The Advent of GPU Ray Tracers

Kroeze, JCW, North West University, Vanderbijlpark, South Africa, thejcwk@gmail.com
Jordaan, DB, North West University, Vanderbijlpark, South Africa, dawid.jordaan@nwu.ac.za
Pretorius, P, North West University, Vanderbijlpark, South Africa, phillip.pretorius@nwu.ac.za

Abstract
Ever since it's introduction in 1980 the ray tracing algorithm has been a tricky puzzle. On the one hand it offers photo-realistic rendering superior to the more common rasterization algorithm, but on the other it's much too slow to serve as a replacement.

Much of the rasterization algorithm's performance advantage lies in the use of special purpose hardware. This hardware, known as graphics cards, has evolved rapidly over the past decades, driven by demands for ever higher polygon counts. More recently, this pressure has lead to the introduction of programmable shader units to the graphics card architecture.

These units became ever more programmable as researchers realized their potential as general parallel processing units. Eventually this trend lead to the exploration of the graphics card architecture as a ray tracing platform.

This paper aims to explore the advent of this type of ray tracer and explain the developments that have been made in this field in order to see what the current state of the art is and where further research is required. It will also explore the defects that can be identified in the current literature and discuss how to address them.

Keywords
GPU, GPGPU, Ray Tracing, Computer Graphics.

Introduction
The ray tracing algorithm was first formulated by Turner Whitted in 1980 (Whitted, 1980). This algorithm traces the path of light rays based on classical ray optics. Each ray can reflect off the surface of an object, or refract through it (Whitted, 1980:344). By modelling the behaviour of each individual ray, a highly realistic rendering of the scene can be produced.

Unfortunately, the algorithm proved to be much too slow for use in interactive graphics programs (Whitted, 1980:349). This has led to the development of many techniques to speed-up the calculation of images by ray tracing.

Recently there has been a lot of interest in the execution of ray tracing algorithms on the graphics processing unit (GPU) present on current graphics cards (Horn, Sugerman, Houston & Hanrahan, 2007). Since these cards are usually built to parallel process huge volumes of data at interactive speeds, they may prove to be a good platform for the ray tracing algorithm, which is inherently very parallel.

Another attractive aspect of GPU based ray tracing is the fact that GPUs are very good at generating rasterized images of three dimensional scenes very quickly. This ability can be used to quickly determine the first hit location for a large collection of rays (Horn *et al*., 2007:169; Purcell, Buck, Mark & Hanrahan, 2002:268). Since this is a major part of the work done by a ray tracer, it should speed up computation significantly. The ray tracing algorithm can then be used for the parts it excels at: perfect specular reflection, refraction, shadows, caustics and the like. In addition, the graphics card can be used to perform basic and advanced shading operations (Horn *et al*., 2007:169), since all ray tracers require some form of shading, this capability makes graphics cards attractive platforms for ray tracing.

There is also evidence that GPUs are faster than central processing units (CPUs) for at least some

tasks (Buck, Foley, Horn, Sugerman, Fatahalian, Houston and Hanrahan, 2004:783). There is also the fact that graphics cards have advanced faster than CPUs in the past, since they can always incorporate more pipelines, while it is harder to add more transistors to a CPU (Purcell *et al.*, 2002:268). It is unclear that this argument still holds in the current day however, since the rise of multi-core CPUs has brought some measure of scalability to the CPU.

This paper will review the history of GPGPU ray tracing algorithms after which it will discuss some of the shortcomings discovered in the literature. Please note that this paper's goal is not to discuss the difference in performance between GPU and CPU ray tracers, or to draw any type of comparison between the two – such a comparison is difficult because there's very little data in the literature on the relative performance of GPU and CPU ray tracers. It would be outside the scope of this paper to attempt to reconstruct so many experiments in order to draw these comparisons. However, the authors have mentioned comparisons where they are available in the literature.

Early Predictions

Prior to the emergence of viable GPU architectures, simulation of the GPU architectures that would emerge in the future managed to predict many of their performance aspects. It was predicted that a GPU that was capable of branching would be faster than a GPU without it (Purcell *et al*., 2002:273). According to Purcell *et al*. (2002:273) this would be due in part to extra work and to the coherence that is lost when not using the looping algorithm that branching allows (Purcell *et al*. 2002:275), whereas Foley and Sugerman (2005:21) put the inefficiencies in a non-branching architecture down to the data that must be re-circulated for every ray. This was later confirmed and the performance gains from branching were estimated at a 25 times speed increase (Horn *et al*., 2007:170). This makes sense: loops in computer programs allow for data to be re-used repeatedly in the same context. Having to re-execute a program and re-calculate all the data that's common to each loop will slow the process. The coherence that is the foundation of many of today's performance enhancing technologies and algorithms also greatly benefit from a block of code that is obviously going to repeated. This advance in the design of programmable GPUs is therefore very important.

