# Efficient stackless ray traversal for bounding sphere hierarchies with CUDA

Tomasz Toczek, Dominique Houzet, Stéphane Mancini

# Efficient Stackless Ray Traversal for Bounding Sphere Hierarchies with CUDA

Tomasz Toczek[a*], Dominique Houzet[a†], Stéphane Mancini[a‡]

[a]GIPSA-lab, INPG-CNRS,
961 rue de la Houille Blanche Domaine universitaire – B.P. 46,
38402, Saint Martin d'Heres, France

One of the challenges of GPU-based ray tracing is the hierarchical space indexation structure traversal. Stackless approaches to this problem offer benefits both in terms of maximum representable scene size and performances. [5, 9]

We improve on the current stackless traversal methods by increasing memory access locality and accelerating both the primary and secondary ray shooting through specific optimizations achieved at the cost of additional precomputations. Those supplementary steps can be performed on the host and have their results transferred asynchronously on the GPU while the rendering kernel is running, hence being completely hidden.

## 1. Introduction

Ray tracing as a rendering method is well known for the quality of the produced output and its adequacy for solving difficult problems such as light transport. This is done by casting numerous rays, the more the better for most rendering algorithms. That is why acceleration structures (AS) are systematically used to speed up the ray traversal itself. Since the time spent building the AS is typically far lesser that the one gained by shooting rays through it, this approach makes sense. For increased scalability, the AS tend to be hierarchical space indexing means build using divide and conquer strategies [4]. Therefore, the most straightforward ways to traverse such AS involve using non-tail recursion, and hence a stack. This hinders their adaptability to GPGPUs, which permit to allocate only very limited per-thread amounts of fast on-chip memories. What is more, nVidia architectures' register memory is not addressable, implying the stack a thread would need has to be located in the scarce shared memory. This is a significant problem to overcome in order to make the oftentimes otherwise embarrassingly parallel ray tracing algorithms GPU friendly.

Several approaches have been proposed so far. Of course, one way to avoid stack problems is to use a flat AS instead of a hierarchical one [6, 7]. Such methods suffer however from the well-known [4] drawbacks of non-hierarchical AS, the main one being their inadequacy for handling scenes with spatially variable detail level. When it comes to

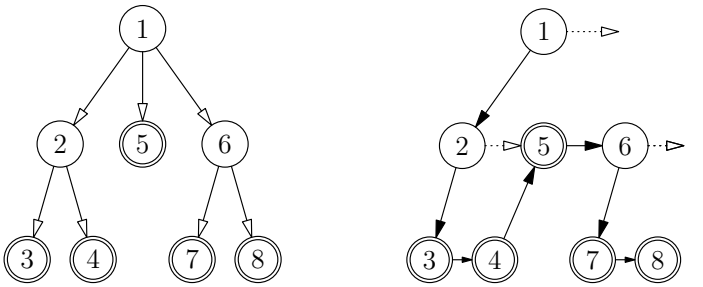
*e-mail: tomasz.toczek@gipsa-lab.inpg.fr
†e-mail: dominique.houzet@inpg.fr
‡e-mail: stephane.mancini@grenoble-inp.fr

hierarchical AS traversal on the GPU, essentially two families of methods can be found: those using kd-trees [2, 5], that is a hierarchical subdivision of space by splitting planes, and those using bounding volume hierarchies (BVH) [1, 3, 9]. In order to perform the traversal without a stack, one can restart from the root of the tree at each iteration [2], which is costly in redundant computation, or to some extend try to store additional cues within the nodes to compensate for the missing information [5], which is costly in memory space and, even worse, bandwidth used during rendering.

Our approach in inspired from [9] and improves on it on several accounts. We propose to perform supplementary precomputations on the host and transfer their results on the GPU while it is busy ray tracing. While the amount of computations on the host side and the size of the AS are higher, the ray traversal is faster and the kernel memory reference locality is kept reasonably high. This allows to make better use of the texture cache. The sections 2 and 3 contain a detailed description of our method. Ray shooting performance measurements are given in section 4. Finally, practical issues linked to the AS updating and transfer between the host and the device are discussed in section 5.

