Ray Tracing on Graphics Hardware

Toshiya Hachisuka*
University of California, San Diego

Abstract

Ray tracing is one of the important elements in photo-realistic image synthesis. Since ray tracing is computationally expensive, a large body of research has been devoted to improve the performance of ray tracing. One of the recent developments on efficient ray tracing is the implementation on graphics hardware. Similar to general purpose CPUs, recent graphics hardware can perform various kinds of computations other than graphics computations. One notable difference between CPUs and graphics hardware is that graphics hardware is designed to exploit a larger degree of parallelism than CPUs. For example, graphics hardware usually processes several thousands of independent pixels (compare it to a few independent tasks on multi-core CPUs). Therefore, the key is to use algorithms that are suitable for parallelization in order to harness the power of graphics hardware. Although the parallel nature of ray tracing seems to be well suited for graphics hardware, there are several issues that need to be solved in order to implement efficient ray tracing on graphics hardware. This paper surveys several algorithms for ray tracing on graphics hardware. First, we provide an overview of ray tracing and graphics hardware. We then classify several algorithms based on their approaches. We also analyze the bottlenecks of current approaches and present possible solutions. Finally, we discuss future work to conclude the survey.

1 Introduction

In order to compute photo-realistic images, we need to accurately simulate all light-surface interactions. The computation of such interactions is called *light transport*, or *global illumination*, in computer graphics. simulating light transport has been an active research area in computer graphics in past decades (refer to Dutre et al. [2006]). A popular method for solving light transport is *ray tracing*. The basic procedure of ray tracing is to compute intersections between rays (which represent light rays) and objects (e.g., triangles). Reflections/refractions of light are simulated by generating a new set of rays based on material properties of objects. Figure 1 shows examples of global illumination rendering.

The problem of using ray tracing for solving global illumination is that it is computationally expensive. For example, generating a single image using ray tracing can take a few hours. As a result, most of interactive computer graphics applications, such as video games, do not use ray tracing because it is too slow for the interactive use. Instead, these applications often use *rasterization*. Rasterization generates images by projecting objects on the screen, and it is usually faster than ray tracing due to hardware acceleration. However, rasterization is not well-suited for solving global illumination. In order to meet the demand of photo-realistic rendering, using ray tracing in interactive applications as well is ideal. Therefore, there is a large body of work to improve the performance of ray tracing.

One of the recent developments on ray tracing is its implementation on graphics hardware. Graphics hardware used to have the fixed functionalities which are mainly for rasterization. However, the recent emergence of *programmable graphics hardware* has enabled us to perform not only interactive rendering using rasterization, but also general computations on graphics hardware as well. Since graphics hardware is fundamentally designed to exploit parallelism over all pixels or triangle vertices, problems with large degrees of parallelism can be efficiently processed on graphics hardware. For such problems, the speedup is often a few orders of magnitude from CPU implementations (refer to Owens et al. [2007] for example). In this paper, we focus on how ray tracing can be implemented on graphics hardware.

Since ray tracing is known as *embarrassingly parallel* (i.e., every ray is independent and can be easily processed in parallel), implementing ray tracing on graphics hardware seems to be easy. However, because of architectural differences between CPUs and graphics hardware, there are several issues that need to be solved to implement efficient ray tracing on graphics hardware. This paper surveys several different ray tracing algorithms on graphics hardware and discusses the issues involved efficient ray tracing on graphics hardware. We first provide an overview of ray tracing and general purpose computation on graphics hardware. Later, we describe several methods by classifying them based on their overall algorithms. We then compare all the methods and analyze the bottlenecks of the current approaches. Finally, we conclude by stating future work.

