Cost Distribution Prediction for Parallel Ray Tracing

Erik Reinhard[1], Arjan J. F. Kok[2], Alan Chalmers[3]
[1,3]Department of Computer Science
University of Bristol, United Kingdom
[2] TNO Physics and Electronics Laboratory
The Hague, The Netherlands

Abstract. Realistic rendering algorithms such as ray tracing are computationally very expensive. The time complexity is determined by a multitude of factors such as image size, geometry, material properties and the quality of the required rendering. This paper presents an algorithm to estimate the cost of ray tracing a scene. Assuming an octree spatial subdivision, the cost per voxel is predicted. For parallel processing purposes, this cost distribution can then be used to calculate an initial data distribution, such that the workload per processor should be more or less equal. The method avoids intersection tests, because those may affect the short response times required for preprocessing stages.

1 Introduction

Ray tracing is a time consuming process, where the rendering time of each scene will be influenced by a multitude of factors, including the size of the image, the quality of the rendering and the geometry and material properties of the environment. Whereas the size and quality of the imagery is relatively easy to assess, the impact of geometry and surface properties on the total rendering time is much more difficult to determine.

For parallel ray tracing, it would be beneficial if it were possible to have such an assessment for each sub part of the environment. That would enable the creation of a suitable data distribution, giving each processor similar amounts of data which will also be accessed a similar number of times. This would help to balance the load distribution across processors and therefore increase efficiency and scalability.

Such an analysis stage should be fairly accurate and in addition it should return an estimate within a short period of time. Rendering a low resolution image, i.e. profiling, is an obvious way of estimating the cost of a larger image and should meet the criterion of accuracy. However, since the complexity of the environment is unknown, rendering just a few rays may already take a long time. Therefore, an alternative method is chosen here which does not involve computing intersections between rays and objects. This paper demonstrates that without profiling an accurate estimate of the cost per sub part of the environment can be obtained.

Given that it is relatively inexpensive to generate an octree for a scene, this octree could be analysed. Previous research has shown that it is possible to produce an estimate of the cost (in number of intersection tests) of a single ray based on an octree [5]. The research described in this paper extends this previous idea by predicting the total

[1]E-mail: reinhard@cs.bris.ac.uk
[2]E-mail: kok@fel.tno.nl
[3]E-mail: Alan.Chalmers@bris.ac.uk

number of rays generated for each voxel. These two factors together can be used to give an estimate for the total number of intersection tests to be performed for a single image.

The impetus for this work is to be able to use a cost prediction algorithm to subdivide the scene over a number of processors to facilitate parallel ray tracing. If the cost per voxel is known in advance, the voxels could be distributed over the processors in such a way, that the workload per processor is more or less similar. In data parallel approaches, having an equal workload per processor is crucial to retain any significant performance.

In demand driven approaches, identifying the voxels with the highest cost should be used to replicate those voxels over all available processors (cache pre-loading). Data fetches thus are minimised for the most often used voxels. The remainder of the voxels could be distributed, allowing very large scenes to be rendered.

This paper is organised in a number of sections which explain the different steps involved in computing the required cost estimate (sections 3 through 7). Then results and conclusions are presented in sections 8 and 9. First, however, some previous research.

2 Previous work

Previous research has mainly focussed on generating optimal spatial subdivisions. To this extent Cleary and Wyvill [1] derive an expression that confirms that the time complexity is less dependent on the number of objects, but more on the size of the objects. They calculate the probability that a ray intersects an object as function of the total area of the voxels that (partly) contain the object.

MacDonald and Booth [4] use a similar strategy, but refine the method to avoid double intersection tests for objects that cross voxel boundaries. In addition, they propose a scaling factor to account for the fact that a ray may terminate early due to intersection with another object. Empirical evidence suggests that this factor is proportional to the density of objects in the scene.

Finally, Subramanian and Fussel [6] estimate the average number of cells traversed by a ray and the average probability that a ray intersects an object in a cell. This is accomplished by using the projected area of the box enclosing the objects in a cell.