## 2. Overall principle

### 2.1. Traversal




(a) Original BVH (b) Resulting traversal list

Figure 1. Building a traversal list from a BVH is essentially equivalent to recording the steps of a depth first traversal. On the fig. 1(b), dotted lines represent escape links. They are not shown for the primitives (the leafs), since they always lead to the next element (already pointed by the array). Following escape links leading to nothing ends the traversal.

The basic principle of traversal is much the same as exposed by Thrane and Simonsen [9], and we will use from now on the terminology coined by them. We build a BVH tree (using spherical nodes instead of boxes in our very case), and produce from it a series of lists which will be iterated upon during traversal. How such a list is obtained from the BVH is represented on the figure 1; the list is stored as an array obeying the following rules:

- each element of the list corresponds to a node of the BVH
- for each element corresponding to an internal node: the next element in the array corresponds to its first child

- for each element corresponding to a leaf: the next element corresponds to its immediate sibling in the tree, or, if it has none, to the one of its first ancestor having one (let us call it a super-sibling)

In other words, the elements are in the order given by a depth-first traversal of the BVH tree. In order to be able to skip whole nodes should a ray not intersect them, a relative escape index is stored for each element, pointing to the element's super-sibling (that is, where the depth-first traversal would continue should the current node have no children). Clearly, this index is 1 if and only if the corresponding node is a leaf, hence enabling us to differentiate bounding spheres from primitives from its value alone. Two special values of the relative escape index, 0 and -1, identify respectively the last primitive, and an internal node with no super-sibling. When such a special escape link is followed, traversal ends (intersection may or may not have been found). In practice, to represent a node, we use an "escape pointer" of 32 bits and a word of 128 bits representing either a bounding sphere geometry or a list of triangle vertex indices. The vertex positions themselves are stored in a separate array shared by all the traversal lists, making each of them comparatively smaller.

The resulting traversal algorithm is quasi-identical to the one given in [9]:

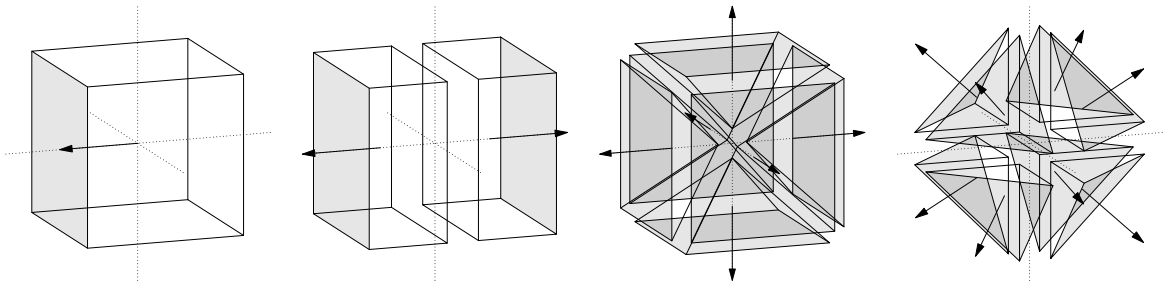
```
node <- traversalListHead(dir)
param <- infinity
intersectedTriangle <- none
do
    relativeNext = node.relativeNext
    if node.isTriangle then
        if ( node.isIntersectedByTheRay
          && intersectionParameter < param) then
            param <- intersectionParameter
            intersectedTriangle <- node
    else
        if ( node.isIntersectedByTheRay
          && intersectionParameter < param ) then
            relativeNext <- 1
    node <- node + relativeNext
while (relativeNext > 0)
return (param, intersectedTriangle)
```

Thrane et al. chose to generate only one such list, and regret the ray traversal is costly in the worst cases. We attempt to work around this by making several lists, by ordering the nodes of the BVH tree along different spatial axes. For each ray, we determine before traversal which list to use by picking the one which matches its direction the best.

There are several ways to pick the node sorting directions, which are illustrated by the figure 2. This gives the following options:

- A single list is constructed (figure 2(a)), the nodes being ordered along an arbitrary direction. This case has been studied by [9].

- Two lists are constructed (figure 2(b)), the nodes in one being stored in the reverse order of those of the other. The framerate is less dependant on the point of view than on the previous case. We did not study this case because it does not show some of the interesting properties of the following ones.