Figure 1: *Example images with global illumination. Ray tracing simulates complex light-surface interactions, such as the shiny reflections on Buddha model (left), soft shadow, and the refractions from the teapot (right).*

2 Overview

2.1 Ray Tracing

The goal of ray tracing is to compute intersections between rays and objects. Since ray tracing often uses only triangles as its geometric representation, we focus on ray tracing triangles in this survey. The main obstacle for efficient ray tracing is that the number of rays and triangles can be extremely large. For example, using a resolution of *1080p/i* (HDTV) requires a few million pixels to be processed. Since each pixel often uses several hundreds of rays in order to solve global illumination, the total number of rays could be a few hundred million. Similarly, a few million triangles are often used to achieve sufficient geometric complexity. For instance, the Buddha model in Figure 1 has roughly one million triangles. If we compute all of the ray-triangle intersections, it requires about 10^{14} ray-triangle intersection computations. Unfortunately, this brute force computation is too expensive. The current fastest ray-triangle intersection method can only achieve a few hundred million ray-triangle intersections (10^8) per second [Kensler and Shirley 2006]. In order

*e-mail: thachisu@cs.ucsd.edu

to implement efficient ray tracing, we need to decrease the number of redundant ray-triangle intersections.

In ray tracing, such reduction of redundant ray-triangle intersections is achieved by *acceleration data structures*. Acceleration data structures utilize the fact that there is spatial sparsity in both rays and triangles. In acceleration data structures, rays or triangles are clustered by a spatial subdivision scheme in the precomputation step. Ray tracing queries the precomputed acceleration data structure to cull redundant ray-triangle intersections by utilizing the spatial sparsities of rays and triangles. Querying an acceleration data structure is called *ray traversal*. The actual algorithm of ray traversal depends on the types of acceleration data structure.

Recent development of interactive ray tracing on CPUs [Wald et al. 2001; Reshetov et al. 2005; Wald et al. 2006] has shown that an efficient acceleration data structure and efficient ray traversal are the main factors that achieve significant performance improvement of ray tracing. Although the performance of ray-triangle intersection affects the overall performance of ray tracing, the bottleneck is ray traversal in current ray tracing algorithms [Wald et al. 2001]. This is mainly because the number of ray-triangle intersections is significantly decreased by acceleration data structures. For example, the efficient usage of an acceleration data structure performs only 2 to 3 ray-triangle intersections per ray on average [Reshetov et al. 2005].

2.2 Programmable Graphics Hardware

Recently, graphics hardware became programmable in order to accommodate the increasing demand of complex visual effects in video games. Usually, those visual effects involve *shading*, which determines the appearance of objects based on illumination setting and material properties. The shading process has been one of the fixed functionalities in graphics hardware. However, in programmable graphics hardware, programmers can write an arbitrary shading code to perform various computations beyond its fixed functionalities. We provide an overview of two different approaches to using the programmable functionalities on graphics hardware.

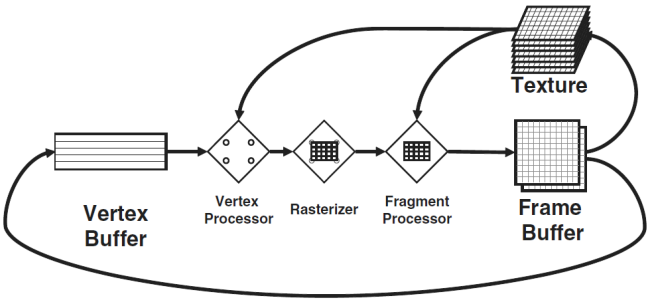

Figure 2: *Graphics hardware shader pipeline [Owens et al. 2007]. Both vertex processor and fragment processor are programmable. The vertex buffer contains vertex data (which describes objects) and frame buffer contains pixels data (aka. fragments) which are displayed to the screen. Textures contain pixels that are not displayed to the screen, which are often used as additional data for computations.*

The first approach is to use a programmable shading language in the shader pipeline on graphics hardware. In the shader pipeline, there are two programmable parts called the *vertex program* and *fragment program*. The vertex program processes vertex data (e.g., positions, texture coordinates, and normals), and the resulting vertex data is converted into fragments using rasterization. Figure 2 summarizes the shader pipeline. For example, perspective projection of triangles onto the screen will be written as a vertex program. The fragment program processes fragments that determine the pixel colors for display. For example, shading computation is often done in a fragment program.

