A GPU-Based Data Structure for a Parallel Ray Tracing Illumination Algorithm

Diego Cordeiro Barboza Esteban Walter Gonzalez Clua

Universidade Federal Fluminense, Instituto de Computação - Media Lab, Brasil

Abstract

Ray tracing is a largely employed technique for generating computer images with high fidelity and realism. However, this technique is very costly, mainly because of the intersection calculations made by the algorithm. Still, ray tracing is a highly parallelizable algorithm, since the calculations for a single light ray are independent from the others. This way, the implementation of the ray tracing on the GPU is a natural process. Data structures can be employed for reducing the processing load of the ray tracing algorithm, minimizing the number of intersection calculations. In recent works, these structures are being ported to the GPU, so they can be used to accelerate various parallel algorithms. In this work, we proposed a GPU ray tracing implementation using an octree as acceleration structure. Every single step of the algorithm runs in parallel on the GPU, so the communication bottleneck between the GPU and the CPU is eliminated. This work also proposes the study of which is the best way to represent the octree on the GPU, by comparing two different approaches.

Keywords: Data structures, Octree, Ray tracing, GPU

Authors' contact:
{dbarboza,esteban}@ic.uff.br

1. Introduction

Ray tracing is a well known illumination algorithm for generating synthetic images, due its high visual fidelity. Movies and non-interactive applications, such as medical visualization systems, may employ the ray tracing technique to generate high quality and complex images. However, this is a very costly recursive pixel per pixel illumination technique, which makes it hard to employ in real time applications, such as digital games.

The ray tracing algorithm simulates the physical light behavior to generate images. The typical usage is the inverse ray tracing, where rays are cast from the view point and each ray is followed through the scene towards a light source. It is necessary to calculate intersections of every ray with all the primitives that composes the scene in order to find out the color of each pixel, so the cost for high resolution images in complex scenes makes it impracticable for real time applications.

This algorithm is naturally parallelizable, since the computations for a single ray are independent from others. So, the modern graphics processors are employed to run the ray tracing in parallel with as much rays being processed at the same time as the number of available processors.

But the parallelization itself should be not enough. The individual processing of each ray is still an expansive task, especially because of the intersection tests. Spatial structures are be used to reduce these intersection tests. Instead of testing every single ray with every primitive in the scene, we first traverse the structure where every step reduces the remaining primitives. At the end, only a small amount of the original primitive list remains, so the ray-primitive intersection is done.

In this paper, we propose the usage of an octree – a data structure where each node has none or eight children nodes and the data is stored in the leaves – to partition the space and reduce the amount of intersection calculations done by the ray tracing algorithm. Since the ray tracing runs in parallel in the GPU, this octree needs to be modeled so it can be stored in GPU memory and accessed from there, without any data search in the main memory. This paper also presents a comparison of two approaches for modeling the octree, an indexed octree and a hash table based octree. We discuss the advantages and disadvantages of each one.

The section 2 presents some related work about this paper's subject. The section 3 presents some information regarding octrees and section 4 describes the ray tracing technique. Section 5 shows the data modeling developed for the parallel ray tracing on the GPU and section 6 describes the octree usage itself for the ray tracing. Finally, section 7 presents the conclusions and suggestions for future works.

2. Related Work

Ray tracing and acceleration structures are highly studied fields. [Purcell 2002] presents a GPU-based ray tracing implementation and compares it to the CPU one, taking in account points like data traffic and processing power of each device. [Wald and Slusallek 2001] presents the state-of-art of ray tracing, including its parallel implementation in computer clusters.

[Horn et al. 2007] and [Revelles et al. 2000] presents studies of acceleration structures for ray

tracing and efficient octree traversal. [Garanzha and Loop 2010] presents a ray tracing implementation using a breadth-first bounding volume hierarchy traversal.

A hash table based octree implementation for GPUs is presented by [Madeira 2010]. The author focuses in optimized search that can't be directly applied to the ray tracing algorithm, since hierarchy is very important in this case, so some adaptations have been done over his structure.

3. Octree

An octree is a hierarchical data structure for tridimensional space partition, where each element is represented by a cube and may have none or eight children in a level immediately bellow [Castro 2008]. The octree is built from its root, where a bounding box is created. The root is then divided into eight children and for each resultant node, we verify if it intersects the scene's primitives. If so, the node is internal and will be subdivided again until the max tree level is reached or the node contains desired primitive count.