(a) 1 main direction (b) 2 main directions (c) 6 main directions (d) 8 main directions

Figure 2. Several interesting partitions of the directions of space

- Six lists (figure 2(c)) constructed by ordering the nodes along the three main axes, in both ways. The overhead due to the redundant geometry storage is becoming rather high, but it can be noticed that this configuration allows for geometry culling. Indeed, assuming the scene geometry is given as a set of one-sided triangles, if a ray direction belongs to a given sextant, it is impossible for it to hit a triangle whose normal belongs to the opposite sextant. Therefore, node redundancy is more like 5 than 6; for each sextant list, approximately 16.67% of the geometry is culled. It is the case we have studied, as a good trade off.

- Eight lists (figure 2(d)). Geometry culling is still possible, but only to an average of 12.25% of primitives par octant list. On the other hand, the optimizations presented in the section 3.1 work better for this division method than for the previous.

The ray shooting process tends to benefit from the fact that the nodes a ray may intersect are stored in compact lists with only one pointer per node, and with a part of the scene geometry already discarded. This increases the locality of the traversal process. What is more, due to the limited size of the texture memory caches on current GPUs, expecting a given geometry node to remain in the texture cache between two consecutive ray launches is rather unrealistic. That is why creating multiple traversal lists is expected to increase memory reference locality despite also increasing significantly the overall memory consumption.

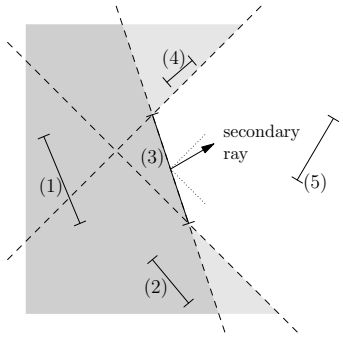
### 2.2. Bounding Sphere Hierarchy construction

For our test we used a top-down BVH generation algorithm very similar to sparse octree generation [11]. In order to build a node from a set of triangles:

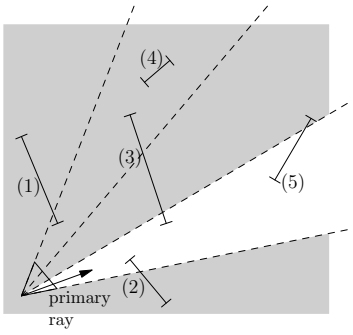
- we determine the tightest axis-aligned bounding box containing all the triangles; let $C = (C_x, C_y, C_z)$, the barycenter of the bounding box

- the shape of the bounding volume is that of the smallest sphere containing all the triangles and having $C$ as its center

- each triangle big enough compared to the node becomes one of its children




(a) Per-polygon head computation; in this example, traversal should start from (4) or, even better, (5)




(b) Approximate frustum culling; in this example, traversal should start from (2)

Figure 3. Partially computed traversal

- the other triangles are split in eight groups, depending on the sign of $G_i - C_i$, for each $i \in \{x, y, z\}$, $G$ being the barycenter of a given triangle; the process is then applied recursively on each of the non-empty groups, which are then added as children to the current node

Implemented correctly, this algorithm can be quite fast and suited for dynamic scenes with a reasonable poly count. Of course, any other algorithm could have been used for the initial BVH generation.

## 3. Partially precomputed traversal

It is interesting to notice that rays sharing some common geometric features may not need to start from the first element of the traversal list for their direction. For instance, primary rays of some part of the screen may all miss all of the primitives. Similarly, secondary rays originating at a given triangle may also systematically miss a given number of consecutive intersection tests, because of the very geometry of the triangle. In this section, we comment on how to handle such cases.

The figure 3 provides 2D illustrations of the methods exposed below, considering a list of 5 primitives roughly ordered along the x axis, numbered from (1) to (5).

### 3.1. Secondary rays

When aiming for interesting shading effects, most of the rays traced are secondary rays. It is therefore important to speed them up to the maximum extent. Sadly, secondary rays are much more tricky to accelerate than primary rays using our approach. However, it is to some degree feasible.

To do so, the heads of the traversal lists can be specifically computed on a per-triangle basis. When a ray bounces on a triangle, those heads are read with the rest of the material information. The algorithm to compute the skipped elements is very similar to ray shooting and hence can be performed on the GPU if necessary. It is demonstrated on a 2D example on the figure 3(a).