Although programmable shading languages are originally designed to perform rendering tasks, they can be used to perform general parallel computations very efficiently. This is because the computation of each element (vertex or pixel) is performed in parallel with arbitrary vertex/fragment programs on graphics hardware. For example, NVIDIA GeForce 8800 GTX is capable of running 128 threads in parallel to process vertices/pixels. Those units are used for both vertex/fragment processes and capable of performing arbitrary computation specified by shader programs. Because graphics hardware can be considered as cost-effective vector processor, general purpose computation on graphics hardware has been an active area of research in computer graphics for the last few years (refer to [Owens et al. 2007] for survey).

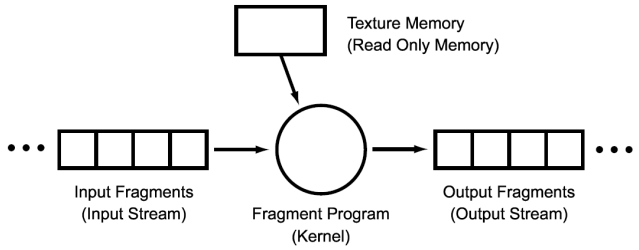

Figure 3: *Mapping of graphics pipeline onto streaming computation [Purcell 2004]. The words in parentheses show the corresponding terminologies for streaming computation. The kernel performs the same computation on all the elements in input stream, which makes the parallel processing of elements easy. The size of input stream and output stream is usually exactly the same.*

The second approach is to use *streaming computation languages* based on a *streaming computation model*. In the streaming computation model, a processor performs the same computation on a set of elements and outputs the resulting elements. The computation code is called a *kernel* and the set of elements is called a *stream*. This model makes parallel processing on all the elements easy because kernels perform exactly the same operations on all the elements. Moreover, it is usually assumed that computation in a kernel is independent from the result of output stream. In other words, we can stream data into computation to get resulting data, without worry about the dependency chain between data. In fact, the shader pipeline of graphics hardware can be mapped into the streaming computation model by considering vertex/fragment data as a stream (Figure 3).

Streaming computation languages are often designed for specific hardware in order to enable more flexible control over the programmable functionalities. For instance, NVIDIA CUDA is only for GeForce 8 series or later. The syntax of streaming programming languages is often very similar to other high-level languages on CPUs. In CUDA, we can write a program on graphics hardware that is similar to a multi-threaded C++ program. We now have various choices of streaming processing languages, such as OpenCL (http://www.khronos.org/opencl/) and CTM (http://sourceforge.net/projects/amdctm/).

3 Ray Tracing on Graphics Hardware

3.1 CPU/GPU Hybrid Ray Tracing

The earliest work on ray tracing on programmable graphics hardware is "Ray Engine" by Carr et al. [2002]. In Ray Engine, a CPU performs ray traversal on an acceleration data structure and generates a list of candidate triangles for ray-triangle intersection computation. Graphics hardware then computes ray-triangle intersections for these triangles by transferring triangle data from CPU memory. Since the process of ray-triangle intersections on graphics hardware does not need to know the underlying representation of the acceleration data structure, the computation on graphics hardware becomes relatively simple. The main reason behind this system design is that early programmable graphics hardware had severe limitations (such as the number of instructions, precision on computations and the amount of memory), which made it difficult to implement ray traversal on graphics hardware. Figure 4 shows the diagram of the algorithm and Figure 5 shows one of the results produced by Ray Engine.

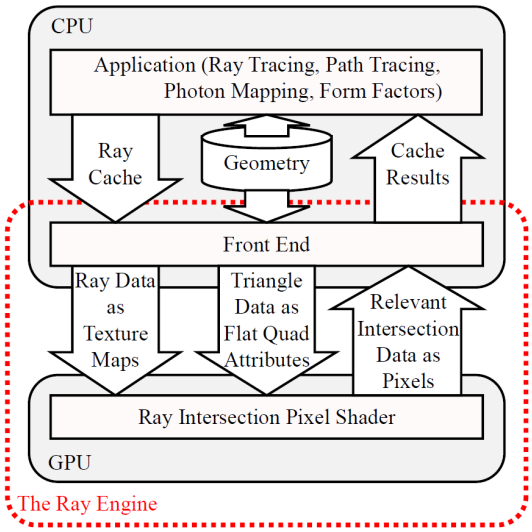

Figure 4: *The organization of Ray Engine [Carr et al. 2002]. Note that both ray and triangle data are maintained on the CPU. The only role of graphics hardware (GPU) is to compute ray-triangle intersections. Because the CPU and graphics hardware need to cooperate, there is significant amount of data transfer between them.*