There are some ways to represent an octree. The most usual are: the pointer representation, where each node has pointers to its children; the linear representation, where the tree is stored in an array and the access to the children is done by manipulating the parent's index (the child index is given by 8 * parent index + child index); and a hash table representation, where the children access is done by applying a hash function to the parent's key.

The pointer-based and the linear octree are pretty straightforward. The hash table representation is proposed for GPU usage by [Madeira 2010]. The tree is also stored in an array, but the index of each node is given by its key. In order to navigate through the octree, we need to use a hash function that will result in the child's key, based on its geometric position inside the octree.

The node's keys are coded with the Morton Code, as used by [Castro 2008] and [Madeira 2010]. The tree root has the key 1 and the key for each child is created by adding three bits correspondent to its geometric position. Figure 1 shows how it works for a quadtree, but the process is analogous for an octree.

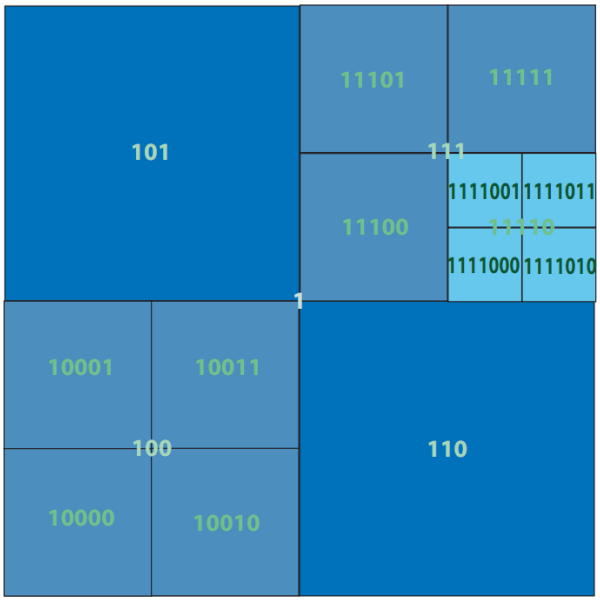

Figure 1: Morton code for each node in a quadtree [CASTRO 2008].

4. Ray tracing

Ray tracing was introduced in the late sixties by [Appel 1968]. It reduces the generation of computer images to finding intersections of rays casted from a view point and the employment of an illumination model at these points [Kuchkuda 1988].

The resulting color of each pixel in the visualization area is determined by the used illumination model, which is flexible enough for adding or removing components as it is needed. [Phong 1975] presents a very basic model that has been improved over the years with the addition of new components. [Hall and Greenberg 1983] and [Whitted 1980] presents models that takes in account various components, such as diffuse, specular, diffraction, reflection, transmittance and so on.

In the ambient, there are infinite light rays emitted from light sources. The reproduction of this number of rays is impractical is simulation systems. Usually, we use the reverse ray tracing, where rays are cast from the view point and tracked back to the light source. Figure 2 shows a schematic view of the ray tracing, where:

O = Observer;
P = Sampled pixel in the visualization area;
F = Light source;
S = Shadow ray;
R = Reflected rays;
T= Transmitted ray.

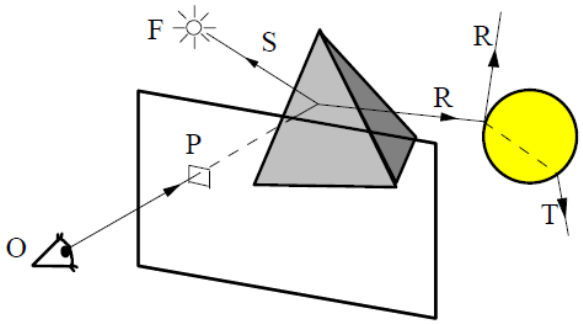

Figure 2: Schematic view of the ray tracing components [Santos 1994].

[Whitted 1980] observes that around 75% processing time of the ray tracing algorithm is spent in intersection calculations. This cost is associated with the fact that the exhaustive ray tracing tests intersections between the rays with every object in the scene. The usage of an acceleration structure, as the octrees, is recommended for reducing the algorithm cost.

5. Data modeling

The following sections presents the data modeling developed in this work for using the octrees in the GPU together with the parallel ray tracing.