It was also predicted that secondary and shadow rays would be less cache friendly than the primary rays that spawned them (Purcell *et al*. 2002:276), this was later confirmed (Horn *et al*., 2007:170). Primary rays that correspond to nearby pixels tend to hit objects close to each other, which are usually close in memory. This also generates similar instruction sequences. Both of these situations are good for coherence, and therefore performance. But because secondary rays tend to scatter because of reflection and refraction, the coherence tends to fall off after the first hit. Naturally this is a problem on the CPU as well, but because GPUs are so parallel and based on the very idea of coherence, it is a bigger problem on the GPU (Horn *et al.*, 2007;170; Carr, Hall & Hart, 2002:38).

Acceleration data structures were first implemented on a simulated GPU architecture by Purcell *et al.* (2002; Horn *et al.*, 2005:168). This was a huge step forward, since acceleration structures have always been so important for performance. Their simulation was also the first GPU algorithm to make use of a uniform grid – although they lament the fact that it performs poorly on some scenes (Purcell *et al*. 2002:276). Interestingly, Purcell *et al*. (2002:276) proposed the use of the rasterizer on the graphics card to traverse a uniform grid acceleration structure. To the best of the authors' knowledge, this approach has not been implemented, but it sounds promising. It could be an interesting research topic to pursue.

The Stream Model

The previous section has discussed some of the advantages that might be realised with the use of a GPU ray tracer. While these advantages are attractive in theory, extracting them in practice has proven to be more difficult.

In part, this difficulty is due to the fact that graphics cards express their programmable units in terms of graphics concepts such as textures and shaders. This is not ideal for the design and implementation of a ray tracer, since these concepts do not map well to ray tracing. It makes more sense to view a GPU as a streaming processor in which data is modelled as streams with specific dimensions that

flow through a sequence of kernels (Purcell *et al*., 2002:270). Each kernel then performs operations on its input stream and produces an output steam that serves as input to the next kernel (Buck *et al*., 2004:778).

The stream model has several advantages: it encourages independent execution which increases parallelism, it forces kernels to do many calculations versus memory bandwidth utilized and it hides memory latency with the use of pre-fetching (Purcell *et al*., 2002:270).

In order to capture these advantages and ease the implementation of general algorithms on the GPU a programming environment such as Brook is important. Brook allows programmers to express their algorithms in terms of the streaming model (Buck *et al.,* 2004:777) and was implemented on the GPU and tested with a ray tracing algorithm as early as 2004 (Buck *et al*., 2004). Brook would prove to be influential in the early research on GPU ray tracing, as it was used by both Foley and Sugerman (2005:17) and Horn *et al*. (2007:167) for their implementations.

Brook has not seen widespread use in the most recent papers, this is likely due to the increasing ease of programming that recent GPUs offer. However, the realisation that a generic programming language is important likely eased the development of future GPU ray tracers. In the authors' experience, a programming environment that is close to the problem domain is usually very helpful.

Initial Hardware Implementations

To the extent of the authors' knowledge, the first use of graphics card hardware in ray tracing was the use of the cards' rasterization capabilities to speed up the calculation of eye rays' first hit with scene geometry (Carr *et al.*, 2002:38). This was the only part of the ray tracing process accelerated by the graphics card in their approach (Purcell *et al.*, 2002:277). This approach has the advantage that the CPU can be used for the tasks it is best suited for: complex algorithms and data structures and the GPU can be used for the parallel and repetitive tasks for which it was intended (Carr *et al*. 2002:41). Carr *et al*. (2002:41) achieved good results with this approach, but their ray tracer's performance was limited by the slow transfer rates between video card and CPU that was the case at the time. Given the recent advances in the technology bridging GPUs and CPUs in the PCI express specification, this approach could be revisited.

The first GPU ray tracing algorithm to make use of the *k*-d tree was described by Foley and Sugerman (2005; Horn *et al*., 2005:168). Due to memory limitations imposed by the GPU hardware the generic *k*-d tree algorithms had to be adapted to run without a stack (Foley & Sugerman, 2005:15). Typically, an optimized *k*-d tree will process the child of a node nearest to a ray first and place the further child on a stack (Horn *et al*., 2007:168). These stack operations can be eliminated by keeping track of the start and end points of a specific ray, and updating the start point to equal the start of the next child's extents when the algorithm finishes with a leaf node (Foley & Sugerman, 2005:16). When the algorithm then reaches a leaf node with no intersections, it can simply restart from the root and quickly find the node it should search next – this technique is called *kd-restart* (Foley & Sugerman, 2005:16). By further manipulating these start and end points, the algorithm can determine the parent of the next node to be searched, eliminating a couple of traversal steps (Foley & Sugerman, 2005:17) – this optimization is termed *kd-backtrack*. There is one major problem with *kd-backtrack* however, as this strategy requires 256 extra bits of storage (Foley & Sugerman, 2005:18). This cost would prove too large for Horn *et al.* (2007:168), who were worried about the effects it would have on packetization and bandwidth. All in all, the loss of a stack only increased the cost of *k*-d tree traversal by a linear factor (Foley & Sugerman, 2005:20). While this is impressive, it did set the algorithm itself back when compared against the CPU version, which is unfortunate. This is a problem with the GPGPU approach – the GPU is not as flexible as the CPU and its memory is generally very limited.