As argued in [5], a low resolution ray tracing pass could be used to obtain a reasonable estimate of the total cost of ray tracing a larger image. The drawback of this method is that the whole procedure of data preparation and resource scheduling has to be performed. Another drawback is the fact that the complexity of the rendering need not necessarily scale linearly with the size of the image. Adaptive aspects of rendering (super-sampling for instance) and the exploitation of coherence make that for larger images the average time taken to compute a pixel value, decreases. Finally, low resolution rendering could be quite a costly process, making it a less suitable profiling method.

In an earlier stage of the process it would be helpful to get an initial estimate which would not necessarily incorporate all rendering details, but would return a quick answer. In the next step, shading parameters could be included. This leads to the following two step approach:

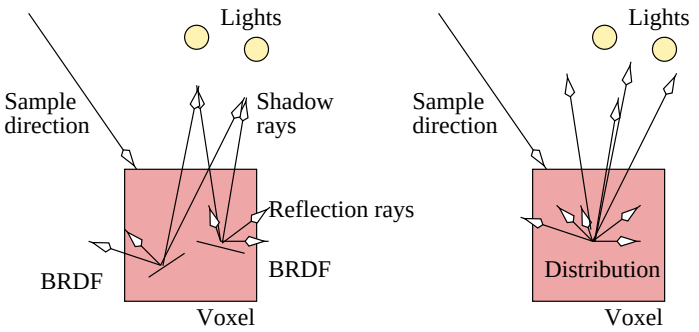

Fig. 1 Generating a reflection distribution function for a voxel. Light sources, geometry and material properties are taken into account.

- Estimate the average cost per ray on the basis of averaged geometrical properties of the scene (introduced in [5]).
- Estimate the number of rays using geometric properties as well as surface properties (brdf's) and the position of the light sources (this paper).

In [5], a method for computing the average cost of tracing a ray through an octree (the first step) was introduced. This algorithm proceeds by first computing the average depth $\bar{D}$ of the leaf cells of the octree and weights these depths by the surface area of the cells (k is the number of leaf nodes in the octree and h_i is the depth of the i^{th} leaf node):

$$\bar{D} = \frac{\sum_{i=1}^{k} h_i 4^{-h_i}}{\sum_{i=1}^{k} 4^{-h_i}}$$

The probability that the objects in a voxel block a ray entering the voxel, is estimated to be:

$$p_i = \sum_{j=1}^{n_i} \frac{A_{b_j}^x + A_{b_j}^y + A_{b_j}^x}{A_{h_i}^x + A_{h_i}^y + A_{h_i}^x}$$

Here, A_h is the surface area of an octree cell and A_b is the surface area of an object's bounding box. These blocking factors p_i are weighted and averaged:

$$\bar{p} = \frac{\sum_{i=1}^{k} p_i 8^{-h_i}}{\sum_{i=1}^{k} 8^{-h_i}}$$

The ray traversal cost is estimated using the average tree depth and the average blocking factor by summing the probabilities that a ray is blocked in the i^{th} voxel but not in the $i-1$ preceding voxels:

$$C_t \cong \sum_{i=1}^{\lfloor 2^{\bar{D}} \rfloor} ip(1-p)^{i-1} + 2^{\bar{D}}(2^{\bar{D}} - \lfloor 2^{\bar{D}} \rfloor)p(1-p)^{2^{\bar{D}} - \lfloor 2^{\bar{D}} \rfloor} + 2^{\bar{D}}(1-p)^{2^{\bar{D}}}$$

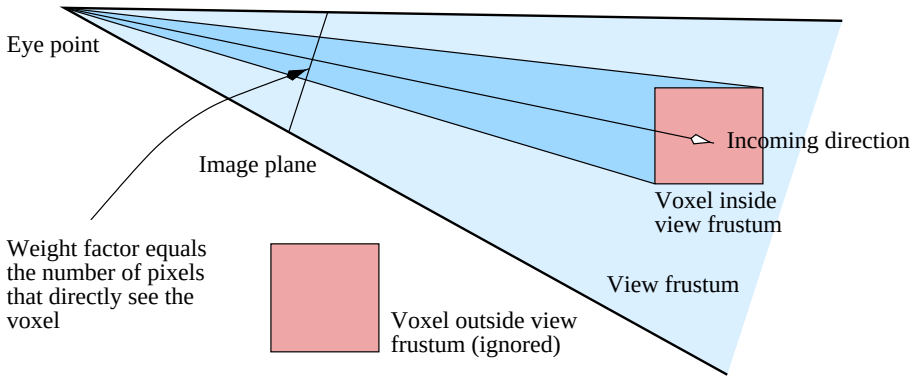

Fig. 2 Inserting primary rays into the estimation process

Finally, the number of ray-object intersections for a single ray is estimated to be :

$$C_{ray} = \alpha C_t + \beta\, \bar{m}\, C_t$$

where $\bar{m}$ is the average number of objects per leaf node and α and β are algorithm and machine dependent constants.

