Pyramid Clipping for Efficient Ray Traversal

Maurice van der Zwaan, Erik Reinhard, Frederik W. Jansen

Faculty of Technical Mathematics and Informatics,
Delft University of Technology, Julianalaan 132,
2628BL Delft, The Netherlands

Abstract: Rays having the same origin and similar directions frequently appear in the form of viewing rays and shadow rays to area light sources in ray tracing, and in hemisphere shooting or gathering in radiosity algorithms. The coherence between these rays can be exploited by enclosing a bundle of these rays with a pyramid and by classifying objects with respect to this pyramid prior to tracing the rays. We present an implementation of this algorithm for a bintree spatial subdivision structure and compare the performance with a recursive bintree traversal and the standard grid traversal algorithm. In parallel implementations the technique can be used to create coherent intersection tasks, allowing demand-driven scheduling with low communication overheads.

1 Introduction

With high quality rendering becoming more and more widespread, the demand for shorter rendering times increases. Both ray tracing and ray tracing-based radiosity algorithms may satisfy the needs of users in terms of quality, but still lack behind when it comes to speed of execution. Spatial subdivision techniques have greatly improved the efficiency of ray tracing but further optimizations in that direction will be difficult to achieve.

Parallel processing offers an interesting alternative to further speed up the ray tracing process. A problem, however, occurs with object models that are too large to be replicated at each processor memory. Then object data will have to be communicated to the intersection tasks or the tasks will have to be brought to the data. In both cases severe communication overheads and/or load unbalances may be introduced. Efficient parallel rendering can only be achieved when enough data coherence is present in the handling of subsequent ray intersection tasks [1].

A way to create coherent ray intersection tasks is to bundle neighboring rays and to pre-select those objects that are likely to be intersected by this bundle of rays (see figure 1). In Shen et al. [2] such a technique is used to schedule ray tasks on a number of demand-driven intersection processors. Shooting a coherent bundle of rays is a powerful computing primitive that can be applied to tracing primary rays, tracing shadow rays for area light sources and in ray tracing based radiosity algorithms for shooting or gathering energy from one patch to another. Also with brdf-reflection models, incoming rays will spawn a large number of coherent reflection rays.

Earlier ray coherence methods exploiting the notion that rays with similar directions and origins, are likely to intersect the same objects, can be found in [3, 4, 5]. Also shaft culling [6] is a method that classifies objects as in or outside of a shaft constructed,

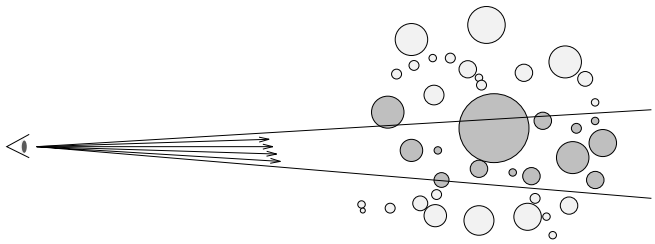

Fig. 1. Coherence between rays.

for instance, between a surface and a light source, or between a shooting patch and a receiving patch.

Recently, Greene published an algorithm for intersecting arbitrary convex polyhedrons with rectangular solids that can be used to efficiently cull polygons that are inside a viewing pyramid from an octree spatial subdivision structure [7]. The viewing pyramid is intersected with the octree and each (axis-aligned) cell of the octree is recursively tested for overlap with the pyramid. A maximum of three tests may be needed to conclude if the pyramid intersects with a cell. In the first test, the bounding box of the pyramid is tested against the cell. If some overlap is found, the second test is invoked, which tests on which side of each of the planes making up the pyramid the cell lies. If the cell is inside any of these planes, the third test is necessary. In this test, the cell and the pyramid are projected onto the three orthographic planes. If in all three views the projection of the cell is outside the projection of the pyramid, then the two do not intersect.

We have been experimenting with a similar algorithm that we originally gave the name of cone clipping [8] and recently renamed in pyra-clip, an abbreviation of pyramid clipping. Our method only differs from Greene's method in that a bintree is used instead of an octree, and for the second and third test a Cohen-Sutherland clipping test is used to determine whether the cell intersects the pyramid. Our version is explained in more detail in sections 2 and 3. In section 4 we compare the method with two regular single-ray traversal methods, one for a bintree and one for a grid. The results are discussed in the final section.