A year later, Carr *et al*. (2006) developed a method based on the idea of storing an acceleration structure in a MIP map texture as a geometry image. Their method was able to ray trace dynamic scenes and was competitive with other techniques at the time (Carr *et al.*, 2006:207). Unfortunately, they could only ray trace scenes containing a single mesh with no sharp edges (Carr *et al.*, 2006:207,

Popov *et al*., 2007). This is probably why their method has fallen by the wayside, despite having competitive performance characteristics for the techniques of the time. It is also likely that the community's familiarity with *k*-d trees pushed research in that direction, rather than into novel approaches.

Around the same time Huang *et al*. (2006) developed the traversal field method. This method constructs a series of ray relays at the faces of the bounding boxes that enclose objects (Huang *et al*., 2006:65). These relays then sample all the possible incoming directions of rays and associate them with the triangles they would intersect (Huang *et al*., 2006:65). While their method had a good performance profile when measured against the efforts of Carr *et al*. (2006), it required user intervention (Huang *et al*., 2006:67) and was subject to aliasing effects caused by the sampling nature of the algorithm (Huang *et al*., 2006:69). The algorithm also had difficulty dealing with convex objects (Huang *et al*., 2006:67) and experienced severe performance and memory footprint penalties when the amount of triangles in a scene reached 2^{16} (Huang et al., 2006:70). These difficulties are likely the reason that researchers didn't explore this algorithm further. The requirement for user intervention alone would make their algorithm unsuitable for use in an interactive program, and the convexity requirement and limitation on the amount of triangles would have been a big step backward.

The performance figures comparing GPU ray tracing to CPU ray tracing were disappointing at this point in history. Foley and Sugerman (2005:21) report that their implementation is an order of magnitude slower than a CPU implementation. This large discrepancy was reportedly due to data re-circulation (Foley & Sugerman, 2005:21) – a problem that was later solved by the use of the new looping features on more modern cards (Horn *et al.*, 2007:172). Zhou *et al.* (2008:126:2) summarily states that the algorithms described in Carr *et al*. (2002), Carr *et al*. (2006) Purcell *et al.* (2002), and Foley and Sugerman (2005) are slower than heavily optimized CPU ray tracers. However, Buck *et al.* (2004:783) claim significant improvement over a fast CPU implementation on graphics cards with lots of memory bandwidth, but their figures compare ray-triangle intersection per second, rather than the more common and appropriate frames per second. It is uncertain whether their algorithm outperformed the CPU algorithm in terms of animation speed as their focus wasn't on ray tracing, *per se*.

Unfortunately, at this point the potential benefits of a GPU based ray tracer had not been realized yet. It would take more research and hardware development to reach acceptable speeds.

Advanced Implementations

The case for GPU ray tracing became much stronger in 2007 with the introduction of at least three algorithms that outperformed CPU ray tracers – Horn *et al*. (2007), Chen and Liu (2007) and Popov *et al*. (2007). Horn *et al*. (2007:171) achieved nearly double the performance for a single Opteron 2.4 GHz CPU, which is encouraging. Unfortunately there are no figures comparing the performance of their algorithm to recent CPUs.

This algorithm consists mainly of refinements to the approach suggested by Foley and Sugerman (2005). These refinements are called *push-down* and *short-stack* (Horn *et al*., 2007:167). The focus of these algorithms is to exploit the additional functionality that had been introduced into the programmable units on the graphics cards from 2005 till 2007 – e.g. looping and branching (Horn *et al*., 2007:167). The *short-stack* optimization provided the majority of the performance improvement – reducing the count of visited nodes by 48 – 52% over the *k*-d tree with push-down, which had already reduced counts by 3 – 22% (Horn *et al*., 2007:170). This is quite impressive, but the overall performance improvement they achieved is largely attributable to the extra capabilities of the hardware, and not to new insights into the nature of the ray tracing algorithm on GPUs.

