Interactive Ray Tracing

the replacement of rasterization?

A.J. van der Ploeg

Abstract

In the last three decades the quality of interactive computer graphics has increased drastically and there is still a demand for higher quality. However the standard method of computing images, called rasterization, does not allow for advanced effects such as reflections and shadows. For this reason we look at an alternative method which does allow this, called ray-tracing, and investigate if it can replace rasterization. To answer this we look at hardware and software support for ray tracing. First we look at what kind of hardware we need and when this will be available. Secondly we investigate software that can maintain acceleration structures for ray tracing in moving scenes. When we have investigated those two critical topics we speculate about the future of ray-tracing and rasterization.

Table of Contents

1. Introduction

In the last decade the quality of computer graphics has increased drastically. However there is still a need for even higher quality interactive graphics. The computer game industry has grown to a 7 billion dollar industry in the US[2] alone and demands higher quality graphics for more atmosphere and realism in their games. Scientific data sets have grown to a size where it is impossible for any human to analyse them without visualisation, thus there is a need for high quality complex visualisation.

We need high quality interactive graphics effects such as shadows, reflections, bump mapping and refraction. The implementation of these effects on modern graphics hardware is complex and non-intuitive[18] and in many cases the implementation of such effects on modern graphics hardware can only be an approximation due to the limits of the rendering method.

Therefore there is a need for a more flexible and higher quality rendering method for interactive computer graphics. For non-interactive applications *ray tracing* has long been the standard. Recent research has shown that ray tracing is also possible for interactive applications. In this thesis we will investigate what is already available and what is still needed for ray tracing to be an alternative to the current rendering method.

In chapter 2 we will briefly explain what ray tracing is and what its characteristics are. In chapter 3 we will investigate what kind of hardware we need for interactive ray tracing and if this is already available. In chapter 4 we will look how we can maintain an interactive (e.g. moving) scene for interactive ray tracing. In chapter 5 we will summarise and conclude.

2. What is ray-tracing?

In computer graphics, if we have a three dimensional scene we typically want to know how our scene looks trough a *virtual camera.* The method for computing the image that such a virtual camera produces is called the *rendering method.*

The current standard rendering method, know as *rasterization,* is a *local illumination* rendering method. This means that only the light that comes directly from a light source is taken into account. Light that does not come directly from a light source, such as light reflected by a mirror, does not contribute to the image.

In contrast ray tracing is a *global illumination* rendering method. This means that light that is reflected from other surfaces, for example a mirror, is also taken into account. This is essential for advanced effects such as reflection and shadows. For example if we want to model a water surface reflecting the scene correctly we *need* a global illumination rendering method. With a local illumination rendering method the light from the water surface can only be determined by the light directly on it, not the light from the rest of the scene and thus we will see no reflections.

An example of an image produced by a interactive ray tracing system[6] is shown on the cover of this thesis. In this image we see reflections, refraction and shadows. These are advanced visual effects which can only be delivered by a global illumination rendering method such as ray tracing.

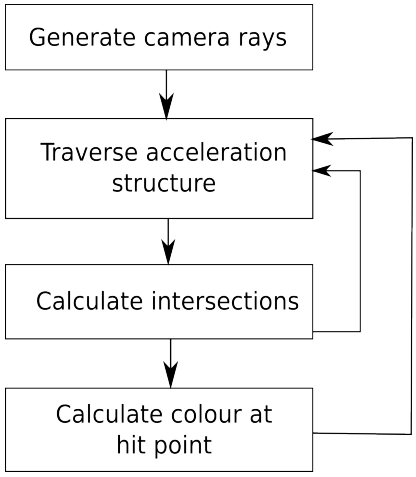
Ray tracing works by following the path of light. We follow the path of *rays* of light, i.e. lines of light. For an example of such a path consider a ray of light from your bathroom light-bulb. This particular ray of light hits your chin, some of it is absorbed, and the rest of the light is reflected in the colour of your skin. The reflected ray is then reflected again by the mirror in your bathroom. This ray then hits your retina, which is useful otherwise you would not see your self shaving. In exactly this way a ray of light in the virtual camera gives the colour of one pixel.

Of all the rays of light that come from the light sources a lot of rays do not end up in our virtual camera. We only want to know about the light on our virtual camera, for this reason we follow the rays of light *backwards.* This means we start at the virtual camera and *trace,* i.e. follow backwards, the ray that determines the colour of the pixel we want to know about. If we encounter the point where the light is coming from we want to know the colour at that point. To compute this colour we need to know what the incoming light at that point was. The incoming light consists of the rays that fall on that point, so we recursively trace those rays. Because rays can originate from both light sources and other surfaces ray tracing is a global illumination rendering method.

In contrast, rasterization works by calculating the on screen position of a triangle and then drawing the pixels of that triangle(*rasterizing* the triangle). The colour of the triangle is only determined by its texture and the positions and a few lights.

In ray tracing the main problem is to find the nearest intersection of the ray with an object. Because there are often millions of objects in a scene simply checking the ray against each object is very inefficient. Often a ray tracing algorithm cuts down on these intersection checks by using a *spatial index structure.* Using such a structure we can check if a ray is in the vicinity of the object before we check for an intersection. For example we can divide the space into a grid, and only check

for intersections in areas of the grid the ray passes trough. In this way we can minimize the amount of intersection checks.



When using such a spatial index structure ray tracing scales very well with the size of the scene. More precisely ray tracing scales logarithmically with the size of the scene[18] because adding a number of objects to a scene does not mean a linear increase in the amount of intersection checks. This is because objects are only checked when the ray passes trough its area. With rasterization however the time would increase linearly as we would simply draw extra primitives. Using a good spatial index structure is very important for the performance of ray tracing, we will explain more about them and how to keep them up-to-date when the scene is changing in chapter 4.

Generally the process of ray tracing can be seen as illustrated in the figure on the right, this diagram is a simplified version of that of Purcell et al.[3]. First we generate rays originating from the camera. We then follow (traverse) the ray trough an spatial index structure, such as a grid. For each area the ray passes trough we calculate if the ray intersects with an object in that area. If there are no intersections then we continue traversing the next area of the spatial index structure. If the ray does hit something then we calculate the colour of that point. This is also known as the *shading* computation. The shading computation may require that we trace more rays, this makes ray tracing a global illumination rendering method as these rays can come from both light sources and other surfaces.