2 Pyramid - bintree intersection

The pyra-clip algorithm assumes the presence of a bintree structure. This structure is created in a preprocessing step. The pyra-clip algorithm itself consists of two parts. In the first part, a pyramid is created (this can, for instance, be the viewing pyramid or a subpart of it) that encloses a number of rays having the same origin and similar directions. This pyramid is intersected with the bintree structure. This results in an ordered list of bintree cells that are (partly) inside the pyramid. Then, in the second step, the individual rays that formed the pyramid are traced through the cells. The algorithm is done when for each ray an intersection is found. The first step is discussed in this section, while the ray traversal is discussed in section 3.

Central to the first step is the classification of each bintree cell to the pyramid. This classification uses a variant of the Cohen-Sutherland line clip algorithm. To this extent, the bintree is recursively tested for intersection with the pyramid. Each level requires

at most two tests to determine whether an intersection occurs between a cell and the pyramid.

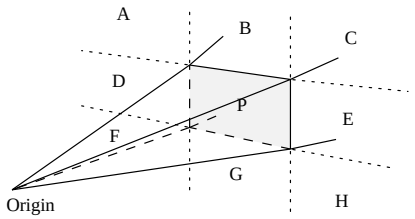

Fig. 2. The planes defining the pyramid generate nine subspaces.

The first test examines the position of the vertices of a cell with respect to the planes of the pyramid. The four planes of the pyramid define nine subspaces which may contain one or more of the vertices of the cell, see figure 2. The position of the vertices is derived from the distances of the vertices to the planes of the pyramid. The following cases now may occur:

- All distances are positive indicating that the vertices are inside sector P (see figure 2). This means that the cell is completely contained within the pyramid.
- Not all vertices are inside sector P. The cell will be partially inside the pyramid.
- The vertices are in three consecutive subspaces, excluding P (for example A-B-C or C-E-H). Now the cell will be completely outside the pyramid.
- The vertices are in both subspace B and G or in both D and E. Because of the convexity of the pyramid, the cell is partially inside the pyramid.
- The vertices are located in the subspace A, C, F and H. This means that the pyramid is contained within the cell.

These tests can be efficiently implemented with bitwise operations. For each vertex of the cell, a bitmask is set, indicating which of the subspaces it is in. The bitmasks for each vertex are OR-ed together, resulting in a bitmask indicating the position of the entire cell with respect to the pyramid.

All these tests are relatively simple and therefore fast. For bintree cells that are completely inside or completely outside, no further testing is needed. The cells that are partially inside require their descendants to be recursively tested. This test is expected to be conclusive for most situations, but there are exceptions. An example is if vertices are both in partly opposite subspaces (e.g. D and C). For these situations, a more conclusive second edge test is needed.

The second test explicitly classifies the edges of the bintree cell with respect to the pyramid. Because it is not necessary to know what exactly the shape of the intersecting volume is, no clipping in the true sense of the word is performed. The following test is performed for each pair of vertices defining an edge. For each vertex of a bintree cell, the distance to each of the pyramid planes is known. Using these distances, we can easily compute whether an edge intersects the pyramid or not, see figure 3. In this example, the distances between the vertices and the pyramid planes are given by a_1, b_1

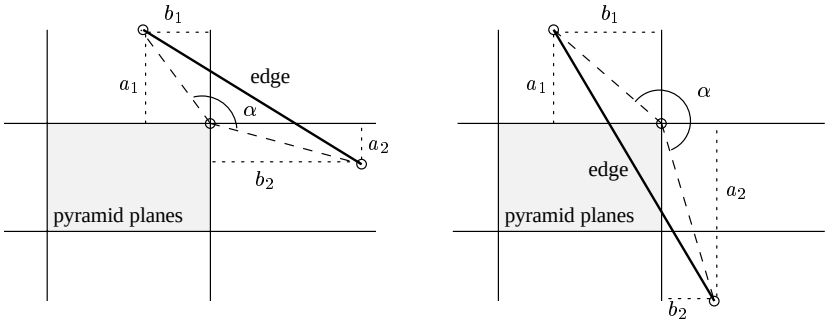

Fig. 3. Simplified clipping test.

and a_2 and b_2 respectively. An edge intersects the plane if and only if

$$\frac{a_1}{b_1} \leq \frac{a_2}{b_2} \tag{1}$$

A special situation occurs if the vertices of an edge are in diagonally opposite sectors, as is depicted in figure 4 (left). In this case, two evaluations of equation 1 are necessary. If the first of these tests indicates that the edge does not intersect the pyramid, the second test can be omitted, otherwise the second one is conclusive.