These optimizations together with improvements in the hardware's computational power resulted in more than a 25 times performance increase over the work done by Foley and Sugerman (Horn *et al.*, 2007:170). Most of this performance improvement is due to the introduction of looping into the algorithm (this was previously impossible due to limitations present in the platform), which

eliminated the data recirculation problems encountered by Foley and Sugerman (Horn *et al.* 2007:170).

That said, the hardware still proved to be problematic. The graphics card that was used by Horn *et al.* provided four wide SIMD instructions, but only two scalar operations could be performed at once (Horn *et al*., 2007:170), which slowed down the algorithm when compared with processors that are fully four wide. This was a problem with the hardware available at the time, so it should not be paid much attention today.

The figures for the packetization introduced by Horn *et al*. (2007) are less rosy. While there is no real penalty or improvement when using ray packets that bounce only once on the GPU, packetization becomes more problematic when more bounces are added (Horn *et al.*, 2007:170). This is thought to be due to incoherent branching, which is a major problem on the GPU architecture due to its nature (Horn *et al*., 2007:171). Because of this problem and the limited register memory that is available on current graphics cards, the use of large ray packets is unfortunately unlikely (Horn *et al*., 2007:172). A modification to the *k*-d tree that results in larger leaves might alleviate this problem in the future (Horn *et al*., 2007:172). This would again be a very interesting topic to study for future research.

Chen and Liu (2007:1050) report that they were able to get a 62% - 157% performance boost over a pure CPU solution from just using the graphics hardware to speed up the first hit calculation, even when taking into account the overhead of transferring data between the graphics card and CPU. This is very encouraging, and implies that another hybrid approach might be the best way to go for the ray tracing community in general.

At the same time Popov *et al*. (2007) developed an extension to *k*-d trees that significantly reduces the amount of work that is done traversing the tree. In their algorithm, the *k*-d tree maintains "ropes" at its leaf nodes (Popov *et al.,* 2007). These ropes link a leaf node's bounding box faces to the node that is on the other side of that face (Popov *et al.,* 2007). This has a number of advantages: first, the resulting algorithm does not require a stack, which saves on memory bandwidth and second, it can reduce "down"-traversals by 5/6 over the method described by Foley and Sugerman (2005). Since the algorithm requires no stack, it could potentially be used as an improvement on the *kd-restart, kd-backtrack, short-stack* and *push-down* algorithms mentioned earlier.

Popov *et al*. (2007) state that their GPU implementation of this algorithm outperforms the CPU implementation. Their figures also indicate that their algorithm beats the performance attained by the OpenRT system that is designed for CPUs (Popov *et al*., 2007). This is certainly encouraging, but a comparison with other heavily optimized ray tracers available at the time would have been welcome.

Curiously, the method described by Popov *et al*. (2007) doesn't seem to have penetrated the ray tracing research community, as their research is not incorporated into any later papers to the authors' knowledge. It seems like a very effective scheme, however, and more investigation should be done.

The difference in hardware and the algorithms used between the different papers in the literature muddy the waters significantly. There is a need for a standardized platform to compare different approaches on the same hardware.

Current State of the Art

Previous techniques did not fully exploit the highly parallel nature of modern GPUs. Zhou *et al.* (2008) describe a real-time *k*-d tree construction algorithm that is tailored to this type of architecture. The algorithm builds the tree in breadth-first order, instead of depth-first (Zhou *et al*., 2008:126:1). This leads to a large number of threads being spawned, taking advantage of the GPUs high parallelism (Zhou *et al*., 2008:126:1). In addition, the algorithm iterates over primitives for the top levels of the trees, making sure that the GPU is fully utilized for the complete run of the algorithm (Zhou *et al*., 2008:126:1). This type of refinement seems characteristic of recent research on GPU ray tracing. Earlier worked focused more on adapting CPU-based techniques for the GPU. Researchers are now working out the peculiarities of the platform and optimizing for them.

These techniques are enough to bring their ray tracer up to speed with CPU techniques, as their results trump those of two recently published CPU-based results (Zhou *et al*., 2008:126:7). However, the performance benefit for GPU over CPU ray tracers seems to be anything but clear cut. Even this algorithm (which is one of the fastest at the moment) is inferior to a CPU algorithm running on eight cores (Zhou *et al*., 2008:126:7) for at least one scene.

Using a simulator that makes very favourable assumptions about the memory bandwidth available on modern GPUs, it is possible to determine that current techniques are limited by the work distribution mechanism on modern graphics cards (Aila & Laine, 2009:146-149), rather than the memory bandwidth available on these cards as is commonly thought.

Aila and Laine (2009:147) argue that the work distribution problem is caused by the fact that each ray is usually assigned as a packet of work to each of the pipelines on a GPU. However, GPUs execute the same instruction on each pipeline at the same time (SIMD). If one ray takes significantly longer to compute than another, then most of the pipelines will remain idle (Aila & Laine, 2009:147).