3 Algorithm

The algorithm proposed here estimates for each voxel in the octree the number of rays traversing through it. There are two main steps to achieve this. First, the number of primary rays incident upon each voxel is estimated. If a voxel receives a number of primary rays, in preparation for the next step, a distribution of outgoing secondary rays is computed. This depends on the brdf's of the objects within each voxel, their orientation as well as the orientation of the voxel with respect to the viewing frustum and the view point (see figure 1). This step is explained in more detail in section 4.

The result of the first step is that each voxel has stored zero or more primary rays and a distribution of outgoing secondary rays. In the second step, each voxel may receive a number of secondary rays from other voxels. The ray count for each voxel is incremented accordingly and based upon the incoming rays from other voxels, each voxel creates a new distribution of tertiary outgoing rays. This step may be repeated a number of times to reflect the depth of the ray tree during rendering. This step is explained in some more detail in section 5.

4 Primary ray distribution

Creating a distribution of outgoing rays for a voxel starts by clipping the voxel against the viewing frustum. The voxel is then projected onto a screen area. The number of pixels the voxel projects onto is a measure for the number of primary rays traversing the voxel and is therefore used to estimate the number of primary rays for that voxel (figure 2).

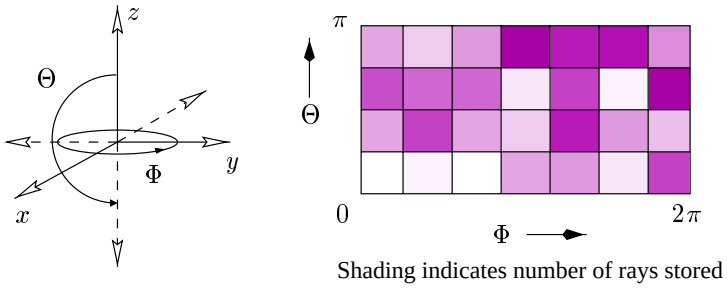

Fig. 3 Discretising a sphere

Based on the direction vector from the view point to the centre of the voxel, a distribution of outgoing rays is computed. This will be used in subsequent steps, where rays are exchanged between pairs of voxels. In order to store for each direction the number of secondary rays that are generated within the voxel, a discretised sphere is placed around the voxel. It is stored as a 2D array of polar coordinates (see figure 3).

Each object in the voxel is now tested in turn to see where secondary rays would be shot if the incoming ray direction were determined by the vector pointing from the viewpoint to the centre of the voxel. Shadow rays and specular reflection and refraction are taken into account, as well as diffuse reflection and refraction (see figure 1). If a general brdf is specified for a particular surface, this is used instead. The secondary ray directions obtained this way, are used to index the 2D array. Each direction that is found, is incremented by the number of primary rays incident on the voxel.

This procedure is followed for each voxel and, therefore, the time complexity of this step is $O(N)$, where N is the number of leaf voxels in the octree.

5 Secondary and higher order ray distributions

For each cell, the number of incoming secondary rays from other voxels needs to be determined next. This will then constitute a first estimation of the cost of the voxel. The number of rays received by a voxel is determined by the number of rays that each of the remaining voxels sends towards this voxel (figure 4), as determined by the previous step. By summing those contributions, the required cost estimate can be improved. In addition, given the incoming rays and the geometry within the voxel, a new ray distribution can be computed for the receiving voxel to be used in subsequent repetitions of this step. This step should be repeated a number of times to reflect the maximum ray depth during rendering.

