A vectorized traversal algorithm for Ray Tracing

José María Noguera
Dept. de Informática
Escuela Politécnica Superior
Universidad de Jaén
23071 Jaén
jnoguera@ujaen.es

Carlos Ureña
Dept. Lenguajes y
Sistemas Informáticos
ETS Ingeniería Informática
Univ. de Granada
18071 Granada
curena@ugr.es

Abstract

This article presents an optimized ray tracing algorithm which improves standard existing algorithms. In order to do this, this algorithm processes simultaneously a large set of rays and carries out a single traversal of the spatial indexing of the scene with all rays, which allows us to efficiently make use of hardware SIMD functionality and produces coherent memory accesses. Furthermore, during the single traversal, rays are grouped in such a way that these advantages are maintained even for non coherent ray sets.

The algorithm was observed to reduce the computation costs with respect to other standard solutions, especially for non coherent ray sets. It is worth noting that its characteristics make it especially suitable for graphics hardware (GPUs).

1 Introduction

Ray tracing is the dominating technique in the realistic or high quality image synthesis field, when the computation time is a secundary issue. The large processing times are due to the high computational costs of ray tracing. For each pixel in the image one or more rays must be sent through the scene and intersected with the objects it contains, the colour of the pixel must be calculated, and finally the pixels must be shown in the screen.

For applications that interact with the user in real time, algorithms which produce images very fast must be available. In these, other rendering techniques are usually used, such as triangle rasterization with z-buffer. Moreover, the last few years have seen a high increase in the development of optimized graphics hardware which can speed up triangle rasterization rendering time even more.

Nevertheless, ray tracing offers lots of advantages with respect to rasterization based algorithms, and it would be useful to have them in real time rendering:

- Image quality and correctness: The visual effects obtained with triangle rasterization methods are very limited. Many of the effects require careful manual design of the scene. The appearance of programmable graphics hardware (GPUs) has allowed large advances in this issue, but it is difficult to take advantage of these characteristics since they are difficult to implement and even then, they are just approximations of reality. This contrasts to the easy and natural way in which ray tracing allows the calculation of refractions, reflections, shadows, indirect illumination and other effects in a physically correct way.

- Complex scenes: For complex scenes, graphics hardware is limited since rasterization presents a linear cost in the num-

ber of scene primitives. Therefore sophisticated preprocessing techniques are required to reduce geometry. The problem is exacerbated if effects which require multiple rendering passes of the scene are used. In contrast, ray tracing presents logarithmic complexity in the number of scene primitives and includes *occlusion culling*. Only those calculations which are really needed for the final scene need to be done, since the *shaders* are only processed once visibility has been determined. Lastly, the nature of the ray tracing algorithm permits a trivial parallelization.

Ray tracing is not only useful in visualization. A multitude of algorithms, especially in the realistic image synthesis field, are based on ray tracing, such as *photon mapping* [15]. Furthermore, these algorithms usually work with non coherent rays. The availability of efficient ray tracing algorithms would allow a substantial decrease in the execution times of these algorithms.

Due to all these reasons, the development of new techniques which bring together ray tracing and real time applications is interesting.

It should be taken into account that using this algorithm for ray tracing or more generally for Global Illumination, implies that the application must generate large ray sets in order to calculate their intersections simultaneously. This means that the algorithm must be organized in a breadth first, rather than a depth first, scheme. For example, in ray tracing, all the primary rays must be generated first, then all the secundary rays, and so on.

This may mean an increase in memory use; however, we believe that the reduction in computation time makes this worthy. An additional benefit is that this scheme allows an efficient vectorized implementation of other parts of the algorithm, such as the evaluation of the global illumination model, or secundary ray generation.

This article is organized as follows: section 2 provides a general description of the design ideas of the algorithm. Section 3 describes in depth how the algorithm works. Lastly, section 4 offers an efficiency comparison to other methods.

1.1 Previous work

Even though ray tracing is a very old technique [1], its use for interactive applications is relatively new.

GPUs are optimized for triangle rasterization; however, different attempts to adapt their use for ray tracing acceleration [3, 8, 5, 4] have been made. Currently results obtained on GPUs are hardly better than conventional CPU results. This is due to the fact that even though GPUs are very powerful parallel architectures, they present difficulties for applications with a complex flow control. This may possibly change in the future due to the continuing development of GPUs. Therefore, it is still interesting to develop CPU ray tracing acceleration techniques which are suitable for GPU porting.