It is therefore possible that Zhou *et al*.'s algorithm is only utilizing a fraction of the graphics card's power. If this is the case, then GPU ray tracing performance could far exceed the performance of CPU algorithms in the near future. More research should be done to implement Zhou *et al*.'s algorithm using the work distribution method described by Aila and Laine (2009).

It is entirely possible, however, that Aila and Laine's findings (2009) are not applicable to the algorithm introduced by Zhou *et al*. (2008). Aila and Laine's technique described above makes many assumptions and therefore can only provide approximate data (Aila & Laine, 2009:146). Since the memory architecture of the simulator used by Aila and Laine (2009) is so optimistic, there is room for error in their conclusions.

That said, the results of the optimizations suggested by Aila and Laine (2009) are compelling. The situation described above can easily be solved by using persistent threads and utilizing speculative traversal (Aila & Laine, 2009:147-149). These improvements bring the performance of GPU ray tracers to within 10% of the estimated upper bound on performance as determined by Aila and Laine (2009:146).

Ironically, these modifications allow the GPU algorithms to reach an efficiency level where memory bandwidth may indeed become a problem (Aila & Laine, 2009:149). Future advances in GPU memory bandwidth will therefore be very beneficial to ray tracing.

Kalojanov and Slusallek (2009) also developed a highly parallel construction algorithm, but for uniform grids. They reduce the problem of constructing a grid to a sorting problem, that is easily solved by an implementation of the radix sort algorithm present in the SDK they were using (Kalojanov & Slusallek, 2009:24). They store their acceleration structure in texture memory on the graphics card in order to make use of the speedy texture cache (Kalojanov & Slusallek, 2009:26). While their construction algorithm is very quick, the results from the ray tracer is not encouraging. Kalojanov and Slusallek (2009:26) state that their results are inferior to those already seen on the CPU. However, their ray tracer was not as sophisticated and optimized as the ones they were comparing against. Their true contribution is the fast construction algorithm, which looks very promising. Kalojanov and Slusallek's approach may be useful for dynamic scenes were the acceleration structure must be rebuilt quickly – as their approach can completely hide the computation done to upload new geometry to the GPU (Kalojanov & Slusallek, 2009:26). However, the memory problems they encountered (Kalojanov & Slusallek, 2009:26), together with the slow ray tracing speed of their approach will likely mean that their approach will not be used for complex scenes.

Most of these approaches have looked at ways to improve the amount of rays that can be traced per second. However, there are other factors impacting the performance of a GPU ray tracer that may

become stumbling blocks in the future. Further improvements to the GPU ray tracing algorithm may include strategies for speeding up the rasterization step, early termination for shadow rays and using the GPU's advanced shading capabilities (Horn *et al.*, 2007:172). Research into these ideas may yield surprising gains.

Summary

The preceding sections of this paper have looked at the development of GPU ray tracing from the perspective of various improvements and inventions. This section will take a high-level view to illustrate the flaws inherent in the current research paradigm.

	Carr *et al.* (2002)	**Purcell *et al.* (2002)**	**Buck *et al.* (2004)**	**Foley & Sugerman (2005)**	**Carr *et al.* (2006)**	**Huang *et al.* (2006)**
Acceleration Structure Type	Octree & 5-D ray tree.	Uniform grid.	Uniform grid.[i]	K-D tree.	Bounding volume hierarchy.	Traversal field.
Focus of Research	Performing ray-triangle intersection on the GPU.	GPU simulation.	Measuring the performance of the Brook programming environment.	Application of the k-d tree acceleration structure to GPU ray tracing.	Storage of acceleration structure in texture memory.	Development of the traversal field structure and ray relays.
Interactive Rendering Speeds Achieved	No.	No.	No.	No.	No.	No.
Approximate FPS	N/A.[ii]	N/A.[iii]	N/A[iv]	~1[v]	N/A.[vi]	2 – 10.[vii]

Table 1: Comparison of some GPU ray tracers.

Table 1 and table 2 summarize the approaches used by each of the papers discussed earlier. Almost every study introduces its own take on performance enhancement, ignoring many of the advances, observations and improvements that were made previously – promising results from a previous study are rarely developed further. It is possible that incorporating the ideas from previous studies could enhance the insights in future studies and make the algorithms developed there even faster.