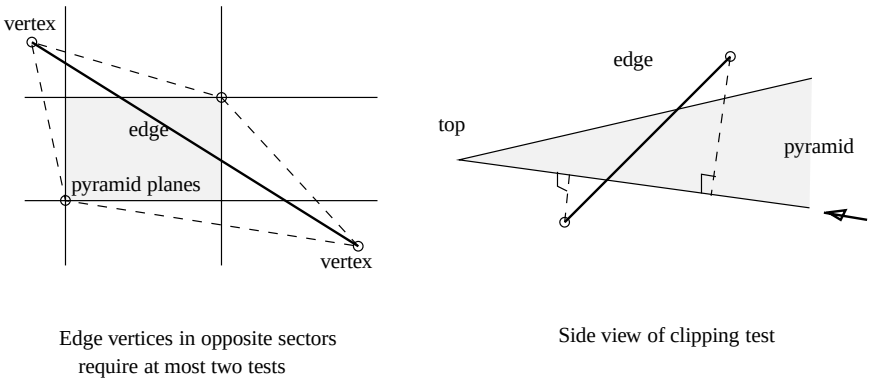

Fig. 4. Left: opposite sectors. Right: side view.

All edges are tested this way until an edge is found to intersect with the pyramid or until all twelve edges are tested. Then we can still have the case that no edge intersects the pyramid, but the cell completely surrounds the pyramid. In order to handle this case, for each edge a mask similar to the one in the first test, is updated. After each edge is tested, the orientation of the pyramid with respect to the cell is known and the intersection routine returns with the result.

In 3D, the distances that are used to classify an edge as inside or outside, are all with respect to the planes of the pyramid. Because for both vertices defining an edge, the distances to the same planes are used, the method is valid in 3D. Looking along the line indicated by the arrow in figure 4 (right), the image of figure 4 (left) would be seen.

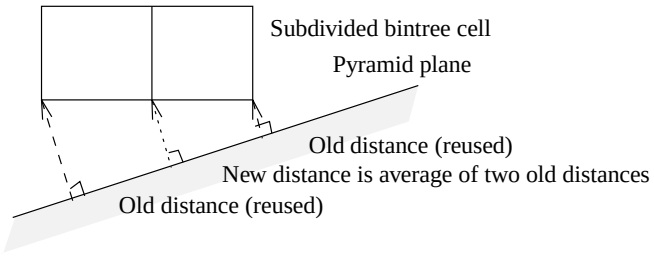

Fig. 5. Distances can be computed incrementally.

The distances are not computed for each vertex of each cell anew. Instead the regular structure of the bintree is used to efficiently calculate at each level of recursion the distances of the children nodes out of the distances of their parent node, see figure 5. Per subdivision only four new distances have to be computed, each by averaging two old distances. Eight old distances can be retained. Then, the pyramid - cell intersection routine can be called for the two new bintree cells.

The cell farthest from the view point is put on a stack and the cell nearest the view point is processed first. In this way, the first leaf cell reached is the one closed to the view point. All following leaf cells are automatically derived in the correct order.

The ordered cells are stored in a list, which is called the cliplist. Each pyramid thus generates its own cliplist during the traversal of the bintree that effectively contains all the cells that are traversed by at least one ray of the bundle and most likely by a majority of these rays (see figure 6). The six planes of each cell (x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}) are also copied to the cliplist. These are needed for the final ray traversal.

In addition, the objects in the cells may also be tested with respect to the pyramid planes and marked as in or out (excluding objects that will never be intersected by any of the rays).

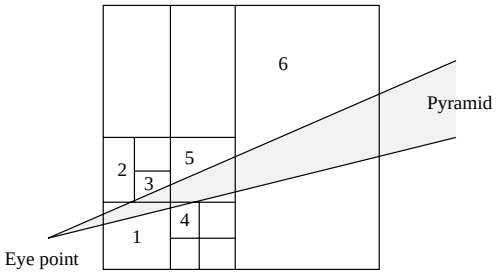

Fig. 6. Ray traversal generates cliplist (cells 1-6).