This simple but flexible rendering method makes ray tracing a much more suitable environment for advanced effects than rasterization[18]. With rasterization we need complex and non-intuitive operations for such effects and often these effects are not possible at all because of the limitations of a local illumination rendering method.

Although ray tracing is a far more suitable environment for advanced effects it is traditionally known as being slow compared to rasterization rendering. The big difference between ray tracing and rasterization is that they work the other way around: Rasterization takes a primitive and draws it on screen, which is a very fast operation. With ray-tracing we look for each pixel which primitive is under it. Suppose for example we would want to render a simple cube: With rasterization we would simply draw 16 triangles. With ray tracing we would need more computations as we would need to trace rays for each pixel on screen. This is why ray tracing is traditionally known as being “slow” compared to rasterization rendering.

Ray tracing is not actually “slow”, the rendering time being logarithmic with the size of the scene. It does however have a high *initial cost*. With a very complex scene with advanced effects, it would be more efficient to use ray tracing than rasterization[15]. This is because the rasterization approach would always draw *all* triangles and overwrite triangles which are further away. This means a lot of redundant operations. Additionally if the advanced effects are at all possible they would need multiple rendering passes. With ray tracing we would simply have no redundant calculations and would not need multiple rendering passes.

With highly complex and high quality graphics the cost of rasterization rendering, with its redundant calculations and ineffective advanced effects, is higher than the cost of ray tracing.

Computer graphics are using increasingly complex scene and advanced visual effect and are nearing the boundary where ray tracing becomes more efficient. This boundary has already been crossed by state-of-the-art applications[18].

3. Suitable hardware for ray tracing

3.1 Introduction

The first thing we need for interactive ray tracing is suitable hardware to run our ray tracer on. Therefore in this section we will look at what suitable hardware for ray tracing is and if this is already available. In this section we will introduce what the characteristics of suitable interactive ray tracing hardware are and how we can make optimal use of the hardware. In the rest of this chapter we will look at different types of hardware and how suitable they are for ray tracing.

Please note that the discussion below is about system which run ray tracing on simple triangles but most of it also holds for bezier curves, volumetric data sets and other more complex types of surfaces.

3.1.1 Requirements for suitable hardware

For hardware to be able to efficiently run ray tracing it needs[6]:

- Parallel calculations

  Ray tracing is known is as being “embarrassingly parallel”. This is because rays do not depend on each other. Because of this it would be very efficient if the hardware can trace rays in parallel.

- Large amount of floating-point operations

  The computation of the intersections of the rays and, when the rays hit an object, the computation of the colour of the surface requires a lot of floating point operations.

- Complex flow control

  When a ray hits an object other rays may be traced recursively and it is always uncertain how many rays deep this will go and how many branches the calculation will have. This means the hardware needs to be able to handle complex flow control like recursion.

- A lot of memory access

  Every ray may intersect with any point in the scene and the appropriate data should be fetched for computation. This means a lot of memory access.

Using these criteria we will see if hardware is suitable for ray tracing or not.

3.1.2 Mapping ray-tracing to hardware

While it is important that the hardware is suitable for ray tracing it is also important that the ray tracing implementation is suitable for the hardware. To make a ray tracing implementation suitable for hardware it needs to be optimized for the specific characteristics of the hardware.

An often used type of optimization that is effective on different kinds of hardware is tracing *coherent packets of rays*[1]. A coherent packet of rays is a set of rays that are close to each other. Often when we calculate a portion of 3x3 pixels of the screen the 9 rays that must be traced will be

bundled in a packet, since they lie very close to each other.

If we trace a coherent packet of rays at once instead of a single ray we will have two benefits:

- Tracing *coherent packets of rays* can dramatically increase cache utilization because rays from the same packet have a high chance of using the same data, making a cache hit more likely. Additionally data that is needed for several of the rays in the packet at the same time only needs to be fetched once, because we can merge the memory requests. This makes the memory access less unstructured so we can optimally use caches and have less strain on main memory.

- Tracing *coherent packets of rays* enables us to make use of the *SIMD instructions* available on a lot of modern hardware. A Single Instruct on Multiple Data or SIMD instruction is an instruction which executes a floating point operation in parallel on a number (often two or four) of data values. We can use those instructions to operate on a whole packet of rays with one instruction.

  We will see that using such optimizations is necessary to make optimal use of the hardware.

3.2 Traditional PC

3.2.1 Description

Recently Wald et al.[1]have researched interactive ray tracing on a cluster of commodity PCs. In their set-up they use a client-server model, with a single server and arbitrary number of clients. The server gives tasks to compute a part of the screen to the clients. When the clients have finished their task they give the result back to the server which composes all the results into a single image. Because a network is relatively slow compared to main memory each client has a complete copy of the scene data in its memory. This system was able to handle all of the functionality of ray tracing, e.g. shadows, reflections.

Wald et al. have found trough measurements that a ray tracer on a CPU is mainly bound by the bandwidth to main memory. To minimize the use of main memory they have tried to make optimal use of caches by tracing coherent packets of rays as described in 3.1.2. They also used the packets of rays with SIMD instructions. Together this gave a very significant speed-up but the ray tracer was still bound by bandwidth to main memory.

This system running on five PC's (Pentium III-800Mhz) gave interactive frame rates of 7.7 frames per second on a static scene with 907k triangles.

3.2.2 Discussion

The fact that this implementation is largely bound by the memory bandwidth, even when optimally using caches suggest that the memory bandwidth of a traditional PC is simply not enough for ray tracing.

Although we can increase the performance drastically by tracing packets of rays, the need for

a cluster of PCs suggest that PCs do not have enough floating point computing power and lack parallelism themselves. In fact the sheer amount of hardware required to get acceptable frame rates shows that a traditional PC is not efficient for ray tracing.

This suggests that a traditional CPU lacks three things needed for ray tracing: parallelism , floating point computing power and high memory bandwidth.

3.3 Programmable Graphics Processing Unit

3.3.1 Description

Graphics Processing Units (GPUs) traditionally can run a lot of floating point computations in parallel and have relatively high memory bandwidth, which is beneficial for ray tracing. However modern GPUs cannot handle the complex flow control required by ray tracing.