Wald [14, 13] presents a CPU based algorithm which exploits the existing coherence in primary rays. It is based in the idea that coherent rays will very probably make the same traversal through the hierarchical spatial indexing structure which contains the scene, and will intersect the same objects.

His proposal thus consists in traversing the spatial index and making the ray-triangle intersections with packets made of four coherent rays at the same time. This way, using SIMD structions of current processors, the computation time of the algorithm is reduced when doing the calculations for the four rays in parallel. In addition, Wald presents a careful design oriented towards minimizing code complexity and cache faults.

This idea is valid for primary rays since it is easy to locate rays which a priori have high probabilities of being coherent (Those crossing the same pixel or neighbour pixels in the image). But it fails for secundary rays where this supposition is not applicable. This article presents an extension to this idea, which permits the parallelization of calculations for different rays in any situation, even when these are not coherent.

2 Ray tracer implementation

The following sections describe our ray tracer implementation. Firstly a general description of the design ideas used is shown, and their application to the algorithm. The following sections describe the implementation details.

2.1 Motivation

Given a spatial indexing tree which contains the scene, the classic ray tracing algorithm travels recursively across the tree once for each ray. Wald, in contrast, proposes travelling across the tree once for each four ray packet. Our proposal furthers this idea, and proposes travelling across the tree once for all rays at the same time.

This approach provides the following advantages. As long as two or more rays which must traverse the same tree node exist, they will do so simultaneously, allowing the parallelization of the calculations by means of the hardware SIMD instructions. This translates to much higher probabilities of having some rays to process in parallel in each tree node, especially in the higher tree nodes. Even when there is no coherence among the rays, as the traversal of the structure advances they will be automatically classified so that the cited advantages remain.

Another advantage is that since the tree nodes are visisted only once, and not tens of thousands of times as is the case of previous algorithms, the number of accesses made to the spatial indexing structure of the scene is greatly reduced. A lower number of main memory accesses translates to a reduction in latency and cache faults.

A new data structure which contains the rays to be processed is needed. This structure does not mean more memory accesses with respect to classic ray tracing algorithms, since the same information which also must be stored and accessed in classic algorithms is simply reorganized in groups.

2.2 Parallelism due to SIMD

Current processors include SIMD extensions which allow computing many floating point operations (usually four) with one instruction, therefore increasing the performance of these calculations. The Intel SSE instruction set was used to exploit the data parallelism in packet ray traing because of its high availability.

3 Traversing the kd-tree

Our implementation uses a kd-tree in which the plane divides the voxel through the center and divides the voxel in two sub-voxels with the same size, for the spatial indexing of the scene. The algorithm can be easily changed to use division planes in other positions. The main reason of this choice instead of other structures such as Octrees or Bounding Volume Hierarchies (BVHs) is the simplicity of the traversal algorithm. For each node and ray only two decisions must be made: whether to follow or not each of the two child nodes. Additionally, this hierarchical structure is well adapted to scene complexity. And it is easy to parallelize using SSE instructions.

In order to parallelize the calculations, at each node the list of all the rays traversing it must be available. Firstly the workings of the algorithm will be shown for one ray, and then the algorithm will be expanded for the case of all rays at the same time.

3.1 Traversal algorithm for one ray

An adaptation to kd-trees (with dividing planes in the center of the voxels) of the incremental parametric traversing algorithm for octrees presented in [10] was used. Let $r = p+td$ be the equation of a ray expressed in parametric form, being p the origin vector, d the unit director vector and $t \geq 0$ the real parameter which determines a point along the ray.

For each node, t_{x0}, t_{x1} are the values of the ray parameter which intersect with the two planes perpendicular to the X axis delimiting the node voxel (similarly, we name t_{y0}, t_{y1} the corresponding values in Y, and t_{z0}, t_{z1} the values in Z). We assume $t_{x0} < t_{x1}$. These values

are calculated for the root node and from there on they are incrementally recalculated in each step of the tree traversal.