	Horn *et al.* (2007)	**Chen & Liu (2007)**	**Popov *et al.* (2007)**	**Zhou *et al.* (2008)**	**Aila & Lane (2009)**	**Kalojanov & Slusallek (2009)**
Data Structure	K-D tree.	Bounding volume hierarchy.	K-D tree with "ropes".	K-D tree.	BVH.	Uniform grid.
Focus of Research	Application of Foley and Sugerman's work (2005) to a branching GPU architecture.	Use of the hardware Z-buffer algorithm to speed up first hit calculations.	Development and performance analysis of the improved K-D tree structure.	K-D tree construction improvements.	Work distribution improvements.	Fast construction of uniform grid.

Interactive Rendering Speeds Achieved	Yes.	Yes.	Yes.	Yes.	Yes.	Yes.
Approx. FPS	N/A.[viii]	~10 depending on scene[ix]	4.0 – 12.7[x]	4.8 – 32.0[xi]	N/A[xii]	3.5 – 7.7[xiii]

Table 2: Further comparisons of GPU ray tracing techniques.

This is not the only problem, however. There is also a great deal of variation in the experimental methods used by each paper. No agreement has been reached in the GPU ray tracing community regarding an acceptable standard performance metric or a set of representative and common testing scenes. This will be illustrated by tables 3 and 4.

	Carr *et al.* (2002:43)	**Purcell *et al.* (2002:275)**	**Buck *et al.* (2004:783)**	**Foley & Sugerman (2005:19)**	**Carr *et al.* (2006:207)**	**Huang *et al.* (2006:72)**
Performance Metric	Rays / second.	SIMD efficiency, traversal steps and intersections.	Ray / triangle intersections per second.	Elapsed milli-seconds and various traversal counts.	Elapsed milli-seconds.	Rays / second and inter-sections / ray.
	Horn *et al.* (2007:170-171)	**Chen & Liu (2007:1049-1050)**	**Popov *et al.* (2007)**	**Zhou *et al.* (2008:126:6-7)**	**Aila & Lane (2009:146)**	**Kalojanov & Slusallek (2009:26)**
Performance Metric	Frames per second and millions of rays / second.	Elapsed seconds and percentage speed-up.	K-d tree statistics, traversal steps and frames per second.	Elapsed seconds and frames per second and speed-up factor.	SIMD efficiency, millions of rays / second and percentage of simulated performance.	Frames per second and milliseconds.

Table 3: Performance metrics used by each paper.

Rays per second, elapsed time and frames per second are used as metrics several times, but there is still very little unification between papers. This means that it is very difficult to compare the performance of one ray tracer to another.

It is also unclear which of these measurements is the best, and if any of them are suited to the comparison of experimental results. There is a need for research to be conducted to investigate which of these measurements describes the performance of a ray tracer in the most precise manner. Such a metric will have to eliminate as many variables as possible.

	Carr *et al.* (2002:43)	**Purcell *et al.* (2002:275)**	**Buck *et al.* (2004:780)**	**Foley & Sugerman (2005:19)**	**Carr *et al.* (2006:207)**	**Huang *et al.* (2006:70)**

Scenes	"Teapot room", "office" and "soda hall".	"Soda hall", "forest" and "bunny".	"Glassner"[xiv]	"Robots", "kitchen", "Cornell box" and "Stanford bunny".	"Stanford bunny" and "Mult."	"Desk", "cube", "teapot", "bear", "venus", "simplified bunny", "approxi-mate bunny", "teapot house" and "bunny couple".
	Horn *et al.* (2007:170-171)	**Chen & Liu (2007:1049-1050)**	**Popov *et al.* (2007)**	**Zhou *et al.* (2008:126:6)**	**Aila & Lane (2009:146)**	**Kalojanov & Slusallek (2009:26)**
Scenes	"Cornell box", "kitchen", "robots" and "conference".	"Bunny", "dragon" and "easter".	"Shirley6", "bunny", "forest" and "conference".	"Toys", "museum", "robots", "kitchen", "fairy forest" and "dragon".	"Conference", "fairy" and "Sibenik".	"Thai statue", "soda hall", "conference", "dragon", "fairy forest", "sponza", "ruins".

Table 4: Scenes used by each paper.

Like the performance metrics, there is a wide variety of scenes in use by the ray tracing community. While several scenes are used repeatedly, there is still too little correlation to make comparisons easily.

If we are to obtain meaningful experimental results that are comparable, then all variables must be controlled for. Certainly the use of certain scenes is one such variable. Haines (1987) and Lext, Assarsson and Möller (2001) have made some progress towards this ideal, but their scenes are seldom used: as shown by table 4 – only "kitchen" and "robots" from Lext *et al.*'s library is used 3 times.

The viewpoint from which a scene is rendered is also important. Most of the papers surveyed did not specify this viewpoint, even though it is an important variable. Some objects may not even be visible from a particular viewpoint, which could heavily influence the performance of certain algorithms. More care should be taken in the future with regards to stating the particular viewpoint used in an experiment.