6 Intersection tests

As no intersection tests are performed to compute a ray distribution, interreflection and shadowing between objects in a voxel, is not accounted for so far. Within each iteration, the total number of rays n that were distributed at the start should be the same

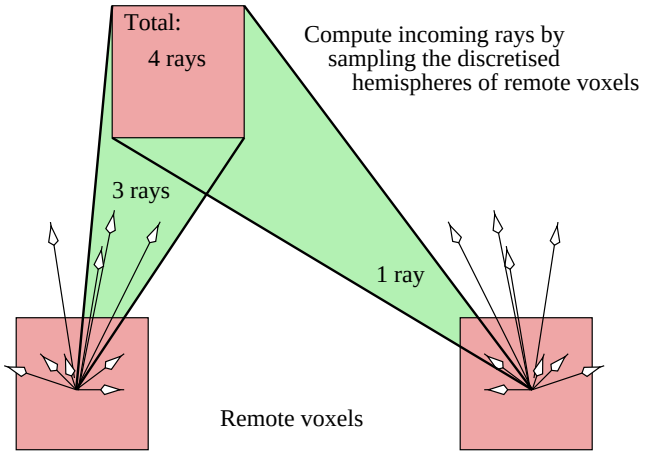

Fig. 4 Cost estimation for leaf nodes

as the total number of rays m received by all voxels. Since there are no intersection tests done, a ray will traverse several voxels and will therefore contribute to more than one voxel, so that $m > n$ will generally hold. The new ray distribution for each voxel should therefore be scaled by a factor of $\frac{n}{m}$.

An alternative method to account for intersections would be to scale the ray distribution function after each iteration by a factor which accounts for the probability that a ray travelling from voxel i to voxel j will not reach j, but may be blocked by a surface in a voxel inbetween. If $\bar{p}$ is the average blocking factor (see section 2), computed by averaging the blocking factors of all voxels, and d is the number of voxels traversed between i and j, then the probability that a ray emanating from i reaches j is given by $dp(1-p)^{d-1}$. Scaling the new ray distribution by this factor takes intervening intersections into account.

Currently, the first option is implemented, since it is cheaper. Testing the second one remains future work.

7 Controlling time complexity

This section discusses the time complexity of the algorithm presented in the previous sections. All computations work with leaf voxels which are not empty (of which there are N). The different steps of the algorithm have the following time and memory complexities:

Initialisation This step allocates memory for each of the voxels. The time complexity is therefore $O(N)$. The amount of memory used depends on the size of the array used to store the average reflection distribution function. If the number of outgoing directions is k, then the memory consumption is bounded by $O(Nk)$.

Incoming primary rays For each voxel the number of primary rays incident on that voxel is computed. This step is $O(N)$ in time.

Secondary ray distribution The time complexity of the ray distribution step is $O(N^2)$ as each voxel sums the contribution of all the other voxels. For very large scenes this time complexity may become problematic, as it is the only step that has quadratic time complexity.

In order to limit the complexity of the algorithm, a maximum depth of the octree may be specified. Subtrees below this level are treated as one larger voxel. If D is the maximum tree depth allowed, then the number of leaf voxels is 8^D and the time complexity is then bounded by $O(8^{2D})$.

A modest change to the algorithm is required to account for the fact that some internal nodes are now treated as leaf nodes. If a ray enters a leaf node, the ray counter in the leaf node is incremented. For internal nodes, adding one ray to the total number of rays traversed through this node, may not be enough, as the ray would have traversed a larger number of leaf nodes in the subtree beneath the internal node. To account for this, whenever an internal node receives a number of rays, this number is multiplied by C_t, the average ray traversal cost for that particular subtree (see section 2).

As the secondary ray distribution step is the most complex one, the total complexity of the ray estimation algorithm is $O(N^2)$, or $O(8^{2D})$ if the octree depth is limited to D.

8 Implementation and results

The work was implemented using Radiance [7]. A separate program was created which computes the estimates and the renderer itself was augmented to gather statistics of the actual renderings. For each leaf voxel in the octree an estimate of the number of rays passing through is computed. If a ray traverses more than one voxel, it is counted in all voxels traversed. Therefore, the total number of rays counted using this method can be substantially higher than the actual number of rays. However, this is a necessity, as it allows us to compare renderings with our estimates.