5.1 Scene representation

In this work, we represent the scene in two different ways: a polygonal mesh or a point cloud.

A polygonal mesh is created from a set of triangles, each composed by three vertices in the tridimensional space. A vertex may be shared by different faces, which results in some memory saving.

An issue resulting from this representation is that a single face may belong to more than one octree node, resulting in a higher polygon count to the scene. [Madeira 2010] uses only the triangle center to determine if it is inside the node or not. In his system, there is no mention to hierarchy, so it works ok. But here, the hierarchy must be respected, so a triangle should be replicated in all nodes that contain it.

A point cloud is a structure composed by a set of vertices in the tridimensional space. This representation reduces the amount of stored data, since only the vertices coordinates are stored, opposed to the mesh where we also need to store the vertices' index for each triangle.

For the construction of octrees in the tests proposed by this paper, we created a regular grid and populated its cells with points. Each point is treated as a sphere of radius R by the ray tracing algorithm. The smaller octree node – the leaves in the deeper level – have length equal to R all the points are always centered within its grid cell, so a "sphere" is never contained by two different octree nodes.

The usage of a point cloud is explored in works like [Linsen et al. 2007] and [Deul et at. 2010]. It is used here for testing the octree performance without interference from other issues, such as the higher polygon count resulted from the usage of polygonal meshes.

Whatever the chosen scene representation, we need to populate its data into an octree so the ray tracing algorithm can traverse the scene with a lower cost. In this paper, we propose the usage of a hash table and a linear octree, presented in the following sections.

5.2 Octree representation in a hash table

A hash table is a data structure that uses a hash function to map keys into positions inside a table. Given a key, the function is applied to find out the position of the data inside the table.

Depending on the data volume and the hash function, more than one key may be mapped into the same position. When this happens, a new function is used to treat the collision. There are basically two approaches for treating collisions: open chaining, where a list stores the collided data and the new one in stored at the end of the list (this is more useful for CPU implementations with a linked list. On the GPU, it is needed to allocate a fixed size array, which raises the memory usage); and closed chaining, where a second hash function is used until a free space is found and the data is stored.

In this work, we use the hash function as the modulus function, given by $h(k) = k \bmod m$, where m is the hash table size and k is the item key. But, instead of using the rehash function $h'(k) = h(k) + 1$, as proposed by [Madeira 2010], we use a fixed size list for collision treatment. This avoids the worst case where the whole hash table is searched to find a node.

The key of each node is coded with the Morton Code. The octree traversal is done by acquiring the code for each child and applying the hash function in this code to find where it is stored in the table.

5.3 Octree representation in a linear list

An octree may also be stored in a linear list, where the parenthood relations between the nodes are given by its indexes. In this model, the root is stored at the position 0 and the position p of each child is given by $p = 8 * parent\ index + child\ index$, where each child is given an index ranging from 1 to 8. The children of a node are always stored at adjacent memory positions.

Figure 3 shows an example of a linear octree. The root node is at position 0 and its children occupy the

position 1 to 8. The children of node 1 are at positions 9 (8 * 1 + 1) to 16 (8 * 1 + 8), and so on.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17

Figure 3: Example of linear octree.

5.4 Octree data storage on the GPU

The octree is stored on the GPU using some data structures to reduce memory usage at the same time that it doesn't incur in great performance loss.

Each octree node has a key, an index to a primitive list, a type (internal, leaf, empty leaf or invalid), a depth level and a bounding box. The primitives are not stored in the nodes because we want to use CUDA's texture memory for faster access and also we need to store primitives only for leaves nodes, so the internal or empty nodes doesn't occupy unnecessary space.

For the polygonal mesh, the primitives are stored in two different lists: the first stores each triangle and the second stores the vertices. The access to the primitives takes two steps, first we need to fetch the triangle and read its vertices indices. Then, we fetch the vertices and make the intersections tests. For point cloud, we can access the vertices directly and apply the intersection test.

Since the octree stores a high amount of internal and empty nodes, if every node had a fixed allocated space for a list of primitives, too much unused memory would be lost. To solve this problem, only the leaves have a primitive list. Each leaf node has an index to an array allocated in the global memory.