Let t_{xm} the parameter for which the ray crosses the plane (perpendicular to X) which divides the voxel in two equal subvoxels. It may be calculated as: $t_{xm} = (t_{x0} + t_{x1})/2$. For a voxel of the kd-tree divided by a plane perpendicular to X, the values of the intersections for the first sub-node visited are t_{x0}, t_{xm}, and for the second t_{xm}, t_{x1}. The values t_{ym}, t_{zm} are defined, calculated and used analogously for the voxels divided by planes perpendicular to Y or Z. In general, the calculation of the values of the child nodes of a node only require adding and dividing by two. It can be easily proved ([10]) that the ray intersects a node if the following holds:

$$max(t_{x0}, t_{y0}, t_{z0}) \; < \; min(t_{x1}, t_{y1}, t_{z1})$$

The algorithms works recursively. For each inner node, the ray parameters are updated for its first child, and intersection is determined. If an intersection exists, the node is processed. Later, the process is repeated for the second child.

If the tree is traversed selecting first the first node in the ray direction, once an intersection in the visited node is found the recursivity can be ended, since any other intersection which might be found will always be farther than the one just found.

3.2 Traversal algorithm for multiple rays

The kd-tree traversal algorithm for multiple rays works similarly to the previous one. The idea is processing at the same time all available rays, making only one traversal of the spatial indexing for all rays.

The pseudocode can be seen in Algorithm 1. Each node gets a list of intersecting rays, with its parameters adjusted for that node. If the list size is zero, we return. If the node is a leaf, the function TrgIntersec calculates if the rays intersect the triangles indexed in the node.

Otherwise, we are in an inner node. The fuction calcNode uses SSE to update the parameters of all the rays for the first child and to determine which rays intersect it. All rays intersecting the child are copied to a new list. Therefore this new list will contain between zero and the number of rays which intersect the parent node. Then, recursively, the first child is traversed with the new list of rays just calculated. Once the traversal of this child is finished, the second child is traversed analogously.

```
Traverse ( node, RayList )
   if nRays = 0
      return;
   if isLeaf(node)
      TrgIntersec( node, RayList );
   else foreach child h_i of node
      RayListC_i=calcNode(h_i,RayList);
      Traverse( h_i, RayListC_i );
```

Algorithm 1: Traversal for multiple rays

These lists do not include those rays for which an intersection was already found, nor those rays farther than the maximum allowed ray parameter, avoiding innecesary calculations.

The total number of operations required to determine if four rays intersect a node are: one addition, one multiplication, one maximum, one minimum and a comparison between these last, all SSE.

When working with multiple rays simultaneuosly it may happen that different rays require the spatial indexing tree in different order. The order in which a ray must visit the nodes in each step of the algorithm depends on the signs of the three components of the director vector of the ray. Therefore, this might happen if rays with director vectors of different signs exist, which is very probable. In these cases, it is not possible to traverse the tree only once for all rays simultaneously without losing the desirable characteristic that once the first intersection is found, it is the nearest one.

The solution implemented to solve this problem consists in classifying in a first step all rays in eight groups, classifying them by the signs of the components of their director vectors. Now, since all the rays in a group will

traverse the indexing tree exactly in the same order, the traversal algorithm can be applied with no problems for each group.

With this idea, the tree must be traversed a maximum of eight times, much lower than the tens of thousands of times needed for conventional algorithms. In practice, for primary rays the traversal is done at most four times, as long as the normal vector to the vision plane is parallel to one of the edges.

3.3 Ray memory layout

The data memory layout chosen to store the rays must take into account the fact that the implementation uses SIMD instructions in order to maximize efficiency and time [6].

Two data structures are used to manage the rays. One of them is global, can be accessed from any node, and stores all rays and attributes; while the other belongs to each specific node and only stores some data needed for the traversal, plus pointers to the global structure.

It is necessary to store globally for each ray:

- Its director vector and origin vector.
- The maximum allowed value of the ray parameter.
- A flag indicating whether an intersection with a triangle was found.
- The identifier of the nearest triangle intersected.
- The value of the parameter of the ray at the intersection.
- The parametric coordinates of the triangle at the intersection point.

As was stated previously in the algorithm description, during tree traversal each node receives a list of intersecting rays. This list is a local data structure for each node, and stores for each ray the entry and exit values of the parameters of the ray for the voxel $(t_{0x}, t_{0y}, t_{0z}, t_{1x}, t_{1y}, t_{1z})$, since they depend on the node we are currently in. A pointer to the

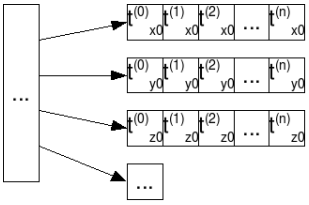

Figure 1: structure of vectors

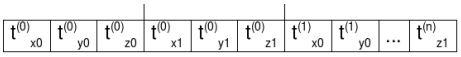

Figure 2: vector of structures

global structure which stores the rest of the ray information is also stored per ray.