	Carr *et al.* (2002:44)	**Purcell *et al.* (2002:273-276)**	**Buck *et al.* (2004:782)**	**Foley & Sugerman (2005:19)**	**Carr *et al.* (2006:206)**	**Huang *et al.* (2006:70)**
GPU	Radeon 8500 / GeForce 3 / GeForce 4 Ti4600	Not stated.	Radeon X800 XT Platinum / GeForce 6800 (Pre-release)	256 MB ATI X800 XT PE	GeForce 7800 GTX (430 MHz clock, 1.2GHz memory clock)	256 MB NVIDIA 6800GT
CPU	Not stated.	Not stated.	3 GHz Pentium 4 (875P Chipset)	Not stated.	2.2 GHz Athlon 3500+	2 x 3.2 GHz Pentium 4
Memory	Not stated.	Not stated.	Not stated.	Not stated.	Not stated.	2 GB
	Horn *et al.* (2007:170)	**Chen & Liu (2007:1049)**	**Popov *et al.* (2007)**	**Zhou *et al.* (2008:126:6)**	**Aila & Lane (2009:145)**	**Kalojanov & Slusallek (2009:25)**
GPU	512 MB Radeon X1900 XTX (650 Mhz clock & 750 Mhz memory clock)	Radeon X300SE	GeForce 8800 GTX	768 MB GeForce 8800 ULTRA	GeForce 285 GTX	1 GB GeForce 280 GTX
CPU	2 x 2.4 GHz Core2 Duo	1.8 GHz Athlon64 3000+	2.6 GHz Opteron	3.7 GHz Xeon	Not stated.	4 x 2.66 GHz Core2 Quad
Memory	Not stated.	Not stated.	Not stated.	Not stated.	Not stated.	Not stated

Table 5: Hardware used by different papers.

Table 5 illustrates the wide variety of hardware used to test the performance of the various algorithms discussed in the papers above. The great difference between the performance of the various components identified obscures the differences between the performance of the algorithms discussed.

Ideally, hardware would not be a variable when comparing different algorithms. It could be eliminated by running each algorithm on the same hardware, or by some other method. Unfortunately, it is difficult to tell which algorithms are superior with the current approach.

Conclusion

This paper has presented the development of GPU ray tracing algorithms from their inception to the current state of the art.

Great strides have been made towards a viable real-time ray tracing algorithm on the GPU, but there is considerable confusion in the existing literature. The experimental set up for most of the papers that have been reviewed here is ad-hoc. Two different algorithms are sometimes compared by their performance on completely different hardware platforms. This is unfortunately not a fair comparison, and may distort overt or subtle differences in the performance of the various algorithms being discussed.

Advancing hardware is also an issue. It is difficult to compare experiments in this field because the algorithms are so heavily dependent on the newest technology. This is especially troublesome for

comparing papers that were published several years apart, since computer hardware advances at such an incredible pace.

All of these issues make comparisons between algorithms very difficult – even for papers that were published only in the last three years.

If the GPU ray tracing community is to learn which algorithms are effective, then it must find a way to compare the results from different studies in a fair way. Currently, the literature lacks a methodology that is capable of achieving this.

Such a methodology will need to find performance metrics that are independent from the underlying hardware and the properties of specific scenes. It will need to be widely acceptable and must be easy to use so that it will be used consistently.

It is the author's belief that research into such a methodology will benefit the GPU ray tracing community and may lead to great advances in the field.

Another problem in the literature is that many of the recent papers do not incorporate the ideas and optimizations from previous ones. It is likely that a much faster GPU ray tracer can be constructed by using the core ideas from several algorithms at once. A good example is the load balancing improvements made by Aila and Laine (2009) and the *k*-d tree improvements made by Zhou *et al.* (2008). Superficially, at least, it seems that these insights could be combined to yield a very high performing GPU ray tracer indeed.

As such, there is also room for significant research geared towards the integration of compatible ideas from the existing GPU ray tracing literature.

---

i Buck *et al*. (2004:782) state that they based their ray tracer on Purcell *et al*.'s work, therefore it is assumed that they used the same acceleration structure.

ii FPS is not stated, but the ray tracer achieved speeds of 100 000 – 200 000 rays per second which far exceeded the CPU ray tracers available at the time.

iii The research does not include any timing information.

iv The research contains no timing information, but states that between 45 and 186 ray-triangle intersections were performed per second (Buck *et al*., 2004:783).

v There is no data about FPS in the research *per se*, but the ray tracer described achieved rendering speeds of ~950 ms on the most complex scene rendered.

vi The research does not include any data on frames-per-second achieved, but states that an image was rendered at 1272 x 815 in approximately half a minute.