The main drawback of this approach is that there is significant amount of data transfer between CPUs and graphics hardware, which is relatively costly compared to computations. For instance, this approach requires transferring triangles data from CPUs to graphics hardware and intersections data from graphics hardware to CPUs. Because of this data transfer, Carr et al. concluded that the performance is not comparable with implementation on CPUs even if we assume infinitely fast computations on graphics hardware. For instance, Wald et al. [Wald et al. 2001] reported 300K-1M rays per second on a single core CPU, in comparison to 250K rays per second in Ray Engine with infinitely fast computations.

In contrast to the overall performance of ray tracing, the performance of ray-triangle intersections on graphics hardware is significantly faster than that on CPUs. The graphics hardware implementation of ray-triangle intersection achieved 114M intersections per second on ATI Radeon R200 (the core clock is 275 MHz), whereas the fastest CPU ray-triangle intersection at that time [Wald et al. 2001] only achieved between 20M to 40M intersections per second on a Pentium III 800MHz. This is mainly because the ATI Radeon R200 has 4 independent processing units for pixels. This makes it possible to process 4 ray-triangle intersections in parallel. As a consequence, this work showed that ray traversal needs to be accelerated on graphics hardware as well to fully exploit computational power of graphics hardware.

Figure 5: *Example image rendered by [Carr et al. 2002]. The number of triangles in this scene is 34,000. The absolute performance is 114,499 rays per second on NVIDIA GeForce4 Ti.*

3.2 GPU Ray Tracing

Since the efficient usage of acceleration data structure is the key to improve performance of ray tracing, most of recent work on ray tracing on graphics hardware has focused on ray traversal on graphics hardware. We discuss these methods based on the classification by acceleration data structure.

3.3 Uniform Grids

The first complete ray tracer that includes ray traversal on graphics hardware used a uniform grid [Purcell et al. 2002]. A uniform grid subdivides the bounding box of an entire scene into uniformly sized voxels, which are rectangular subspaces of the bounding box. Each voxel contains a list of triangles that overlap with itself. As the initialization of ray traversal, the intersection between a ray and the bounding box of the scene is computed. Based on the intersection, the voxel where the ray enters into the uniform grid can be found. After we found the initial voxel, the ray advances into the next voxel based on the ray direction. If the voxel contains any triangles, ray-triangle intersections are performed to obtain actual intersection points. Since ray-triangle intersections are performed only for voxels that are pierced by the ray, uniform grid reduces the number of ray-triangle intersections.

Purcell et al. proposed the first ray tracing algorithm that runs entirely on graphics hardware [Purcell et al. 2002]. They consider programmable graphics hardware as a *streaming processor*. A streaming processor is the processor model used in streaming programming languages, which uses a stream of elements as the input and output data. Those elements are processed in parallel by kernels. Since a kernel does not vary between different elements in the same stream, parallel processing on the stream becomes easy. Programmable graphics hardware can be considered as a streaming processor because it executes the same vertex/fragment program on

a set of vertices/fragments. This viewpoint was later evolved into streaming programming languages on graphics hardware.

Ray traversal on a uniform grid can be easily mapped onto this streaming processor model. The ray traversal of a uniform grid only needs to know the current voxel index, the ray origin, and the ray direction in order to compute the next voxel. The input streams are current voxel indices, ray origins and ray directions. The kernel (which performs the single step of the ray traversal in an uniform grid) outputs the next voxel indices as the output stream. Figure 6 summarizes the algorithm.