The basic idea is to skip all the list elements which are behind the polygon from which a ray is re-emitted (dark gray area on the figure 3(a)). Additionally, for slightly improved efficiency, should also be excluded all the nodes which cannot be hit because of the position of the polygon and the overall direction of the ray (light gray area on the figure 3(a)). How this can be done depends essentially on what direction space partition is used in order to obtain the traversal lists. In all the examples of partitions of the figure 2, skipping the nodes behind a few more planes, depending on which overall direction the ray is going in, does the job. The 2D case is illustrated on figure 3(a), where 2 such additional planes can be seen. In 3D, the number of additional planes is 4 in our case, since we have divided the direction space into sextants (see figure 2(c)).

Theoretically, we should check that any rejected node does not intersect the intersection of all the half-spaces delimited by the planes described in the previous paragraph and oriented towards where re-emitted rays can exist (for a given polygon and a given overall direction). In practice, this is costly and not really necessary. A mostly efficient and inexpensive test is to simply check whether there exists a half-space among the aforementioned ones which a node does not intersect at all. If there is one, the node can safely be skipped.

### 3.2. Primary rays

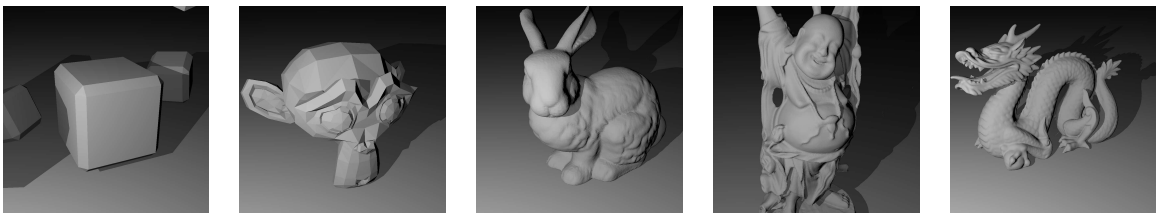
A similar technique can be applied to primary rays, and works significantly better. That being said, advanced rendering algorithms cast only so much primary rays, and hybrid ray tracing is another way to get a significant speed up in most cases (see for instance [8]). However, casting primary rays can be a good way to produce depth of field and fish eye effects.

The basic idea is to divide the screen into a grid, and to compute the heads of the traversal lists for the primary rays of each cell. When using a pinhole camera, a satisfying result can be obtained by using for each cell a set of separating planes as illustrated on the figure 3(b). The resulting effect is superficially similar to frustum culling. Of course, due to the nature of traversal lists, a node which should be culled will not be if it is behind one which cannot. However, this tends to work much better than what is done for secondary rays because the "beam" which is tested for intersections is much narrower.

## 4. Ray shooting performances

We measured the ray shooting performances on the 5 scenes of the figure 4. The benchmarking results are summarized on the table 2. The performances given are those obtained with all of the aforementioned optimization enabled. Alternatively, the table 1 shows how performances vary depending on which optimizations have been turned on. The scene used to perform those measurements is the Stanford bunny. Here is a summary of the meaning of the columns:

- Node ordering: the node ordering policy for the traversal lists. Since spheres and triangles are convex objects, their projections onto the sorting axis are segments. Let us assume the sorting axis is parametrized, and the projection of the nodes is given by the parameter ranges $([x_{min}^{(i)}, x_{max}^{(i)}])_{i\in[1..n]}$. The policy *earliest first* puts the nodes in ascending $x_{min}$ order, whereas *latest last* uses the ascending $x_{max}$ order



(a) Beveled cubes (b) Suzanne (c) Stanford bunny (d) Happy Buddha (e) Dragon

Figure 4. Our not-so-original test scenes. On a GTX 280 card, at a resolution of $1024 \times 1024$ and with a single point light, we got on average 31.36 fps for the cubes, 22.87 fps for Suzanne, 6.01 fps for the bunny, 3.03 fps for the Buddha, 3.20 fps for the dragon.