vii While the research does not include any data on FPS, it states that the ray tracer involved could compute an image in ~100 – 450 ms for one of the scenes. However, this data is only for eye rays which makes it an ineffective measure.

viii The research claims interactive rendering rates and a sustained rate of 15 million rays per second, but makes no mention of any timing information (Horn *et al.,* 2007).

ix There's no timing information in the research, but it does briefly state a computation time of 115ms on the Stanford bunny scene (Chen & Liu, 2007).

x These figures are for the ray tracer running on four different scenes with secondary rays and packet tracing (Popov *et al*., 2007).

xi Four dynamic scenes at 1024 x 1024 resolution (Zhou *et al.*, 2008).

xii The research reported 20-40 million rays per second presumably with secondary rays (Aila & Lane, 2009:149).

xiii This measurement is only for the generation of eye rays (Kalojanov & Slusallek, 2009:26).

xiv It is unclear whether any other scenes were used. However, this scene name is mentioned on page 780 and the performance graphs suggest that only one scene was used.

References

Aila, T and Laine, S. (2009). 'Understanding the efficiency of ray traversal on GPUs'. Proceedings of the Conference on High Performance Graphics 2009, ISBN: 978-1-60558-603-8, 1-3 August 2009, New Orleans, LA, 145-149.

Buck, I., Foley, T., Horn, D. Sugerman, J., Fatahalian, K., Houston, M. and Hanrahan, P. (2004). 'Brook for GPUs: stream computing on graphics hardware'. ACM SIGGRAPH 2004 Papers, 8-12 August 2004, Los Angeles, CA, 777-786.

Carr, N.A., Hall, J.D. and Hart, J.C. (2002). 'The ray engine'. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, ISBN: 1-58113-580-7, 1-2 September 2002, Saarbrucken, Germany, 37-46.

Carr, N.A., Hoberock, J., Crane, K. and Hart, J.C. (2006). 'Fast GPU ray tracing of dynamic meshes using geometry images'. Proceedings of Graphics Interface 2006, ISBN: 978-1-56881-308-0, 7-9 June 2006, Québec, Canada, 203-209.

Chen, C.C. and Liu, D.S.M. (2007). 'Use of hardware z-buffered rasterization to accelerate ray tracing'. Proceedings of The 2007 ACM Symposium on Applied Computing, ISBN: 1-59593-480-4, 11-15 March 2007, Seoul, Korea, 1046-1050.

Foley, T. and Sugerman, J. (2005). 'Kd-tree acceleration structures for a GPU raytracer'. Proceedings of the ACM SIGGRAPH / EUROGRAPHICS Conference on Graphics Hardware, ISBN: 1-59593-086-8, 30-31 July 2005, Los Angeles, CA, 15-22.

Haines, E. (1987). 'A proposal for standard graphics environments'. *IEEE Computer Graphics and Applications*, 7(11), 3-5.

Horn, D.R., Sugerman, J., Houston, M. and Hanrahan, P. (2007). 'Interactive k-D GPU ray tracing'. Proceedings of the 2007 Symposium on Interactive Ray Tracing, ISBN: 978-1-4244-1629-5, 10-12 September 2007, Ulm, Germany, 167-174.

Huang, P., Wang, W., Yang, G. and Wu, E. (2006). 'Traversal fields for ray tracing dynamic scenes'. Proceedings of The ACM Symposium On Virtual Reality Software And Technology, ISBN: 1-59593-321-2, Limassol, Cyprus, 65-74.

Kalojanov, J. and Slusallek, P. (2009). 'A parallel algorithm for construction of uniform grids'. Proceedings of the Conference on High Performance Graphics 2009, ISBN: 978-1-60558-603-8, 1-3 August 2009, New Orleans, LA, 23-28.

Lext, J., Assarsson, U. and Möller, T. (2001). 'A benchmark for animated ray tracing'. *IEEE Computer Graphics and Applications*, 21(2), 22-31.

Popov, S., Günther, J., Seidel, H.P. and Slusallek, P. (2007). 'Stackless KD-tree traversal for high performance GPU ray tracing'. *Computer Graphics Forum,* 26(3), 415-424.

Purcell, T.J., Buck, I., Mark, W.R. and Hanrahan, P. (2002). 'Ray tracing on programmable graphics hardware'. *ACM Transactions on Graphics*, 21(3), 268-277.

Whitted, T. (1980). 'An improved illumination model for shaded display'. *Communications of the ACM*, 23(6), 343-349.

Zhou, K., Hou, Q., Wang, R. and Guo, B. (2008). 'Real-time KD-tree construction on graphics hardware'. ACM SIGGRAPH Asia 2008 Papers, ISSN:0730-0301, 10-13 December 2008, Singapore, Article #126.