1. GPUs have recently become programmable to allow more complex visual effects. This is the first step in a trend for GPUs to become more general purpose, i.e. more like a CPU. In the near future their functionality will be general enough to run a full ray tracer. Purcell et al.[3] have built such a ray tracer based on propositions for the next generation of APIs such as OpenGL. They have researched how such a ray tracer would work and what its speed would be.

Such a ray tracer would be running only in one part of the graphics pipeline (the fragment shader for those who know the graphics pipeline). They would use that part of the pipeline for different programs in different stadia of the rendering process. The scene data would be stored in textures and the (intermediate) results in screen buffers.

Purcell et al. have researched two kinds of future GPU architectures: A multipass architecture in which the complex loop control required by ray tracing has to be handled by multiple passes and a branching architecture on which that complex loop control can directly be implemented without multiple passes. The multipass architecture is derived from proposals for actual new technology, where the branching architecture reflects the researchers view on future hardware but there are no tangible plans to actually manufacture such an architecture. Both resulting systems have been implemented on a simulator.

The multipass Architecture has shown to be memory-bound because of the large amount of memory writes and reads needed for temporary data between the multiple passes. It ran about 2.5 times faster than a ray tracer on a single CPU as researched by Wald et Al. The Branching Architecture has shown to be compute bound, and needs much less memory bandwidth than the Multipass Architecture, because there is no need to store intermediate results in screen buffers. This implementation ran at four times the speed of the multipass architecture.

3.3.2 Discussion

Modern GPUs lack the complex flow control needed for ray tracing. However as GPU technology will become more general, ray tracing on it will become more efficient. In the next 5-10 however it is doubtful if ray tracing on programmable graphics units will become efficient enough to be an alternative to triangle rasterization since the primary intended use of this hardware still is

triangle rasterization. Once the architecture of a GPU becomes as general as the Branching Architecture described above, ray tracing is very likely to be as efficient or more efficient than triangle rasterization, especially on complex scenes.

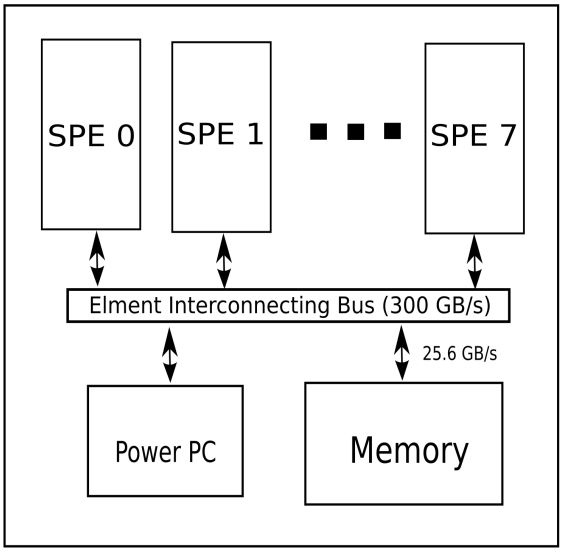
It is likely that if nothing disrupts this trend of GPUs becoming more general interactive ray tracing will gradually be used more in applications. At first it will be very handy that we can do ray tracing for effects such as shadows real time on the GPU while keeping rasterization as the primary method of rendering. While GPUs become even more general and the need for more complex scenes continues there will come a time that ray tracing will be a good alternative to rasterization as a primary rendering method. In this way there could be a slow transition from rasterization to ray tracing[3].

3.4 Next generation processor

3.4.1 Description

It is increasingly difficult for CPU developers to get more performance out of putting more transistors on a single chip. Additionally the amount of power used and the heat produced by CPUs is an increasing problem. For these reasons CPU designers are looking for more performance in parallelism instead of faster serial execution of code. GPUs are becoming more like CPUs and CPUs are becoming more like GPUs. For these reasons there is a trend for processor chips to become *multi-processor systems on a single chip* [4].

The first processor from this new trend is called the *Cell* processor. It contains a Power PC core and eight “synergistic co-processor elements” which are especially designed to work on *streams* of data, sets of data on which the same operations need to be performed. A simplified schematic[4] of the set-up can be seen in the figure on the right. The SPEs are connected to each other , to the memory and the power pc core trough a high bandwidth bus. The intended use of the cell processor is to have each of the SPEs perform a piece of the calculation and send the result for further calculation to next one over the high-bandwidth bus in pipeline fashion.



Benthin et al.[4] have implemented a ray tracer on the cell processor. They found that ray tracing does not map to a pipeline system on the cell very well. This is because of the complex flow control of ray tracing, it is not known in advance how long a part in the process of ray tracing takes. Once a part of the pipeline becomes a bottleneck it would starve the other SPEs. This could be solved by buffering tasks for SPEs, but they have no memory of their own so this would have to be done in main memory. This would be too much for an already heavily memory-intensive application.

Instead of a pipeline design they let each SPE run a full ray tracer. This design resembles the PC cluster design of Purcell et al. The power pc core is like the server, it hands out task to the SPEs. The SPEs trace packets of rays in parallel and use (self-maintained) caches. They show that a single SPE of 2.4 GHz is up to par with a ray tracer of the same clock speed on a full x86 CPU. They also show that performance increases linearly with the number of SPEs. Likewise with the number of full cell processors, two cell processors is twice as fast.

3.4.2 Discussion

The fact that a ray tracer on the cell processor cannot be put into a pipeline model which the cell is intended for suggests that this is not the ideal hardware for real time ray tracing. However the memory bandwidth and parallel floating point computation power are quite beneficial for ray-tracing. Thus the results shown by Purcell et al. are quite good and are much faster than conventional CPU implementations.

Possible a first use of real time ray tracing could come from the use of the cell processor in the upcoming *Playstation 3* video game console, a platform where the demand for high quality graphics is high. Purcell et al. note that on the Playstation 3 there exists a high-bandwidth connection between the cell processor and the GPU. If that can be used so that intersection calculations can be done on the cell processor and shading computations can be done on the GPU, real time ray tracing might very well be a good alternative for rasterization on the Playstation 3[4].

When the trend of processors to become more suitable for parallel computations continues ray tracing on such a next generation could be very efficient, especially if it has more support for complex flow control than the cell processor. Additionally if there is a high bandwidth bus between such a processor and the GPU we could combine the powers of both to do ray tracing.

3.5 Specialized Hardware

Standard PC's have too limited memory bandwidth for ray tracing. GPUs do not support the complex flow control required by ray tracing. On a cell processor we can obtain good results but we note that we cannot make use of the pipeline optimized design of the processor. All of this suggests that it may be a good idea to make *specialized* hardware for ray tracing since the ideal hardware is not available.