| Node order | Back-face culling | Per-polygon head | Frustum culling | FPS | # rays per sec | # traversed nodes per sec. |
|---|---|---|---|---|---|---|
| latest last | disabled | disabled | enabled | 6.81 | 8.43 M | 15.02 G |
| latest last | disabled | disabled | disabled | 6.81 | 8.43 M | 14.54 G |
| latest last | disabled | regular | enabled | 6.78 | 8.40 M | 13.99 G |
| latest last | disabled | regular | disabled | 6.78 | 8.40 M | 16.41 G |
| latest last | enabled | disabled | enabled | 6.81 | 8.43 M | 13.05 G |
| latest last | enabled | disabled | disabled | 6.78 | 8.40 M | 14.85 G |
| latest last | enabled | regular | enabled | 6.82 | 8.44 M | 15.01 G |
| latest last | enabled | regular | disabled | 6.81 | 8.43 M | 13.92 G |
| earliest first | disabled | disabled | enabled | 6.78 | 8.40 M | 16.29 G |
| earliest first | disabled | disabled | disabled | 6.81 | 8.43 M | 13.52 G |
| earliest first | disabled | regular | enabled | 6.79 | 8.41 M | 15.32 G |
| earliest first | disabled | regular | disabled | 6.81 | 8.43 M | 14.87 G |
| earliest first | enabled | disabled | enabled | 6.79 | 8.40 M | 13.75 G |
| earliest first | enabled | disabled | disabled | 6.78 | 8.40 M | 16.17 G |
| earliest first | enabled | regular | enabled | 6.81 | 8.43 M | 13.39 G |
| earliest first | enabled | regular | disabled | 6.81 | 8.43 M | 14.65 G |

Table 1
Efficiency of several optimization methods on the bunny (without the ground). As most of the proposed optimisations happen at the cost of memory access locality, framerate gains are offset by lesser memory efficiency. Our optimisations permit to reduce the average number of nodes traversed per ray, however.

| Scene | # triangles | # vertices | fps | # rays per sec | # traversed nodes per sec. |
|---|---|---|---|---|---|
| Cubes | 222 | 124 | 31.36 | 60.85 M | 70.52 G |
| Suzanne | 970 | 511 | 22.87 | 44.43 M | 48.20 G |
| Bunny | 69453 | 35951 | 6.01 | 11.52 M | 12.41 G |
| Happy Buddha | 293234 | 144651 | 3.03 | 5.69 M | 6.42 G |
| Dragon | 871416 | 437649 | 3.20 | 6.31 M | 6.96 G |

Table 2
Ray tracing performance summary for different scenes. As polygon count increases, the locality of memory references of the rays of a block is progressively lost, leading to a lesser node traversal rate.

- Back-face culling: enables geometry culling at list construction
- Per-polygon heads computation: computes a set of heads of the traversal lists for each polygon, as described in the section 3.1. It is either *disabled*, *simple* (effectively rejecting intersection tests with the primitives located under the surface on which the ray bounces) and *regular* (further rejection using the additional separating planes)
- Frustum culling: divides the screen space into a grid and computes a different head of traversal list for the primary rays of each cell, as described in section 3.2

Since we used a very simple example rendering algorithm (direct illumination with lambertian shading of triangles), all our test cases' performances are memory bound. Indeed, ray shooting performance is supposed to be approximately a logarithmic function of the triangle count. We used 1D texture caches to store the triangles and vertices, therefore memory reference locality concerns impact the framerate the most. This can be seen on the table 2, where small scenes fitting entirely or almost entirely in the cache offer 10 times higher framerates that their bigger counterparts. It is not unrealistic to think that a cleverer memory organisation scheme, or the use of more complex primitives or shading models – such as those used by real-world ray tracing based renderers – could bring the bottleneck back to the computations. Uncached coalesced references have also been shown as a possible remedy [10]. Such considerations are beyond the scope of this article, however.

The table 1 shows the impact of the optimisations described above, presented on the Stanford bunny scene, with the ground removed to maximize their impact. Since we are still memory bound, the framerate stays virtually the same. It can be seen that they have a heavy incidence on the average traversed nodes per ray count, impacting to as much as 19.8%. In fact, the effects of some parameters, such as the node ordering, are somewhat difficult to assess and call for further investigation in relation with the BVH construction method.