Figure 4 presents an overview of this system. The node 2 is a leaf, so its *primitiveIndex* field points to the memory position where its primitives are stored. Whenever the primitives for this node should be consulted, the data will be fetched from the corresponding position in the memory. This approach still brings some memory loss, since not every leaf will have a full primitive list, but the memory space is fixed in the used GPU architecture. Still, the used space is smaller than the trivial solution where every node has its own internal primitive list.

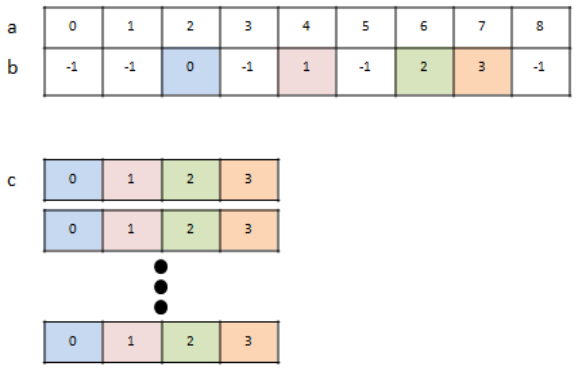

Figure 4: Primitive list representation for each leaf node.

6. Parallel ray tracing with octree

The developed ray tracing system starts with the construction of the octree in the main memory and its upload to the GPU memory. Then, the system runs in a loop of the following steps: upload of visualization data, such as camera and light positions, to the GPU; running of the ray tracing algorithm in parallel on the GPU; acquisition an exhibition of the generated image; update of the visualization data.

The ray tracing algorithm itself runs as follows: for each pixel on the screen a GPU thread is created and a ray is casted. For each ray, we check its intersections with the scene and find the smallest intersection distance. At this point, the illumination model is applied and the pixel color is found.

The intersection calculations depend on the used method. The exhaustive ray tracing tests the ray with each primitive. The octree ray tracing first checks the intersection with the scene root, then tests if there are intersections with its children. Each intersected child is queued for new recursive tests. The recursion stops whenever a leaf, an empty leaf or an invalid node is found, or no intersection happens.

The following listing describes each step of the octree traversal.

```
Function: OctreeTraversal
Input: ray, currentNode
If Intersects(ray, currentNode)
   If(currentNode.type == Internal)
      For each child F of currentNode
         OctreeTraversal(ray, F)
   Else if(currentNode.type == Leaf)
      For each primitive P in currentNode
         If(Intersects(ray, P))
            d = IntersectionPoint(ray, P)
            minDistance=min(minDistance,d)
         return minDistance
```

7. Results

Here we present the results obtained from the tests with the parallel implementation of the ray tracing algorithm using an octree as acceleration structure. The tests were carried out with a GeForce 9800 GT graphics card with 1 GB of memory and 112 stream processors at 1350 MHz running on a PC with Windows Seven and CUDA 3.0 installed.

We tested the algorithm with both point clouds and polygonal meshes with varied complexity. We also tested three ray tracing implementations: the trivial exhaustive one, the hash table based and the linear octree based. Here we present a performance comparison between each one.

The ray tracing was always run to generate a 512x512 pixels image. All the timing results are presented in seconds.

7.1 Point cloud tests

The point cloud tests were made with the point count varying from 1000 to 64000 (for the exhaustive method), 32000 (for the hash table method) and 16000 (for the linear method).

There is a high cost increase related to the data growth with the exhaustive ray tracing. Figure 5 shows how processing time increases fast as the data amount becomes larger. This performance loss is more stable with the hash table octree (Figure 6), even though it is still slower because of the overhead related to the octree traversal and the index manipulation.

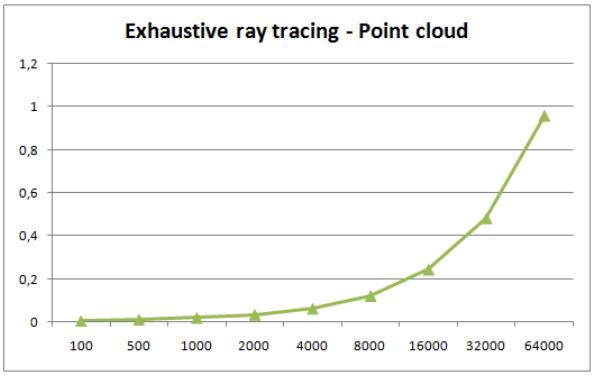

Figure 5: Results for exhaustive ray tracing with point cloud (time in seconds).