3.5.1 Fixed Function Architectures

Schmittler et al.[5] have proposed a design for a hardware architecture especially for interactive ray tracing. Their custom chip is named the SaarCOR system (Saarbrucken's Coherence Optimized Ray tracer). As the name suggests this system is designed to trace coherent packets of rays, as described in section 3.1.2, to optimally use caches.

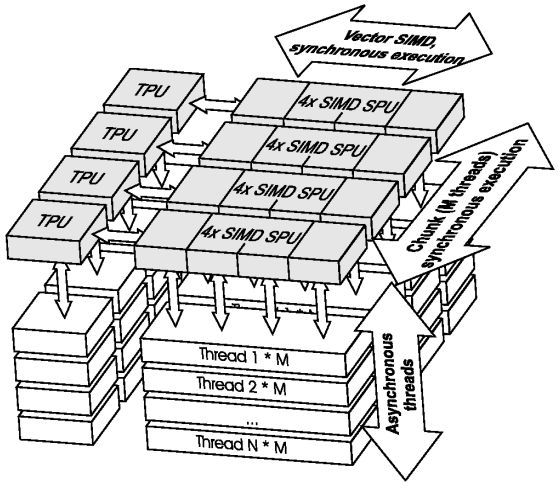
Their system is divided into three parts: ray generation and shading, the memory interface and the ray tracing core. The ray generation and shading part generates new rays and does the shading computation. In the ray tracing core the spatial index structure is traversed and intersections are calculated. The memory interface has separate caches for several types, to optimize cache utilization, and feeds data to the ray tracing cores and the ray generation and shading parts. The

system is designed to be flexible, so while there are two ray tracing cores in the initial design more can be added for extra performance. To hide memory latencies both the ray generation and shading part and the ray tracing cores can continue operating on another ray while waiting for data.

The resulting system could theoretically be built at a cost comparable to the cost of current rasterization chips. In their tests Schmittler et al. have shown that the system would also deliver speed that does not differ much from the speed delivered by current rasterization chips, while at the same time delivering all the benefits of ray tracing.

3.5.2 Programmable Hardware Architecture

With modern rasterization hardware certain parts of the pipeline have recently become programmable. Instead of also making parts of the pipeline programmable in ray tracing hardware Woop et al.[6] have proposed making a general purpose processing unit optimized for tasks like ray tracing. The intent of this system is to combine the best features of CPUs, GPUs and custom hardware. Their system is known as the RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. The system contains a number of SPUs(Shader Processing Unit)s which have similar functionality as modern GPU systems and are intended for general use. The SPU design is optimized for vector operations and differs from modern GPUs in that it allows recursion and memory writes. Each of these SPUs runs a hardware thread at a time and can switch threads when it has to wait for data or when a thread is done. Each of these threads deals with a coherent packet of rays. Spatial index structure traversal on such a SPU would not be very efficient since this requires a significant number of scalar floating point operations, while an SPU is optimized for vector operations. For this reason a specialized Traversal Processing Unit is also present to do this task. This Traversal Unit is called with a special instruction on the SPU, which then runs another thread while waiting for the results.



This system has been optimized to make even more use of the coherence between rays. Each time a number of threads are issued together, and they are executed by a number of SPUs synchronously. Such a group of threads is called a chunk of threads. On the RPU a number of chunks run asynchronously, while within these chunks all operations are run synchronously. This is illustrated in the figure[6] above. When a chunk does a memory request the chances are high that they issue a request for the same data since the packets that were issued to the SPUs were close to each other. These requests can then be combined to drastically cut down on memory usage.

This system was implemented using FPGA(Field Programmable Gate Array, a type of programmable hardware). While such a system is not very efficient compared to true hardware implementation a system running at a mere 66Mhz provided the same ray tracing performance as a modern CPU implementation. The programmable design does have its price tough: this system gives 20% till 50% of the frame rate delivered by the fixed function SaarCOR design, although it is

hard to compare the two systems because of the different set-up in terms of memory and the traversal algorithm used.

3.5.3 Discussion

Clearly specialized hardware implementations have proven to be more efficient than other implementations. However it is questionable how soon this technology will be widely available for desktop PCs. This largely depends on whether commercial hardware manufacturers will make such a board.

3.6 Conclusions

As we have seen the hardware that is currently available is not optimal for real time ray tracing. CPUs cannot run enough floating computations, lack parallelism and have too limited memory bandwidth. GPUs do have enough parallelism and floating point computing power but cannot handle the complex flow control required by ray tracing. The first in the next generation of processors, the cell processor, already gives great results but ray tracing does not map into the pipeline model the cell was intended for. Specialized hardware yields great results on paper but this hardware is not available since there is no hardware manufacturer selling such hardware.

The hardware that we need is like a CPU in terms of generality and like a GPU in terms parallel computations and massive floating point computations. Luckily there is a trend in CPU design towards more parallelism and there is a trend for the massively parallel GPUs to become more general. The lines between a CPU and a GPU are blurring.

On one side GPU manufacturers have announced plans to gradually make more general GPUs. This development allows for a gradual transition from rasterization to ray tracing. This is because when rasterization as well as ray tracing can be done in real time on a GPU, both can be combined and used where it is most efficient.

On the other side all of the major CPU manufacturers have announced multi-core architectures such as the cell processor. Analysing the road maps of the CPU manufacturers we can safely say that hardware on which ray tracing will be very efficient will be available in the next 5 to 10 years[4].

A hybrid scheme where the CPU calculates intersections and the GPU does the shading computations might also be possible depending on the bandwidth of the bus between the GPU and the CPU. Designs for the upcoming playstation 3 video game console already have a bus that has adequate bandwidth for this.

Another possibility is not waiting for CPUs or GPUs to come to a point where ray tracing is efficient but to make such hardware now, as it is already possible. Such an architecture is proposed by Woop et al. They have designed a general stream processor that is in functionality somewhere between a CPU in terms of generality and a GPU in parallelism. It has been shown that such a design can be made using current technology.