3 Ray traversal

After the cliplist has been generated, the first step of the pyraclip algorithm is finished. The next step is to determine which objects intersect with the rays within the pyramid. The ray traversal traverses the cells in list order until an intersection is found. While examining a new cell from the cliplist, the ray parameter value of the exit point for that cell is calculated. The ray exit point is the closest intersection with one of the (three) backfacing cell planes. Which of the six planes are the backfacing planes can be determined from the ray direction. The ray parameter of the exit point is then calculated with three multiplications, three adds and two compare. With an additional compare, it is determined whether the ray actually intersects the cell. If the new ray parameter is smaller than the parameter of the previous exit point, then the cell is not intersected by the ray and can be skipped. For instance in figure 7, the exit point for cell 1 is p_1 (minimum of p_1 and p_2). The exit point for cell 2 is p_2 (minimum of p_2 and p_4). For cell 3 exit point p_1 would be selected again, which is closer to the origin of the ray than the exit point of the previous cell. Therefore, it is concluded that cell 3 can be skipped, after which p_3 is found to be the exit point of cell 4.

What we have obtained with this algorithm, is a traversal for an adaptive spatial subdivision but with a ray traversal cost comparable with the standard grid method.

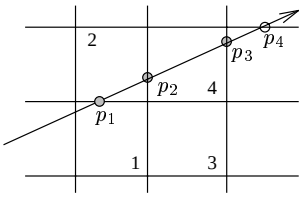

Fig. 7. Ray traversal.

4 Implementation and Evaluation

In order to have an idea how efficient the algorithm is, we have implemented it in Rayshade [9], a widely used public domain ray tracer. Earlier tests have shown that the grid traversal method in Rayshade is one of the fastest ray traversal methods [10, 11]. The grid traverseral in Rayshade is fast, because the standard DDA algorithm is enhanced with raybox testing [12] and ray hit caching. In our tests we have also included a comparison with the recursive bintree traversal algorithm [13, 14], implemented in Rayshade by de Leeuw [15].

For the test we have only traced primary rays (no shadow and reflection rays). The screen is subdivided into a number of non-overlapping rectangular regions and for each region a pyramid is constructed with its top at the view point. The size (width) of the pyramids (number of rays) is an important parameter that will directly affect the performance. If there are too few rays in a pyramid, then the overhead induced by the clipping algorithm will be more than the gain in the reduction of ray object intersections. On the other hand, if the pyramids are too large, the number of cells tested as outside will reduce, resulting in more ray - object intersection tests per ray. Other parameters that will

affect performance are the number of objects (polygons/balls), the maximum number of objects allowed in a cell and the maximum depth of the bintree

For the test we have used two models, balls and teapot, from the standard procedural database [16]. The models are depicted in figure 8. We chose these models because they do not distribute all objects uniformily over object space, which would give an unrealistic test result compared to models normally used in ray tracing and radiosity algorithms. We used the models in different resolutions.

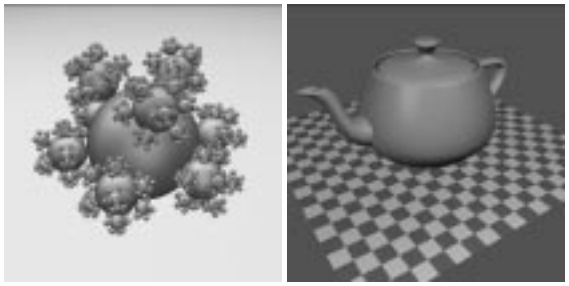

Fig. 8. Standard Procedural Models (balls and teapot)

First we verified the optimum setting of the subdivision parameters. For the grid method (uniform spatial subdivision) the rule of thumb given by de Leeuw [15] was confirmed. The number of voxels should equal the number of objects, or stated otherwise the number of voxels in each direction should be approximately equal to $\sqrt[3]{N}$, where N is the number of objects in the model. In fact, we found $2 + \sqrt[3]{N}$ to be the optimal setting for the two models.

The bintree algorithms were less sensitive to different parameter settings. For both model ranges, a maximum of 12 (or 16) objects per cell proved to be optimal. The maximum number of subdivisions varied from 18 for the small models to 24 for the large models (a maximum subdivision level of 24 corresponds with a maximum resolution of $2^8 = 256$ in both x, y and z directions). For other parameter settings, the times are only a few seconds off.