This data will be processed by SIMD operations; therefore for maximum performance they should be stored with a specific organization called 'structure of vectors' which can be seen in Figure 1. This layout stores a vector with the t_{0x} of all rays, another vector with t_{0y}, and so on. This structure is less natural than the classic 'vector of structures' shown in Figure 2, but it is needed in order to obtain maximum efficiency with SSE.

In these figures, $t_{xi}^{(j)}$ represents the value of the x component of the t parameter for the j^{th} ray. i is 0 for the entry parameters and 1 for the exit parameters. The same applies to components y and z.

A problem which appears is that in order to recover the different parameters of a ray, distant positions in memory must be accessed, especially if the ray list is large. This induces bad results in cache memory. To solve the problem, we use a mixed memory layout organization based on the two previously mentioned alternatives, which can be seen in Figure 3. Rays are stored in four-packs. Each pack is a structure which contains a vector for the x components of the entry parameters of the four rays, another for the y components, and so on.

If the number of rays is not a multiple of four, the last packet will be incomplete and

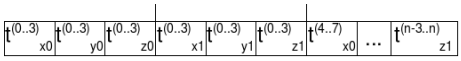

Figure 3: Mixed

will be filled with bogus data which will not be taken into account in the final results.

Each node gets a ray list, and must generate a new list for each child. If every visited node reserved memory dinamically for the child lists, the process would be inefficient. To avoid this, a ray stack is used. On tree construction, memory is allocated for a large enough stack. For each traversal, that scack is initialized with all the rays to be intersected and sent to the root node. The ray lists for the following nodes visited will be pushed and poped into the stack as needed, avoiding the need to allocate memory in each step.

Each node receives on the top of the stack the rays which intersect it and which should be processed. It calculates which of these intersect its first child, and pushes them into the stack. The child gets these new rays. Upon finishing processing the first child node, the rays sent to the child are removed from the stack, and the same process is done for the second child. This procedure ensures that only the rays which intersect one of the children need to be stored in the stack.

3.4 Ray-triangle computation

Two variants have been used for ray-triangle intersections:

- The standard Moller-Trumbore algorithm [7], which tests intersection of one ray with one triangle.
- A variation on the previous one, optimized, with normals and various intermediate results precalculated, with the data of four triangles packed, and making at the same time the intersection of one ray with four triangles with SSE.

The results are better (both in time and memory use) with the second option, (which requires lower depth trees), and therefore this option was used in tests.

Figure 4: *Stanford Bunny*

4 Performance

To study the efficiency of this algorithm, three different scenes were used: *Stanford Bunny* (69451 triangles), *Dragon* (0,8 million triangles) and *Happy Buddha* (1,1 million triangles). The scenes used were the same for all algorithms.

To compare the algorithm proposed, Wald's algorithm ([14, 13]) was simulated using our own algorithm but limiting the maximum number of rays which can be traversed together to four. This way, the spatial indexing must be traversed repeated times with four ray packets. The algorithm proposed is also compared to a classic recursive algorithm which traverses the rays through the structure one at a time.

The results were obtained for a maximum allowed depth of the spatial indexing tree of 16, 18, 20, 22 and 24, and the best results were selected for each algorithm and scene. For most of the cases, depth 20 was shown to be optimal. All measurements were taken on a 1.8 GHz Pentium-M with 512 MB of RAM.

4.1 Coherent ray comparison

For each case, a 100 image animation of the camera rotating around the scene was generated, and the mean results were taken. Since the rays have a common origin and cross consecutive pixels in the image plane, they present high coherence.

Figure 5: *Dragon*

Figure 6: *Happy Buddha*

Table 1 shows the average results obtained in MegaRays per second for the visualization of the different scenes for the three different algorithms. Figure 7 expresses this data graphically.