The results presented below have been obtained using a variety of models, which are depicted in figure 5. Statistics characterising these models are given in table 1. All images were estimated and rendered at a resolution of 512 by 512 pixels. In addition, to speed up the estimation process, the depth of the octree was limited to 2 (see section 7), leading to a maximum of 64 voxels for which interactions were computed. In practice, this typically leads from 20 to 40 leaf voxels. Note that the subtrees beneath level three are not truncated, but the 2D array for storing outgoing ray information is located at that level and represents the accumulated number of outgoing rays for the entire subtree. The outgoing directions are stored in a 2D array of 6 by 12, which is fairly small. However, tests have shown that increasing this resolution does not significantly alter the results.

For each voxel a data pair (estimated number of rays e and number of rendered rays r) was gathered and using the method of least squares, a straight line of the form

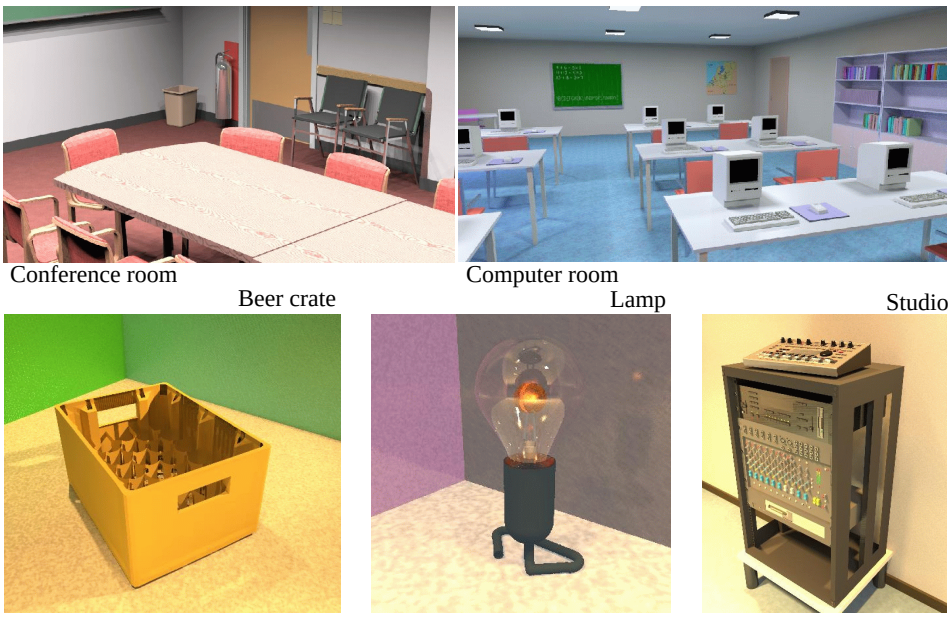

Fig. 5 Test scenes

	Beer crate	Lamp	Computer room	Conference room	Studio
Objects	12715	228	11129	2575	5291
Light sources	12	3	30	120	151
Voxels	157369	34081	84049	9905	63945
Internal voxels	19671	4260	10506	1238	7993
Leaf voxels	79373	26831	48356	6488	37251
Empty voxels	58325	2990	25187	2179	18701
Min level	2	3	3	3	2
Max level	13	11	11	13	14

Table 1 Octree statistics. The minimum and maximum level indicate the minimum and maximum level in the octree that hold empty or non-empty leaf nodes. Note that in some models instancing is used, leading to a relatively low object count (conference room and studio).

	Beer crate	Lamp	Computer room	Conference room	Studio
$\hat{\alpha}$	24129.4	21173.1	27180.2	19149.8	25675.4
$\hat{\beta}$	1.047	0.197	1.148	0.793	0.799
$\hat{\sigma}$	0.024	0.011	0.016	0.010	0.029
$Time$	0:37	0:12	0:28	0:03	0:09

Table 2 Results for primary rays only. Running time for the cost estimation algorithm is given in minutes:seconds format.

	Beer crate	Lamp	Computer room	Conference room	Studio
$\hat{\alpha}$	202708.2	50321.7	61976.0	135913.7	36459.0
$\hat{\beta}$	0.508	-0.015	0.135	0.145	0.116
$\hat{\sigma}$	0.030	0.001	0.001	0.001	0.0007
$Time$	3:52	0:30	8:33	1:45	5:09

Table 3 Results for primary and secondary rays.