As CPUs and GPUs grow towards each other both types of hardware will become increasingly suitable for real time ray tracing, because the optimal hardware for ray tracing is somewhere in between. Additionally if a high bandwidth bus between the CPU and the GPU exists

we can also use both to do ray tracing. If a major manufacturer starts manufacturing a board for real time ray tracing like that of Woop et al. we could have very suitable hardware for real time ray tracing even sooner. With all these developments hardware will be increasingly suitable for real-time ray tracing, and we speculate that consumer hardware will be adequate for interactive ray tracing before long.

4 Support for Interactive Scenes

4.1 Introduction

When we have suitable hardware for ray tracing we also need software to support ray tracing. Critical software for speeding up a ray tracer is about building and updating acceleration structures called *spatial index structures*.

A spatial index structure is a structure that divides the space into areas. When tracing a ray or a packet of rays we use this information to check for intersections with objects only in the areas visited by the ray. For example we can use a three-dimensional grid to divide the space and only check for intersections with objects in areas of the grid the rays actually pass trough instead of checking for intersections with **all** objects. In this way we can really cut down on the amount of intersection checks and speed up ray tracing.

Depending on the type of structure generating a structure can take a very long time. There exist a lot of the types of spatial index structures and a lot of research has gone into them. Havran et al.[7] have performed a statical comparison of spatial index structures and have concluded that building a more efficient spatial index structure takes more time, but this pays off when the amount of rays to be traversed trough the structure is large enough.

For static scenes this spatial index structure can be computed in advance. We would then use that same structure for each frame. In this way the performance increase by the spatial index structure comes “for free” because the building cost does not effect the frame rate. However this approach does not allow scenes to be truly *interactive* because the user can walk or fly trough a scene but cannot interact with the environment. It is impossible to move an object or have animations in the scene without also changing the spatial index structure.

For this reason we will look at how to keep such spatial index structures up to date for moving scenes in this chapter. This support for interactive scenes in interactive ray tracing is essential because otherwise real time ray tracing will be confined to simple walk-troughs, which does not make it much of an alternative to rasterization.

In this section we will look at what types of motion that can occur in a scene and the difference between updating and rebuilding the structure. In the rest of this chapter we will look at different spatial index structures and how they handle the different types of motion.

4.1.1 Types of motion

Different strategies for keeping the spatial index structure up to date perform differently on different types of motion. To be able to analyse and compare the spatial index structures we divide

the animation in a scene into four categories[15,16]: *Static*, *Hierarchical motion, Continuous Dynamic motion* and *Unstructured motion*.

Static is for all objects that remain static at all times.

Hierarchical motion is when a set of primitives undergo the same transformation. These transformations are normally stored in a tree called the *scene tree*, hence the name hierarchical motion. For example when we move a whole object such as a teapot trough a scene.

Continuous Dynamic motion is when the triangles of an object itself are transformed. For example if we have a bending finger.

Unstructured motion is when a set of primitives move in an unstructured way, without relation to each other. For example triangles animated by a particle system.

4.1.3 Rebuilding or Updating

When keeping a spatial index structure up-to-date in a dynamic scene we have two ways to do this: rebuilding the structure for each frame or updating the structure for each frame.

- **Rebuilding**: When we rebuild the structure for each frame we impose no restrictions on the kind of motion that is supported. This is because no matter what kind of motion there is in the scene the whole index structure is built from scratch each time. Although this method is very general and supports all kinds of motion, rebuilding the spatial index structure can take a lot of time, especially for efficient spatial index structures.

- **Updating**: If we update the spatial index structure each time instead of rebuilding it, it can be useful to know what kind of motion occurs where *in advance*. When we have this information we can build our structure around it and keep it updated more efficiently. If we did not have this information it would be harder to keep the spatial index structure up-to-date because the motion would seem completely random.

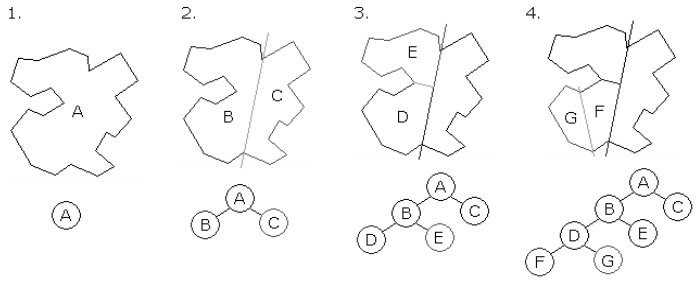
Rebuilding the structure is more general but requires more time while updating the structure is more specialized and needs information about the scene in advance.

4.2 Scene Partitioning Tree

4.2.1 Description

A common type of spatial index structures are *scene partitioning trees*. Such a tree divides the space at each node and the leaves contain the primitives of the scene.

As an example consider a *BSP-tree*, or *Binary Space Partitioning Tree.* A BSP-tree is a binary tree where at the root node the space is split in two by an arbitrary plane trough the space. The left and right subtree then cover the left or right part of the space respectively. Recursively the root node of the right and left subtree divide their space by a plane. The leaves of tree contain the objects in the scene. An example of a BSP-tree is given above[17], if for example there were any primitives in F they would be the children of that node.



Such a scene partitioning tree can then be used to speed up ray tracing by using it to only check for intersections in the areas the ray visits. For example a whole subtree can be discarded if it is determined that the ray does not enter that subtree. We can visit the nodes of the tree of the scene in the same order that the ray traverses trough the scene. For this reason we can stop when we have found the first intersection, since it is the intersection closest to the camera.

The reason that most current real time ray tracing implementations use such scene partitioning trees is that the speed-up gained from them is excellent. The downside is that building an efficient scene partitioning tree is quite expensive. An efficient scene partitioning tree is a balanced tree with about an efficient depth and efficient number of leaves per node. The procedure for building a scene partitioning tree is reasonably complex because it is hard to decide where to place the splitting plane at each node to obtain an efficient tree. The complexity of this procedure is O(N log N)[8], where N is the number of objects in the scene.

Because building a scene partitioning tree typically takes so long it is often infeasible to rebuild the tree every frame to support animation. It is also not possible to update the location of objects in the tree, because after an update an object may intersect with a splitting plane and the object would have to be present in two areas at once which is impossible in a scene partitioning tree.

Most current interactive ray tracing systems, such as the ones presented in chapter two, use a spatial index structure called a *kd-tree* (k dimensional tree although we will always use three dimensions). A kd-tree is simply a BSP-tree but with the extra requirement that all splitting planes must be perpendicular to the coordinate system axes. The complexity of constructing a kd-tree is also O(N log N), where N is the number of objects in the scene. The advantage of a kd-tree over a BSP-tree is that the check if the ray is on one side of the splitting plane is very fast because the plane is perpendicular to the coordinate systems axes.

