Pixel), without any antialiasing. Computer used for tests is Intel Pentium III-450.

The following statistics was obtained for recursive regular grid method R[VOX_NTRG_THR, N_SUB_VOXELS]:

Scene1	R[10,4]	R[16,4]	R[22,4]	R[31,4]
Rendering [min:sec]	01:54	01:54	01:53	01:56
Memory [kb]	7965.5	6280.3	5110.9	4091.6
Preprocessing [sec]	10.044	8.211	6.850	5.718
Scene2				
Rendering [min:sec]	00:41	00:38	00:39	00:40
Memory [kb]	403.004	175.2	92.90	73.24
Preprocessing [sec]	0.37	0.19	0.13	0.1

The following statistics was obtained for octree method O[VX_THRESHOLD, PLANE_WEIGHT]:

Scene1	O[2,15]	O[4,15]	O[6,15]	O[8,15]
Rendering [min:sec]	02:03	02:04	02:07	02:09
Memory [kb]	16324.2	11145.8	8446.92	7003.86
Preprocessing [sec]	9.825	6.529	4.97	4.006
Scene2				
Rendering [min:sec]	00:37	00:37	00:40	00:41
Memory [kb]	172.852	109.9	80.6	60.81
Preprocessing [sec]	0. 08	0.05	0.04	0.03

Above results can be interpreted as follows:

Optimal ray tracer performance in the terms of speed, preprocessing time and memory load is achieved for regular grid at approximately [19,4] point, for octree grid at approximately [4, 15] point.

On general complex scene regular grid algorithm is faster than octogrid one (01:53 against 02:04). Most probably this is due to efficient uniform grid, utilized on upper level. Such uniform grid should save a lot of time on large, densely subdivided scenes, where octogrid method has to perform a lot of descending-ascending operations. To check this hypothesis, we disabled creation of uniform upper level grid in R[19,4]:

	Rendering [min:sec]	Memory [kb]	Preprocessing [sec]
Scene1	02:08	5759.0	7.911
Scene2	01:45	127.16	0.14

Another interesting experiment was to try R[x,2] instead of R[x,4]. Optimum VOX_NTRG_THR for N_SUB_VOXELS = 2 happened to be 19, as well as for N_SUB_VOXELS = 4:

	Rendering [min:sec]	Memory [kb]	Preprocessing [sec]
Scene1	02:08	3107.6	5.44
Scene2	01:40	75,12	0.11

Above timings show that for faster ray tracing one should use either general regular grid algorithm with relatively large N_SUB_VOXELS (more or equal to 4) or specialized octree algorithm, which allows quite fast grid descending-ascending.

On the little scenes with large amount of flat surfaces (like Scene2), our octogrid method wins a little in rendering time (00:37 against 00:38) and has a solid lead in terms of preprocessing time and memory load. Most probably, this is due to a little bit more intelligent treatment of complanar triangles.

In the future we are going to implement a ray-tracing algorithm, which gathers advantages of both methods for implementation of most efficient RTM. Most probably it should be a regular grid algorithm with intelligent treatment of complanarity information and with voxels traversing, optimized for N_SUB_VOXELS = 2.

5. ACKNOWLEDGMENTS

We acknowledge the support in part of the Russian Foundation for Basic Research under a grant entitled "Physically accurate solution of global illumination analysis" (98-01-00547).

I would like also to thank my colleague Vladimir Volevich for great help and assistance in implementation of ray tracing system and for useful conversations, resulted in many good ideas.

Also I acknowledge Integra Inc. for assistantship in benchmarks and comparisons I performed.

6. REFERENCES

[1] Parker S., Martin W., Sloan P.-P., Shirley P., Smits B., and Hansen C., *"Interactive Ray Tracing". In Interactive 3D, April 1999.*

[2] Smits B., *"Efficiency issues for Ray Tracing". Jounal of Graphics Tools, Vol. 3, No. 2, pp.1-14, 1998*

[3] A.S.Glassner. *"Space subdivision for fast ray-tracing", IEEE C.G. & A., 4(10) pp 15-22 1984.*

[4] A.Fujimoto, T.Tanaka and K.Iwata. *"Arts: Accelerated Ray-tracing system", IEEE C.G. & A., pp 16-26 1986.*

[5] D.Jevans and B.Wyvill. *"Adaptive voxel subdivision for ray-tracing", in Proc. of Graphics Interface '89 pg 164 (June 1989).*

[6] E. Jansen and W. de Leeuw. *"Recursive ray traversal", Ray tracing News 5(1) 1992.*

[7] E. Jansen. *"Comparison of ray traversal methods", Ray tracing News 7(2) 1994.*

[8] F. Cazals, G. Drettakis, and C. Puech. *"Filtering, Clustering and Hierarchy construction: a new solution for ray tracing complex scenes", Computer Graphics Forum, Vol. 14 No. 3, 1995.*

[9] S. Klimaszewski W. Sederberg *"Faster Ray Tracing Using Adaptive Grids" IEEE Computer Graphics and Applications , V. 17 N.1, 1997.*

About the author

Kirill A. Dmitriev is the PhD student of Keldysh Institute of Applied Mathematics RAS Moscow, Russia.

E-mail: kadmitr@gin.keldysh.ru