$\hat{e} = \hat{\alpha} + \hat{\beta}\hat{r}$ was derived. For an optimal estimation, the slope of the line $\hat{\beta}$ should be close to 1 and $\hat{\alpha}$ should be close to 0. However, for the estimate to be useful as an indicator for data distributions in parallel ray tracing, $\hat{\alpha}$ and $\hat{\beta}$ need not be close to 0 and 1, respectively. Instead, the closeness of all the data points to this line needs to be evaluated. Assuming that the distribution of the estimates given the actual number of rays rendered, is the normal density [2]:

$$w(e_i|r_i) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left[\frac{e_i-(\alpha+\beta r_i)}{\sigma}\right]^2} \quad -\infty < e_i < \infty$$

(e_i is the estimate for voxel i and r_i is the number of rays that traversed through that voxel during rendering) the maximum likelihood estimate of σ, which should be minimised, is given by

$$\hat{\sigma} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\Big[e_i - (\hat{\alpha} + \hat{\beta} r_i)\Big]^2}$$

In order to assess the accuracy of the first step, the images were rendered and estimated using only primary rays. As subsequent steps depend on the accuracy of the primary ray distribution, these tests are deemed important. In table 2 values for $\hat{\alpha}$, $\hat{\beta}$ and $\hat{\sigma}$ are given (as well as the execution time of the algorithm[1]). To visually verify these results, graphs for all models are given in figure 6 (left column). In this figure, the voxels are displayed along the horizontal axis. Voxels that are nearby in these graphs are spatially close to each other as well, but otherwise the numbering is irrelevant. For the crate model, the raw estimates and rendering results are given in table 4.

From the graphs in figure 6 it is clear that there is sufficient correlation between the estimates for primary rays and the numbers obtained during rendering. The maximum likelihood estimators presented in table 2 support this view: the estimation generally

[1]Timings in tables 2 and 3 were obtained by running the estimation algorithm on a Silicon Graphics O2 with a 180 MHz. R5000 processor.