Then the optimal size for the pyramids was tested. The image size is 512 by 512 pixels. For each of the models, images were computed with $2^2, 4^2, 8^2, \ldots, 128^2$ pyramids per image. The optimum size lies around 32x32 pyramids (16^2 rays per pyramid; see figure 9 (right)). The pyra-clip time to generate the cell list is about 1 second for 32x32 pyramids and 2 seconds for 64x64 pyramids (see figure 9 (left)). The preprocessing time is linear in the number of objects, see figure 10. We choose therefore the 32x32 pyramid subdivision.

Rendering times for the 16^2 rays per pyramid pyraclipping, the bintree recursive and the uniform grid algorithm, are shown in figure 11. The tests were done on a SGI Power Onyx with 256 Mb internal memory. The memory available prohibited testing larger models. The pyraclip numbers give the pure traversal time, without the overhead of the clipping and without tracing secondary rays.

The figures show that the grid performs linear with respect to the number of objects, while the recursive bintree is logarithmic and wins for larger N. A multi grid method would probably have shown a similar performance. In fact this is the first time that we

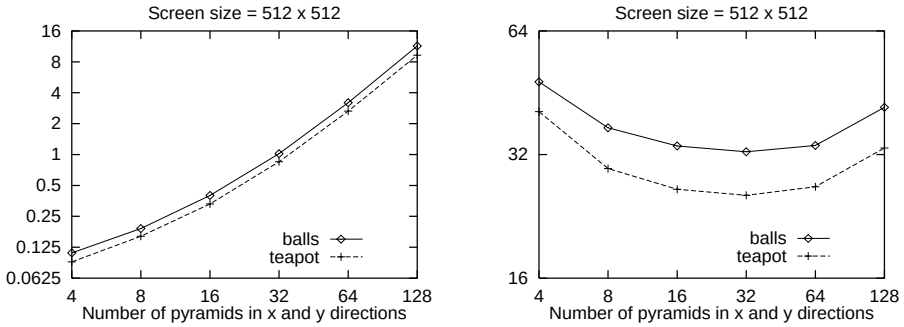

Fig. 9. Preprocessing time (left) and rendering time (right) as function of pyramid size (timing in seconds).

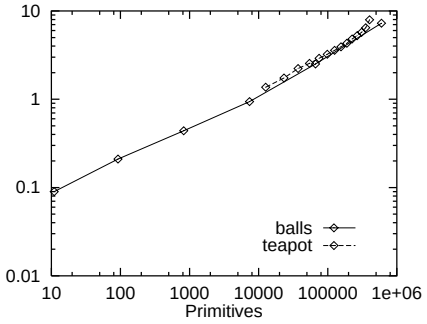

Fig. 10. Preprocessing time (in seconds) as function of the number of primitives.

are able to show that the performance of the recursive bintree traversal is $\log N$ for such large models. Unfortunately, the pyraclip traversal adheres more to the grid method than to the recursive bintree traversal and thus looses in the end compared to the recursive ray traversal. The figures show that there is no use for the pyraclip method for efficiency reasons only. All methods perform equally well (within the tested range). We found this also to be true for the other SPD models.

5 Conclusions and future work

The pyra-clip algorithm intersects a bundle of rays with a hierarchical spatial subdivision structure. First an ordered list of bintree cells is generated using an efficient pyramid - cell intersection method. Then the separate rays are traced through the list of cells.

The tests showed that the pyraclip method is not faster than the standard single ray traversal methods and thus for speed purposes only, the method as such does not give any additional benefit. However, the method is ideal in that it can generate with minor overhead coherent ray intersection tasks (a bundle of rays and matching objects) that can be executed as fast as the fastest ray traversal methods. We will therefore use this method to generate demand-driven tasks in a parallel ray tracing and radiosity algorithm [17].

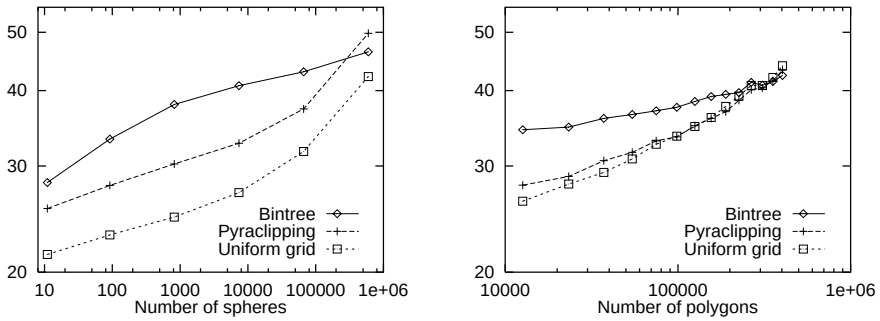

Fig. 11. Pyraclipping compared with uniform grid and bintree methods. Models used: balls (left) and teapot (right); timing in seconds.