4.2.2 Two level scheme

The only use of space partitioning trees for dynamic scenes is given by Wald et al. [12]. They propose a scheme for using BSP-trees in dynamic environments. Their approach is based on the observation that there are typically groups of primitives undergoing the same motion in a scene. Each group has his own BSP-tree, defined in a local coordinate system. Additionally there is a top-level BSP-tree of which the leaves are the groups. When traversing a ray trough two-level structure we only has to transform the ray into the local coordinate system when switching from the top-level BSP-tree to the local BSP-tree of one of the groups.

When groups undergo any kind of *hierarchical animation* (as defined above) we only need to update the coordinates of the group and rebuild the top-level BSP-tree. Rebuilding the top-level BSP-tree is not as costly as it might seem because the top-level BSP-tree does not have thousands of arbitrary primitives as leaves but instead has the axis-aligned bounding boxes of the groups as leaves. Because of the simple shape and alignment of axis-aligned bounding boxes and the relatively small number of groups a top-level BSP-tree can be built very fast. It is therefore not a problem that

this structure is likely to be rebuilt for each frame.

When there is any kind of *Continuous Dynamic motion* or *Unstructured motion* in a group the local BSP-tree needs to be rebuilt. Since optimal BSP-tree construction is very costly it is not feasible to do this for every frame. Instead Wald et al. propose to quickly create a less optimized tree to overcome this problem. Off course this leads to less efficient ray tracing but the speed is still acceptable. This scheme allows for the interactive ray tracing of scenes with an arbitrary number of hierarchical animations and a limited number of *Continuous Dynamic motion* or *Unstructured motion.*

4.2.3 Discussion

Although scene partitioning trees give a very good speed-up on a static scene it is very hard to use them with dynamic scenes. This is largely due to the amount of time it takes to construct such a tree for a scene and the fact that updating a scene partitioning tree is impossible. Typically building a tree takes more time than it takes to render a single image. This makes rebuilding the structure for a whole scene infeasible.

Wald et al. have developed a scheme that uses a scene partitioning tree that shows great results for hierarchical motion and a small number of object undergoing continuous dynamic motion or unstructured motion, which is adequate for a large number of applications. However there are also a large number of applications that have a lot of continuous dynamic motion or unstructured motion. Games for example often use continuous dynamic motion to portray characters and other organisms, for example plants in the wind. Unstructured motion is for example found in particle systems. For these types of applications Wald et. al's approach is not enough.

It is doubtful if scene partitioning trees can ever be used for such purposes, as this would require costly rebuilding large trees often.

4.3 Grid-based approaches

4.3.1 Description

Because the cost of building a spatial index structure matters when ray tracing animated scenes it would be logical to minimize that cost. A type of spatial index structure that takes relatively little time to generate are grid-based structures. A standard grid partitions the space into *n x m x l* equally sized cells. Because of this simplicity any kind of grid-based spatial index structure can be built in O(N) where N is the number of objects in the scene. Ray tracing is then accelerated by simply visiting the cells a ray passes trough and checking for intersections in those cells. Because we can traverse a grid in the same order a ray moves trough space we can simply stop when we've found the first intersection.

Because building a grid takes little time it is feasible to rebuild the whole grid for each frame. Because we can rebuild the grid each time it does not matter what kind of motion was present in the scene thus grids perform the same on all kinds of motion.

On static scenes kd-trees are much quicker than simple grids. This is because grids lack any kind of balance when the distribution of primitives in a scene varies. For example take the classic

“teapot in a stadium” problem. Suppose that for some large undetailed object (i.e. the stadium) the amount of primitives per cell is pretty efficient. The small, detailed object (i.e. the teapot) is so small that it occupies only one cell, but it has a large number of primitives. When rays do not enter the cell where the teapot this scheme would be reasonably efficient, but when a ray enters the cell with the teapot we have to be check collisions with **all** the primitives of the teapot. The problem is that because all the cells of a grid are equally-sized often the amount of primitives in the cells varies greatly. This lack of balance makes grids slower than scene partitioning trees because it will result in more intersection checks.

There exist several types of grids that have a varying resolution to overcome this lack of balance, such as multi-level grids, recursive grids and hierarchical grids. For example a recursive grid is a grid where a cell exist of another grid when the granularity at that cell is high. A cell of such a grid in a grid may then recursively be another grid. In this way the resolution of the grid varies with the granularity of object in the scene. The downside of these approaches is that they have a higher building cost than simple grids although it is still much cheaper than building a scene partitioning tree. However for highly complex scenes even these approaches are too limited because the cells of a grid at a certain level are all of the same size. In general scene partitioning trees partition the space much more tight around objects and are much more balanced than varying resolution grids.

Wald et al.[14] have obtained reasonable results for any kind of motion using a two level grid hierarchy scheme. Reinhadt et al. [9] have implemented both a grid and a hierarchical grid and have shown a recursive grid is worth the effort on scenes with varying granularity and that this can be used on interactive scenes. For hierarchical motion those implementations are not as fast as specialized acceleration schemes as the two level kd-tree.

4.3.2 Discussion

Grid based approaches are the most general when it comes to supporting different kinds of animations. Grids can be built very fast and there is no difference between static, hierarchical animation ,continuous dynamic motion or unstructured motion. There is a cost for this generality however: Because grids do not partition the space tight around object and there is little balance in the amount of primitives in a cell the speed-up gained from grid is far less than other approaches.

An advantage of a grid based approaches is that it can be handled very well by stream processors such as modern or future GPUs or next-generation processors such a the Cell, this is because the construction of a grid does not require complex flow control.

4.4 Bounding Volume Hierarchies

4.4.1 Description

A different kind of spatial index structures are *bounding volume hierarchies*. In a bounding volume hierarchy every node has a *bounding volume.* A bounding volume is a volume, for example a box or a sphere, around the contents of the subtree of the node. At the leafs the bounding volume is around a single object. As we walk up trough the tree each node has a bounding volume around the whole subtree of that node. Ray tracing can then be accelerated by recursively checking if the ray intersects

with the bounding volume and if it does not the whole subtree can be discarded.