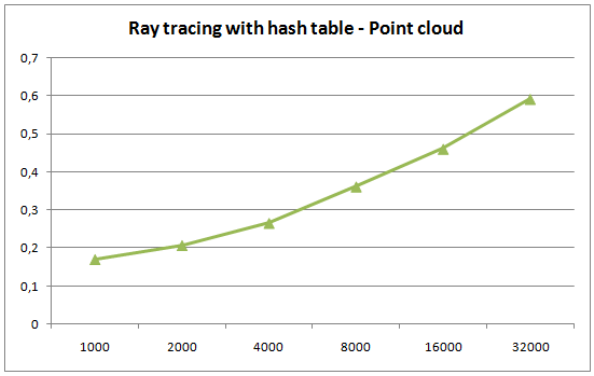

Figure 6: Results for ray tracing with hash table and point cloud.

The linear octree presents a similar performance gain as the cost increase slows as the data volume becomes larger. The exhaustive ray tracing almost doubles its processing time as the data volume doubles, at the same time the linear octree algorithm doubles its processing time as the data volume is increased by 16 times (Figure 7).

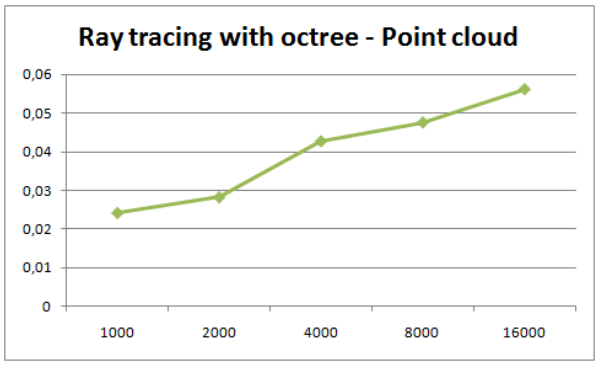

Figure 7: Results for ray tracing with linear octree and point cloud (time in seconds).

If the data volume is high enough, the traversal overhead is compensated and the algorithm presents a great gain compared to the linear scene traversal. Figure 8 presents a comparison between the exhaustive and the octree implementations that demonstrates this. The exhaustive method is faster only with a small data volume where the overhead makes it slower to traverse the octree than to test intersections with the entire scene.

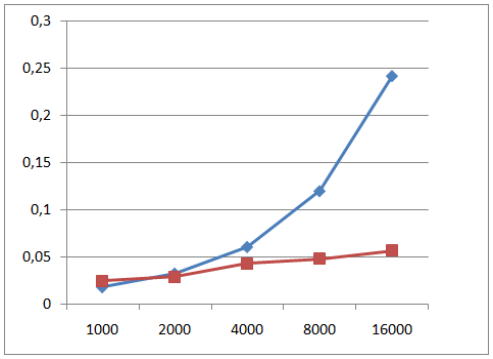

Figure 8: Comparison between the exhaustive (blue) and octree (red) implementations (time in seconds).

7.2 Polygonal mesh tests

We also made some testing with different polygonal meshes. Four meshes were tested with the exhaustive and the linear octree. Because of some faces appear in more than one octree node, the total face count differs from the face number in the raw model, as shown in Table 1.

Table 1: Number of vertices and faces for each polygonal mesh used in the tests.

Name	Vertices	Faces	Octree faces
Cubes	104	156	609
blackmage	824	1644	4757
House	1460	3028	10979
Ant	486	912	4967

Again, the octree implementation is faster with larger models. In Figure 9 the red columns represents the processing time for the exhaustive method and the blue ones are the octree tests. Only the cubes model that has a low polygon count is faster with the exhaustive method.

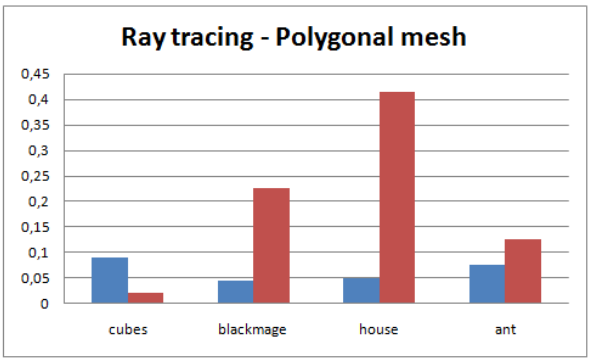

Figure 9: Comparison of the exhaustive (red) and linear octree (blue) implementations with polygonal meshes (time in seconds).