Similarly, Table 2 shows the average frames per second results obtained for the visualization of the different scenes for the three different algorithms. Figure 8 expresses this data graphically.

	n-Rays	4-Rays	1-Ray
Bunny	0,72	0,58	0,24
Dragon	0,51	0,42	0,19
Buddha	0,56	0,45	0,23

Table 1: Average performance in MegaRays per second of the three different algorithms when rendering the scenes at a resolution of 256^2 pixels.

It can be seen that the classic algorithm is less efficient in all situations. The other two algorithms are clearly superior since they are

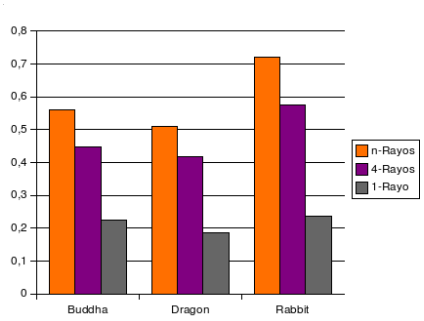

Figure 7: Average performance in MegaRays per second when rendering the scenes at a resolution of 256^2 pixels.

	n-Rayos	4-Rayos	1-Rayo
Bunny	8,31	7,06	3,31
Dragon	6,22	5,34	2,63
Buddha	6,9	5,78	3,17

Table 2: Images per second performance of the three different algorithms at a resolution of 256^2 pixels for the visualizaton of the different scenes.

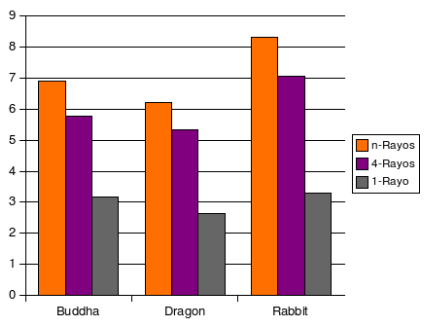

Figure 8: Images per second performance at a resolution of 256^2 pixels for the visualization of the scenes.

able to parallelize calculations. Between these two algorithms, our proposal obtains results slightly better than the four ray packet version. In particular, for the *Stanford Bunny* scene we obtain a speedup near 3 with respect

to the classic recursive algorithm, with 4 being the maximum theoretical speedup obtainable by perfect parallelization of all calculations.

This small improvement of our algorithm is due to the fact that the version with four ray packets, since the rays are coherent, manages to parallelize almost always their computation. However, this is not guaranteed to happen and sometimes the algorithm is forced to break the packet, reducing the degree of parallelism. Our algorithm manages to extract for each node all the data volume to be processed, so the probabilities of having packages with less than four rays are smaller.

4.2 Comparison for non coherent rays

It was mentioned previously in this article that the main advantage of traversing the tree with all rays at the same time appears when non coherent rays are used. These would be automatically classified upon descent through the indexing tree and the probabilities of having various rays in the same node (and therefore parallelize their calculations) would be much higher, especially in nodes near the root.

To demonstrate this a uniform distribution has been generated in order to send random rays with uniform density. For each ray to be generated, two points are taken with uniform probability in the surface of the bounding sphere of the scene [11, 12]. One of the points is used as origin of the ray and the other is used to generate a director vector.

Table 3 shows the average results obtained in MegaRays per second for the computation of non coherent rays which traverse the different scenes with the three different algorithms. Figure 9 expresses this data graphically.

	n-Rays	4-Rays	1-Ray
Bunny	0,39	0,18	0,13
Dragon	0,28	0,15	0,11
Buddha	0,27	0,15	0,11

Table 3: Average performance in MegaRays per second for the three different algorithms. Rays are not coherent.

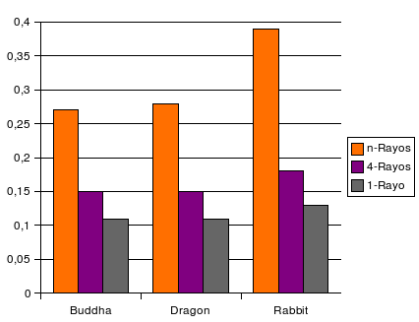

Figure 9: Average performance in MegaRays per second for non coherent rays.

It can be clearly observed that the results obtained with the four ray packet version have decreased notably with respect to the results with coherent rays. The results are very similar to those of the classic algorithm. The reason for this degradation in performance is that when taking packets of four non coherent rays, the probabilities that the packet breaks in the high levels of the tree are very high, and the algorithm degenerates into the conventional non parallel algorithm.