Building an efficient bounding volume hierarchy is similar to the construction of a kd-tree: the efficiency depends on how objects are placed into groups. The complexity of constructing a bounding volume hierarchy is also O(N log N)[10], where N is the number of objects in scene.

A large difference with other kinds of spatial index structures (i.e. grids, space partitioning trees) is that a bounding volume hierarchy does not partition the space into distinct areas. With bounding volume hierarchies the volume (area) of any two nodes may overlap. When two or more bounding volumes overlap we need to check for intersections in all of them and then pick the nearest intersection. This is a disadvantage of bounding volume hierarchies because it does not always allow us to visit areas of the scene from in the order the ray visits them, which is possible with kd-trees or grids. However it has been show[10] that bounding volume hierarchies can compete with kd-trees in terms of performance when using optimized traversal schemes and optimized tree builds.

An advantage of bounding volume hierarchies not partitioning the space into distinct areas is that it is possible to update the bounding volume hierarchy structure when an object moves. When an object moves in a bounding volume hierarchy we need to update the bounding volume of the object and the bounding volume of all its supernodes.

4.4.2 Updating the volumes

Wald and Boulos[10] have researched the performance of using a bounding volume hierarchy in ray tracing. Their approach was to simply generate an optimal bounding volume hierarchy before beginning the animation and then refitting the bounding volumes each frame. This way the bounding volumes change but the topology of the hierarchy does not.

A downside of this approach is that, since the topology of the hierarchy does not change, no objects can deleted or added. This means the scene always consists of the same primitives.

Another downside of this approach is that the quality (efficiency) of the bounding volume hierarchy can quickly degrade. For example suppose two objects are children of the same node were originally very close to each other but are now on opposite sides of the scene then the bounding box of the parent node of those two objects is now as wide as the scene. Because the bounding box of the parent node is no longer “tight” around the objects there is a much larger chance that rays that pass trough the box do not intersect with one of the children at all, although they will be checked for that. This increases the amount of ray intersection checks and thus brings down performance.

It is very unpredictable when this method is efficient. This depends on the tree that was generated by the algorithm at the beginning and how much it corresponds to what is going to happen in the scene. This works well if for example the characters are at a rest-pose at the beginning, however if a character is grabbing his head at the beginning it would fail since the algorithm would not distinguish the arms from the head[10]. If the algorithm for constructing the tree had more information about what was going to happen in the scene it could make a tree reflecting the scene every time.

However Wald and Boulos have shown that for a large fraction of the applications the performance of this strategy is quite adequate. The method fails when rendering unstructured motion scenes because after a while the the hierarchy no longer reflects the structure of the scene. With optimized traversal and hierarchy builds their method yielded results about 2.5 times as fast as

the grid scheme of Wald[14]. For static scenes this method yield about half the frame rate of a highly optimized implementation for static interactive ray tracing.

4.4.3 Rebuilding the Hierarchy

The above scheme is limited to scenes without unstructured motion because the bounding volume hierarchy becomes inefficient over time due to that the topology of the hierarchy no longer reflects the structure of the scene. Lauterbach et al.[13] have proposed to rebuild the hierarchy once in a while to overcome this problem.

The bounding volume hierarchy is rebuilt when the quality of the bounding volume hierarchy has degraded. The criteria for this is if a bounding box has grown a certain amount while the objects within that box have not grown. This implies that the objects in the bounding volume have move further apart. To make a quick rebuild possible they used a faster but less optimal algorithm to generate the hierarchy than Wald and Boulos.

This approaches makes unstructured motion faster than when not using rebuilds but for scenes without unstructured motion this method is slower than Wald and Boulos' because the original hierarchy was less efficient.

4.4.4 Discussion

It has been shown[10] that *bounding volume hierarchies* are competitive with the most efficient spatial index structure, kd-trees. In contrast to kd-trees bounding volume hierarchies can be updated to keep track of animations in the scene. The restriction is that the amount of primitives in the scene cannot change in time, although it may be possible to invent an algorithms to insert and remove primitives for a scene.

Wald and Boulous have shown that simply updating the bounding boxes gives good results on a large fraction of the applications (namely those with only static, hierarchical and continuous dynamics motion). The method fails when the hierarchy no longer reflects the structure of the scene, which is mostly caused by unstructured motion.

Lauterbach et al. have shown that bounding volume hierarchies can also be rebuilt when the hierarchy becomes to ineffective, which allows unstructured motion. However this support comes with a big performance hit so if there is a lot of unstructured motion a grid-based approach would be faster.

4.5 Conclusions

Spatial index structures are very important for interactive ray tracing: the speed of a ray tracer greatly depends on it. Most current ray tracers are limited to simple “fly-troughs” because the spatial index structure is generated once as a preprocessing step before the actual rendering. In this way the spatial index structure cannot change and thus neither can the scene. To support animations and user-interactivity in the scene we need to either periodically rebuild the spatial index structure or update it. There exist several types of spatial index structures and they respond differently to the different kinds of motion: static, hierarchical, continuous dynamics and unstructured.

Scene partitioning trees perform very well for static scenes, and using a two level scheme they can also be used very well for hierarchical motion. When there is continuous dynamics or unstructured motion (part of) the tree needs be rebuilt. This is a very costly operation which makes scene partitioning trees not very suitable for continuous dynamics or unstructured motion.

Grid based approaches are a very general approach and support all types of motion. They do however have problems with scenes with a varying number of primitives per cell, known as the "teapot in the stadium problem". This can be overcome to a certain extent with hierarchical grids. Although grids have a very fast build time they are not that fast in accelerating ray tracing. The simplicity of their design makes them very suitable for streaming processors such as a GPU.

Bounding volume hierarchies can compete with scene partitioning trees in terms of performance. They also behave well on hierarchical and continuous dynamics motion. This is because they can be updated, and do not have to be rebuilt. The downside of this is that no objects can be added or deleted in the scene. Another downside is that we have to know what the structure of the scene is for an efficient bounding volume hierarchy because there is a large chance that the algorithm does not correctly find this structure. Having unstructured motion in the scene degrades the bounding volume hierarchy and is a severe impact on performance. Rebuilding the bounding volume hierarchy can be done between frames but is a severe impact on performance.

Overall scene partitioning trees seem to be the best solution for static objects. Although well build bounding volume hierarchies can compete with scene partitioning, scene partitioning trees are much cleaner and simpler because of their strict partitioning of the space. They can also be used for hierarchical motion but any other kind of motion requires very costly rebuilds.