Conclusion

This paper presented a parallel implementation with the ray tracing algorithm using an octree as an acceleration structure for decreasing the number of intersection calculations and increasing its performance. We studied different ways to represent the octree in video memory and presented a representation of indexed primitives for octree leaves, reducing the memory usage.

We tested different traversal algorithms for the ray tracing, such as the usage of linear octrees, hash tables and exhaustive ray tracing. Tests show that the linear octree implementation is more efficient, even though its higher memory load. The linear implementation is faster than the hash table because of the great overhead related to the index manipulation necessary to traverse the hash table.

As future work, we propose the usage of Fermi graphics cards that allows the usage of pointers and recursion. This solves some problems of the presented solution. Also, the usage of a GPU cluster is advised, since the ray tracing is implemented in a way that every ray runs in a separate thread and all the structure is stored on GPU memory, so this adaptation shouldn't be problematic, while a great gain is expected.

References

AKENINE-MOLLER, T. Fast 3D Triangle-Box Overlap Testing. in:• Proceeding SIGGRAPH '05. New York, NY, 2005.

APPEL, A. Some techniques for shading machine renderings of solids. SJCC, p27–45, 1968.

BARBOZA, D., CLUA, E. Ray tracing algorithm using a GPU-based data structure. In: X Simpósio Brasileiro de Jogos e Entretenimento Digital, 2011, Salvador. Proceedings of the X Simpósio Brasileiro de Jogos e Entretenimento Digital - Computing - Full Papers, 2011. (Em processo de submissão)

CASTRO, R. et al. Statistical optimization of octree searches. Computer Graphics Forum, v. 27, p. 1557-1566, 2008.

DEUL, C., BURGER, M., HILDENBRAND, D., KOCH, A. Raytracing point clouds using geometric algebra. Proceedings of the GraVisMa workshop, 2010.

GARANZHA, K., Loop, C. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. EUROGRAPHICS 2010.

HALL, R. A., GREENBERG, D. P. A Testbed for Realistic Image Synthesis. IEEE Computer Graphics and Applications, v.3, n.8, p10-20, 1983.

HORN, D., SURGEMAN, J., HOUSTON, M., HANRAHAN, P. Interactive K-D Tree GPU Raytracing. I3D '07 Proceedings of the 2007 symposium on Interactive 3D graphics and games. ACM New York, NY, 2007.

KIRK, D; CUDA Memory Model. 2008.

KUCHKUDA, R. An Introduction to Ray Tracing. Theoretical Foundations of Computer Graphics and CAD, Springer-Verlag, Berlin, p1039-1060, 1988.

LINSEN, L., MULLER, K., ROSENTHAL, P. Splat-based Ray Tracing of Point Clouds. Journal of WSCG, 15(1-3), 51–58, 2007.

MADEIRA, D. Uma Estrutura Baseada em Hash Table para Buscas Otimizadas em Octree em GPU. Dissertação de mestrado, Universidade Federal Fluminense, 2010.

MORTON, G. M. A computer oriented geodetic data base and a new technique in file sequencing. [S.L.], 1966.

NVIDIA. Nvidia Cuda. Best Practices Guide. 2011.

PHONG, B. T. Illumination for computer generated pictures. Commun. ACM 18, 6 (June 1975), p311-317, 1975. DOI=http://doi.acm.org/10.1145/360825.360839.

PURCELL, T. J., BUCK, I., MARK, W. R., Hanrahan, P. Ray Tracing on Programmable Graphics Hardware. In ACM Transactions on Graphics 21, 3 (July 2002), 703-712.

REVELLES, J. URENA, C., LASTRA, M. AN EFFICIENT PARAMETRIC ALGORITHM FOR OCTREE TRAVERSAL. JOURNAL OF WSCG, P.212-219, 2000.

SEGENCHUK, S. IMPLEMENTATION OF AN ACCELERATED RAY TRACER. 1997.

WALD, I., SLUSALLEK, P. STATE OF THE ART IN INTERACTIVE RAY TRACING. EUROGRAPHICS '01. 2001.