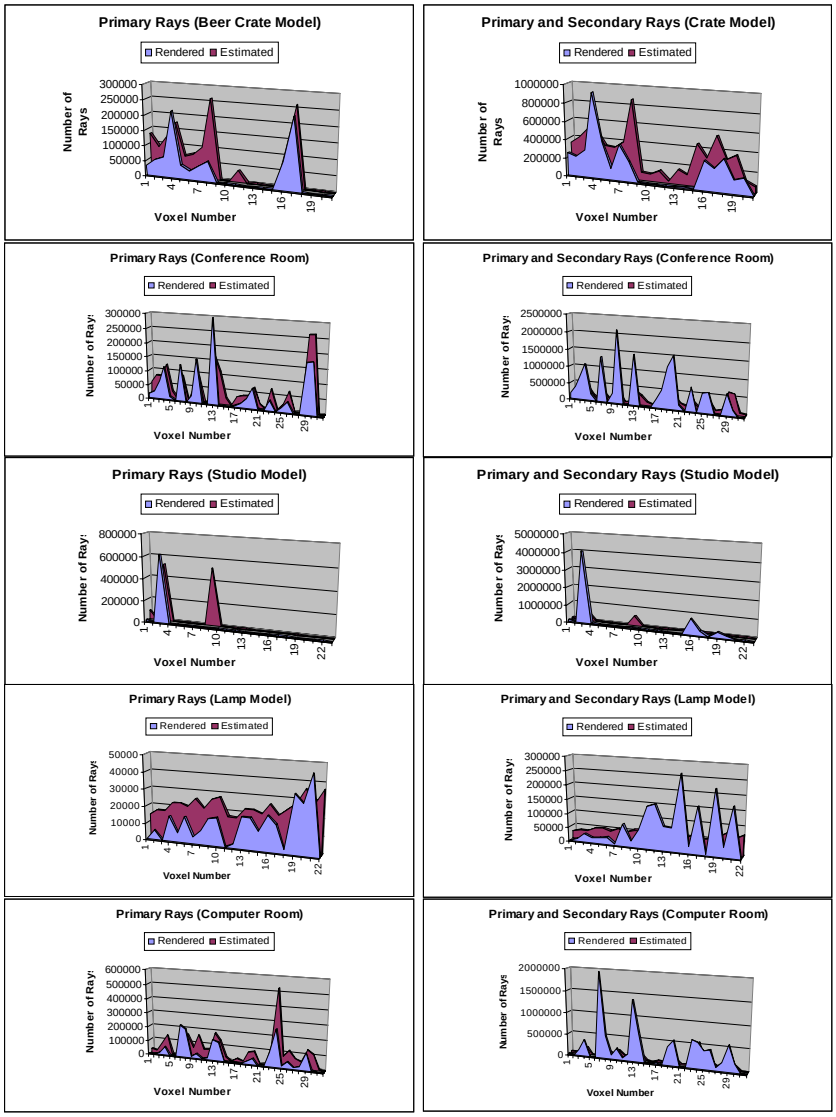

Fig. 6 Results obtained for all five models: primary rays only (left) and primary and secondary rays (right).

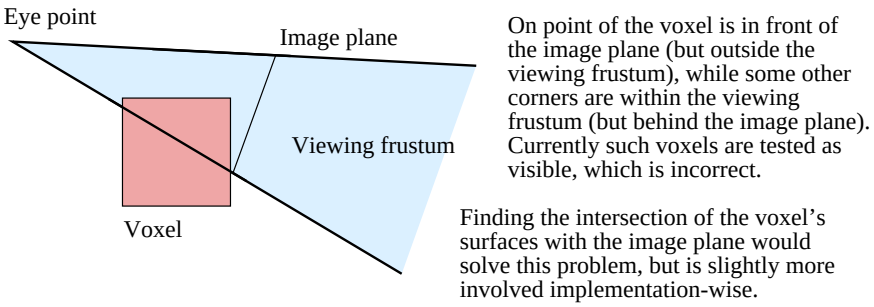

Fig. 7 Implementation issues relating to estimating primary rays.

follows the computed values, although the absolute values are still some way off (this is characterised by the $\hat{\alpha}$ and $\hat{\beta}$ values in this table).

Occasionally the estimation algorithm assigns a large number of primary rays to a particular voxel where there are none during rendering (see for example voxel #11 in table 4. This problem is due to the way voxels are treated that are partially in front of the viewing plane and partially behind (see figure 7). These voxels are now estimated to be completely visible, which is clearly an over-simplification.

Second, the total number of primary and secondary rays was estimated. The method to assess these results is the same and they are given in table 3. The accompanying graphs are given in figure 6 (right column). With the exception of the lamp model, most secondary rays in all models seem to be shadow rays. Transparency and specular reflection is fairly limited (as is to be expected in scenes modelling interiors). It is therefore understandable that the voxels containing light sources show peaks in the rendered and estimated number of rays. Because shadow rays can be aimed quite accurately, the estimation algorithm does not seem to have many difficulties identifying which voxels receive secondary rays. Compared to the $\hat{\sigma}$ estimators for primary rays, the ones for primary plus secondary rays, are smaller. The estimation therefore follows the numbers obtained during rendering better (with the exception of the beer crate model). The absolute estimated values, on the other hand, are further off from the rendered ones (which is characterised by $\hat{\beta}$ in table 3), which is to be expected: the secondary ray estimates are based in part on the results from the primary ray estimation step.

Finally, this method produces approximations only and should be treated as such. All the steps introduce errors for the sake of speed. Sometimes a higher accuracy can be obtained at the cost of using more memory and a longer computation time; for example by using larger arrays to store outgoing directions at each voxel, or using a higher and/or adaptive level in the octree to estimate distributions. This feature and its implications were not explored any further for this paper.

The voxel based approach introduces further deviations from the distribution of rays during rendering. For example, interreflections within a single voxel are not accounted for. Also, all the objects within a voxel are assumed to be concentrated in the centre of the voxel. This has an influence upon the estimated outgoing directions. When the step which computes the distribution of secondary and higher order rays is repeated, this

Voxel number	Primary Rays		Primary and Secondary Rays	
	Estimated	Rendered	Estimated	Rendered
1	130,986	32,677	343,290	247,459
2	90,666	53,266	412,958	218,049
3	130,112	66,175	519,602	293,750
4	174,251	218,948	486,914	925,611
5	67,660	46,227	336,319	408,437
6	76,797	29,746	333,153	127,997
7	101,370	49,816	399,623	396,553
8	261,121	65,973	863,053	225,320
9	0	0	89,606	8,337
10	0	0	74,612	4,871
11	38,360	0	123,645	2,314
12	0	0	27,379	2,591
13	0	0	159,769	12,219
14	0	0	115,161	11,254
15	0	0	453,448	11,167
16	126,217	91,529	317,208	319,983
17	262,144	233,187	558,592	242,577
18	0	0	316,380	357,013
19	0	0	375,796	146,729
20	0	0	119,878	179,315
21	0	0	70,661	5,758
22	0	0	70,661	5,758

Table 4 Data obtained from estimating and rendering the crate model.

error will accumulate. As in our tests shadow rays contribute most to the estimation, these sources of errors do not appear to be excessive.