For hierarchical motion and continuous dynamic motion bounding volume hierarchies perform best although this method breaks if there is too much unstructured motion.

For unstructured motion grid based approaches work best because they can very quickly be rebuilt, they are however far slower than bounding volume hierarchies for hierarchical and continuous dynamic motion and for static objects their performance is but a fraction of that of scene partitioning trees.

There is currently no scheme that handles all kinds of motion adequately. Because each type of spatial index structure has type(s) of motion on which it performs best it would be logical to try to combine those structures in one scheme. In such a scheme the application programmer would have to specify which kind of motion occurs where. For example a kd-tree could be used for static objects, a bounding volume hierarchy for hierarchical and continuous dynamic motion and a grid for unstructured motion. A ray would then need to be checked for collisions in all acceleration structures and the nearest intersection should be used. At the time of writing there was no research published on combining acceleration structures yet but because of the active research in this field we feel certain that this will happen in the near future.

Because there is not a scheme that handles all kinds of motion effectively yet, interactive ray tracing is not much of an alternative for dynamic scenes yet. Rasterization does not need a spatial index structure and handles all kinds of motions equally. Advanced applications do use spatial index structures for static content but with rasterization this can very easily be combined with moving objects. Interactive ray tracing still needs an effective scheme for dynamic scenes to be an alternative to rasterization for dynamic scenes.

5 Conclusion

We have seen in the second chapter that computer graphics applications use increasingly complex scenes and advanced visual effects and there is a demand for even more advanced effects . For these reasons computer graphics applications will cross the boundary where ray-tracing becomes more efficient than rasterization in the near future.

In chapter three we have seen that current hardware is suboptimal for ray-tracing. Optimal hardware for ray tracing would be combination of the generality of a CPU and the parallelism of a GPU. There is a trend for CPUs to become more like GPUs and vice versa. For this reason hardware will be increasingly suitable for ray tracing. It has also been shown that optimal hardware for ray tracing can already be made with current technology. For these reasons we believe that suitable hardware for ray tracing will be available before long.

In chapter four we have looked at how to keep spatial index structures updated to accelerate ray tracing. We have seen that spatial index structures respond differently to different types of motion. We have seen that each structure performs well on certain types of motion but that no structure performs adequately for all types of motion. A scheme that performs well on all kinds of motion is the biggest open topic in interactive ray tracing. Research to resolve this, by for example combining structures, is actively being pursued. Because of the large progress in this topic in the last years we speculate that this will be resolved before long.

Because there is currently no optimal hardware for ray tracing and no scheme to accelerate all kinds of motion adequately ray tracing is not much of an alternative to rasterization yet. However we have seen three developments that will ray tracing make a better alternative to rasterization:

- Graphics applications are crossing the boundary where ray-tracing becomes more efficient than rasterization
- Hardware is increasingly suitable for ray tracing
- Research on how to support interactive scenes is actively being pursued

When these three development are completed ray tracing will not only be an alternative to ray tracing but will perform **better** than rasterization. This is because ray tracing would be able to provide superior image quality, use new technology and be more efficient than rasterization. For these reasons we believe that ray-tracing will eventually completely replace rasterization.

Bibliography

1	WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive Rendering with Coherent Ray Tracing. Computer Graphics Forum(Proceedings of EUROGRAPHICS) 20, 3, 153–164.
2	Computer and Video Game industry http://en.wikipedia.org/wiki/Game_industry
3	PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002.Ray Tracing on Programmable Graphics Hardware. ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) 21, 3, 703-712.
4	BENTHIN, I., WALD, I., SCHERBAUM, M., FRIEDRICH, H. 2006. Ray Tracing on the CELL processor Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, (accepted for publication, minor revision pending, to appear)
5	SCHMITTLER, J., WALD, I., AND SLUSALLEK, P. 2002. Saar-COR – A Hardware Architecture for Ray Tracing. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 27–36.
6	WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) 2005. 24, 3 , 434 - 444
7	HAVRAN, V. , PRIKRYL, J., PURGATHOFER, W. Statistical Comparison of Ray-Shooting Efficiency Schemes, technical report TR-186-2-00-14, Vienna University of Technology, April 2000.
8	WALD, I., AND HAVRAN, V. 2006. On building good kd-trees for ray tracing, and on doing this in O(N log N). Tech. Rep. UUSCI-2006-009, SCI Institute, University of Utah.
9	REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic Acceleration Structures for Interactive Ray Tracing. In Proceedings of the Eurographics Workshop on Rendering, 299–306
10	WALD, I., BOULOS, S., AND SHIRLEY, P. 2006. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. ACM Transactions on Graphics (conditionally accepted, to appear).
11	LEXT, J., AND AKENINE-M OLLER, T. 2001. Towards rapid reconstruction for animated ray tracing. In Eurographics Short Presentations, 311-318.
12	WALD, I., BENTHIN, C., SLUSALLEK, P. A Simple and Practical Method for Interactive Ray Tracing of Dynamic Scenes. Technical Report 2002-04 Saarland University,
13	LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. 2006. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. Tech. Rep. 06-010, Department of Computer Science, University of North Carolina at Chapel Hill.

14	WALD, I., IZE, T., KENSER, T., KNOLL, T., PARKER, SG. Ray tracing animated scenes using coherent grid traversal - ACM Transactions on Graphics (TOG), 2006. 25,3, 485 - 493.
5	LEXT, J., ASSARSSON, U., AND M OLLER, T. BART: A Benchmark for Animated Ray Tracing. Tech. rep., Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, May. Available at http://www.ce.chalmers.se/BART/.
16	GUNTER, J., FRIEDRICH, H., WALD, I., SEIDEL, H., SLUSALLEK, P. 2006 . Ray Tracing Animated Scenes using Motion Decomposition. Computer Graphics Forum(Proceedings of Eurographics). 2006, 25, 3, 517-525
17	Binary space partitioning. http://en.wikipedia.org/wiki/Bsp_tree
18	FRIEDRICH, H., GUNTHER, J., DIETRICH, A., SCHERBAUM, M., SEIDEL, H., SLUSALLEK, P. 2006. Exploring the Use of Ray Tracing for Future Games. Proceedings of ACM SIGGRAPH Video Game Symposium 2006