Our algorithm, in contrast, continues to offer a very high speedup with respect to the classic version of the algorithm. The rays are classified upon descense of the tree, and the probabilities of parallelization of computations are much higher because of the higher number of rays present.

The results obtained with coherent rays were similar between our version and Wald's, but with non coherent rays the results are notably distant. This demonstrates the higher ability of our algorithm to find parallelism in the computation of the rays in all kinds of situations.

5 Conclusions and future work

Until now, acceleration techniques for ray-tracing were useful only when it was easy to find coherent rays to process in parallel, such as primary rays. This article introduces a ray

tracing algorithm which instead of traversing the spatial indexing of the scene once per ray or per ray packet, it traverses the tree only once with all rays simultaneously (or a maximum of eight times if the rays have director vectors of different sign).

This technique has been shown to present the following advantages:

- Mainly, it can extract a higher level of parallelism in ray processing. This happens even in the case of non coherent rays, since these will be classified during tree traversal, and as long as two or more rays visit the same node, this is detected and the computation is parallelized. This is especially evident in the higher levels of the tree, which will be visited by a high number of rays.

- Furthermore, a node will be visited at most eight times, and not tens of thousands as is the case in the rest of the algorithms. This minimizes memory accesses to load the tree nodes and the triangles they contain.

As a future work, the ray stack scheme is not efficient when the number of rays arriving at a node is equal or smaller than four. This may happen in nodes near the leaves. An algorithm which combines the use of the ray stack when the number of rays is higher than four, and which does not use it when the number of rays is lower could solve this problem.

On the other hand, this algorithm tries to increase vectorization in ray processing. For each node in the spatial indexing all the data volume to be processed is extrated, but the operations must be carried out four at a time which is the most SSE can vectorize. Architectures which can vectorize a higher number of operations are hoped to yield even higher speedup ([2]).

GPUs are processors specialized in this sort of vector operations, so this algorithm is very well suited for GPU implementation, in order to make use of their higher parallel processing capability.

Acknoledgements

This article has been partially financed by project TIN2004-07672-C03-02 of the Spanish Ministry of Education and Science, and by project P06-TIC-01403 of the Junta de Andalucía and the European Union (FEDER).

References

[1] A. Appel. Some Techniques for Shading Machine Renderings of Solids. SJCC, pages 27-45, 1968.

[2] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray Tracing on the CELL processor. Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006.

[3] N. Carr, J. Hall and J. Hart. The ray engine. In Proceedings of the ACM Siggraph/Eurographics conference on Graphics hardware, 37-46. Eurographics Association, 2002. ISBN: 1-58113-580-7.

[4] M .Christen: Ray Tracing on GPU. Diploma thesis, University of Applied Sciences Basel, Switzerland, 2005.

[5] T. Foley and J. Sugerman. Kd-tree Acceleration Structures for a GPU Raytracer. In Proc. Graphics Hardware, 15-22, 2005.

[6] A. Klimovitski. Using SSE and SSE2: Misconceptions and Reality. Intel DeveloperUPDATEMagazine, 2001. http://www.intel.com/technology/ magazine/computing/sw03011.pdf.

[7] T. Möller and B. Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. Journal of Graphics Tools, 2(1):21-28, 1997.

[8] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. ACM Transactions on Graphics, 21(3):703-712, 2002. (Proceedings of SIGGRAPH 2002).

[9] T. Purcell. Ray Tracing on a Stream Processor. PhD Thesis, 2004 UMI Order Number: AAI3128683.

[10] J. Revelles, C. Ureña and M. Lastra. An Efficient Parametric Algorithm for Octree Traversal. Proc. WSCG, pp. 212-219, 2000.

[11] L. A. Santaló. Integral Geometry and Geometric Probability. Addison-Wesley, Reading (MA), USA, 1976.

[12] M. Sbert. An integral geometry based method for fast formfactor computation. Computer Graphics Forum (Proceedings of Eurographics'93), 12(3):409-420, 1993.

[13] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. Computer Graphics Forum, 20(3):153-164, 2001. (Proceedings of Eurographics).

[14] I. Wald. Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Saarland University, 2004.

[15] Henrik Wann Jensen. Global Illumination using Photon Maps. In Rendering Techniques '96. Eds. X. Pueyo and P. Schröder. Springer-Verlag, pp. 21-30, 1996