A fair comparison with other methodologies is not straight-forward, since we focused on the traversal itself, not implementing several important rendering related optimizations such as ray shooting by packets [5] or efficient load balancing [10]. Future prospects include those aspects. We have, however, compared our 6-traversal-list based implementation with

| Scene | Scene copy | Frame rendering | Ratio |
|---|---|---|---|
| Cubes | 12.6 ms | 31.8 ms | 39.5 % |
| Suzanne | 12.7 ms | 43.7 ms | 29.1 % |
| Bunny | 18.2 ms | 167.0 ms | 10.9 % |
| Happy Buddha | 38.2 ms | 329.0 ms | 11.6 % |
| Dragon | 88.5 ms | 311.7 ms | 28.3 % |

Table 3
Comparison between memory transfers and computation times

the simpler 1-traversal-list case [9], improved with each of the optimizations mentionned earlier still working in that context. The gains of our approach range from 5% to 15% on our test scenes, the more complex a scene the higher the gain.

## 5. Acceleration structure handling improvements

One of the reasons for the efficiency of the described methodology is the fact the traversal lists are stored as ordered arrays. In this aspect, the traversal lists are similar to AS with implicit pointers. As a consequence, they suffer from the weak points of such structures, i.e. mainly their lack of updatability. However, as can be seen on figure 3, for reasonably sized scenes, transferring the geometry information takes less time than rendering. Since recent NVidia GPU allow for it, those transfers can be done in a non-blocking fashion for the host as well as for the device. Wisely used, the host to device bandwidth permits to modify parts of the traversal lists with little overhead.

It is also worth noting that mechanisms allowing to traverse several lists in a row may be efficiently used. Once a traversal list ends, the head of the next one becomes the current node. Since the parameter of the best intersection is not reseted, the traversal of subsequent lists is in principle faster than that of the original one. Several "layers" can be created in this way; for instance, a static mid-poly one and a dynamic low-poly layer recomputed at each frame.

Also, the classical argument according to which vertices of the primitives within a BVH can be updated without recomputing the whole structure still holds for traversal lists, as long as to host keeps enough information to locate the list elements corresponding to a given BVH tree node.

All in all, traversal lists are less rigid than they might first seem.

## 6. Conclusion

In this paper, we have shown a series of measures significantly accelerating ray shooting on the GPU at the cost of additional precomputations on the host, compared to classical stack-based algorithms. Raw ray shooting performances have been analysed and commented, and are to our knowledge among the best published so far at equal scene complexity. We did not provide accurate performance measurements for the parts precomputed on the host, mainly because our benches involved various static models bearing little resemblance with polygon distributions of typical dynamic scenes. We did, however,

give several ideas to reduce the host computations to the strict necessary. In particular, if a scene can be divided into a static and a dynamic part (as often found in, for instance, video games), the AS for the static part can be precomputed once and for all. Also, we have shown how well-designed DMA usage can reduce both the CPU and GPU loads.

Future work will include studying the compatibility of currently available ray tracing rendering algorithms' optimisations with our ray shooting method, and studying the engine design required to easily and efficiently handle real-time rendering of dynamic scenes with such algorithms.

## REFERENCES

1. Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of Graphics Interface 2006*, pages 203–209, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.
2. T. Foley and J. Sugerman. Kd-tree acceleration structures for a GPU raytracer. In *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWWS '05)*, pages 15–22, 2005.
3. Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Real-time ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, September 2007.
4. Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.
5. Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).
6. Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).
7. Kristóf Ralovich. Implementing and analyzing a GPU ray tracer. In *Central European Seminar on Computer Graphics (CESCG)*, April 2007.
8. Philippe C. D. Robert, Severin Schoepke, and Hanspeter Bieri. Hybrid ray tracing - ray tracing using GPU-accelerated image-space methods. In José Braz, Pere-Pau Vázquez, and João Madeiras Pereira, editors, *GRAPP (GM/R)*, pages 305–311. INSTICC - Institute for Systems and Technologies of Information, Control and Communication, 2007.
9. Niels Thrane and Lars Ole Simonsen. A comparison of acceleration structures for GPU assisted ray tracing. Master's thesis, University of Aarhus, August 2005.
10. Timo Aila and Samuli Laine. Understanding the Efficiency of Ray Traversal on GPUs. In Proceedings of High-Performance Graphics 2009
11. Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics & Applications*, 4(10):15–22, October 1984.