9 Conclusions

The aim for this work was to be able to estimate the cost per voxel in ray tracing applications. To this end, for each voxel the number of rays traversing through them was estimated. The correlation between rendering and estimation suggests that this method is usable for generating data distributions. This is important for both data parallel and demand driven ray tracing.

In data parallel ray tracing, the data needs to be distributed across a number of processors. Each processor traces rays through its own voxels and if no intersections occur, a ray is migrated to a neighbouring processor which will resume tracing that ray. Normally, this approach leads to severe load imbalances, which are mainly due to primary rays and shadow rays forming hot spots. Scalability is therefore restricted to only a few processors. If the voxels were distributed in such a way that the workload per processor is roughly the same, scalability of data parallel ray tracing would extend to far more processors. Our cost estimation algorithm may help to achieve this goal, without introducing the need for data redistribution (which is better avoided anyway).

If large scenes are to be rendered, demand driven algorithms normally break down, because the scene data needs to be distributed. As each processor needs most, if not all, data at some time during the computation, data fetching becomes inevitable. This

may well limit the performance of such approaches. Caching techniques are therefore employed to relieve the strain on the network. Our algorithm may help to decide which data is to be loaded in the caches, as it is straightforward to identify which voxels are going to be traversed most often.

After further development, other applications may become important as well. Much in the same way as currently 3D models can be obtained from free and commercial repositories on the internet, it is expected that in the future not only the basic geometry is distributed via the internet, but that internet based companies will provide a rendering service delivering high quality rendering to customers, who will themselves provide the geometric input. These companies will mainly render scenes that would take too much processing time to be rendered by their customers.

Before an image can be rendered, both parties will have to agree on the amount to be paid and the time taken to render an image. It is therefore important for companies to be able to produce quotations for customers based on an indication of the total rendering time. Such information could also be used to improve the scheduling of tasks on a render farm [3].

Unfortunately, the absolute numbers obtained with our cost prediction algorithm are not yet close enough to the correct numbers of rays per voxel. It is therefore not yet possible to use this method to determine the overall cost of ray tracing. As such, it is not yet possible to use this algorithm for scheduling render jobs or providing quotations for customers wishing to buy renderings of their own scenes. This, therefore, remains work for the future.

Acknowledgements

We would like to thank Simon Wilkins for his valuable thoughts and discussions. Also thank you to Jan Eek and Greg Ward Larson for allowing us to use their models. This work was sponsored in part by the European Commision under TMR grant number ERBFMBICT960655.

References

1. J. G. Cleary and G. Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, (4):65–83, 1988.

2. J. E. Freund. *Mathematical Statistics*. Prentice-Hall International Editions, fifth edition edition, 1992.

3. A. J. F. Kok, J. van Lawick van Pabst, and H. Afsarmanesh. The 3D object mediator: Handling 3D models on internet. In B. Hertzberger and P. Sloot, editors, *High Performance Computing and Networking (Proceedings HPCN Europe 1997)*, volume 1225 of *Lecture Notes in Computer Science*, pages 147–156, Vienna, Austria, April 1997. Springer-Verlag.

4. J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, (6):153–166, 1990.

5. E. Reinhard, A. J. F. Kok, and F. W. Jansen. Cost prediction in ray tracing. In X. Pueyo and P. Schroeder, editors, *Rendering Techniques '96*, pages 41–50, Porto, June 1996. Eurographics, Springer Wien.

6. K. R. Subramanian and D. S. Fussell. Automatic termination criteria for ray tracing hierarchies. *Graphics Interface '91*, pages 93–100, 1991.

7. G. J. Ward. The RADIANCE lighting simulation and rendering system. In A. Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 459–472. ACM SIGGRAPH, ACM Press, July 1994.