Each processor in such a parallel ray tracer would be capable of both performing data parallel tasks and demand driven tasks. The data parallel tasks would provide a basic, though uneven load by tracing non-coherent secondary rays, such as reflection and refraction rays. Primary rays are then handled in demand driven manner and scheduled to processors with a low basic load. A processor receiving a bundle of rays and a cliplist will do the intersection calculations using this data. Shadow rays that sample area light sources are handled in a similar way. Such hybrid scheduling should both minimise data communication and balance the workload, yielding an efficient and scalable algorithm [18].

References

1. Green, S. A., Paddon, D. J.: 'Exploiting coherence for multiprocessor ray tracing', *IEEE Computer Graphics and Applications* pp. 12–27. (1989)
2. Shen, L. S., Deprettere, E., Dewilde, P.: A new space partition technique to support a highly pipelined parallel architecture for the radiosity method, *in* Advances in Graphics Hardware V, proceedings Fifth Eurographics Workshop on Hardware, Springer-Verlag. (1990)
3. Speer, L. R., DeRose, T. D., Barsky, B. A.: A theoretical and empirical analysis of coherent ray-tracing, *in* Computer-Generated Images, Springer-Verlag, Tokyo, pp. 11–25. Proceedings Graphics Interface '85. (1985)
4. Hanrahan, P.: Using caching and breadth first search to speed up ray tracing, *in* Proceedings Graphics Interface '86, Canadian Information Processing Society, Toronto, pp. 55–61. (1986)
5. Glassner, A. S., ed.: *An Introduction to Ray Tracing*, Academic Press, San Diego. (1989)
6. Haines, E. A., Wallace, J. R.: Shaft culling for efficient ray-traced radiosity, *in* Photorealistic Rendering in Computer Graphics, proceedings Second Eurographics Workshop on Rendering, Springer-Verlag 1994, pp. 122–138. (1991)
7. Greene, N.: Detecting intersection of a rectangular solid and a convex polyhedron, *in* P. Heckbert, ed., Graphics Gems IV, Academic Press, Boston, pp. 74–82. (1994)
8. van der Wal, P. O.: De coneclip versnellingsmethode voor raytracing, Master's thesis, Delft University of Technology. (1994)
9. Kolb, C. E.: Rayshade User's Guide and Reference Manual, included in the Rayshade distribution, which is available by ftp from princeton.edu:pub/Graphics/rayshade.4.0. (1992)

10. Jansen, F. W., de Leeuw, W. C.: Recursive ray traversal. In Ray Tracing News, vol 5, no 1. (1992)
11. Jansen, F. W.: Comparison of ray traversal methods. In Ray Tracing News, vol 7, no 2. (1994)
12. Snyder, J., Barr, A.: Ray tracing complex models containing surface tessellations, *Computer Graphics* **21**(4), 119–128. (1987)
13. Jansen, F. W.: Data structures for ray tracing, *in* L. R. A. Kessener, F. J. Peters, M. L. P. Lierop, eds, Data Structures for Raster Graphics, Springer-Verlag, Berlin, pp. 57–73. (1985)
14. Sung, K., Shirley, P.: *Ray tracing with the BSP Tree*, Graphics Gems III, Academic Press, Boston, chapter 6, pp. 271–274. (1992)
15. de Leeuw, W. C.: Recursieve ray traversal, Master's thesis, Delft University of Technology. (1992)
16. Haines, E. A.: Standard procedural database, v3.1, 3D/Eye. (1992)
17. Jansen, F. W., Chalmers, A.: Realism in real time?, *in* 4th EG Workshop on Rendering, pp. 1–20. (1993)
18. Reinhard, E., Jansen, F. W.: Hybrid scheduling for efficient ray tracing of complex images. (1995) Accepted for the International Workshop on High Performance Computing for Computer Graphics and Visualisation, Swansea, United Kingdom, july 3-4, 1995.