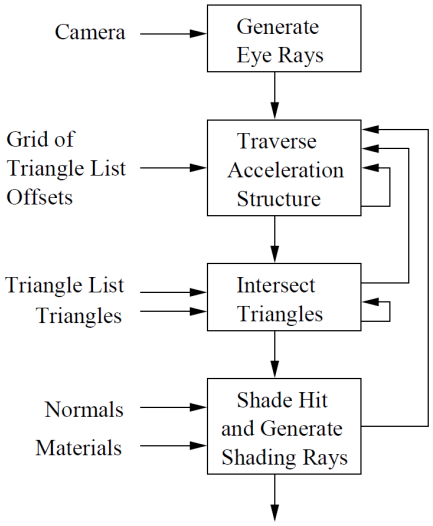

Figure 6: *Streaming ray tracing using uniform grid [Purcell et al. 2002]. The boxes are kernels of streaming computation. Note that the entire ray tracing process, including ray traversal, is mapped to the streaming processing model.*

Based on this streaming processing model, the entire ray tracing process can be directly mapped on programmable graphics hardware, without frequent data transfers between the CPU and graphics hardware. Note that the construction of an uniform grid is still done on the CPU. This algorithm (i.e., constructing the acceleration data structure on the CPU and perform ray traversal on graphics hardware) has been extensively used in other work on ray tracing on graphics hardware. Since the programmable graphics hardware was not flexible enough to run this algorithm, Purcell et al. reported results by software simulation. Karlsson et al. [2004] later implemented this algorithm on NVIDIA GeForce 6800, and reported that the performance is almost comparable with a commercial ray tracing on CPUs (Figure 7 shows one of the results).

The uniform grid data structure is efficient only for uniform triangle distribution, however, it does not work well with irregular distribution of triangles. This problem is known as the *teapot in a stadium* problem. This refers to the fact that a single voxel will contain all the triangles of the teapot if we create a uniform grid over the stadium (see Figure 8 for example). Since we compute ray-triangle intersections in each voxel, the number of ray-triangle intersections for the teapot does not decrease.

3.4 kD-tree

One acceleration data structure that can avoid the problem of the uniform grid is the *kD-tree*. By using a kD-tree, a scene is split into small regions by hierarchy of planes. Each non-leaf node of a kD-tree stores a plane that splits the space into two halves. This subdivision continues until some criterion (e.g., the number of triangles per leaf node) is met. Finally, each leaf node stores a list of triangles that overlap with itself. Figure 9 shows the example of 2D kD-tree. Note that we use a 3D kD-tree in practice.

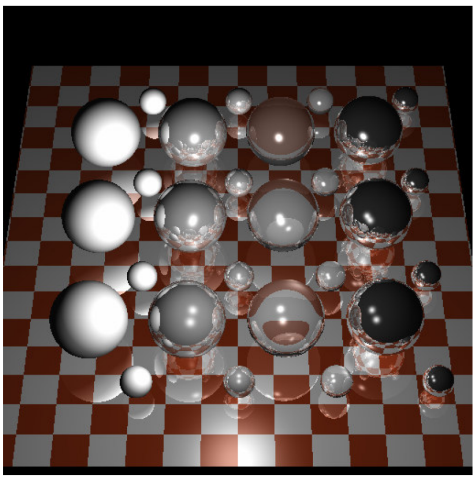

Figure 7: *The example image rendered by [Karlsson and Ljungstedt 2004]. The rendering time is 6 seconds on NVIDIA GeForce 6800 GTX, which is comparable to the performance of the CPU implementation in Autodesk 3ds Max (6.2sec).*

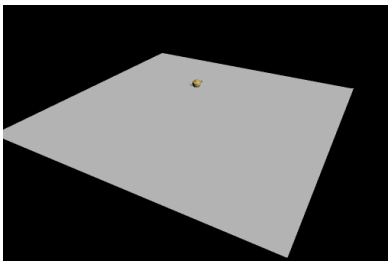

Figure 8: *Example of the "teapot in a stadium" problem. If we generate a uniform grid on the entire scene (including the large ground plane), the teapot in the center is likely to be in a single voxel, which does not help to reduce the number of ray-triangle intersections for the teapot.*

The ray traversal of kD-tree can be efficiently implemented using a local stack. Figure 10 shows the example code of the ray traversal for the kD-tree. Unfortunately, directly porting this algorithm to graphics hardware is not possible because graphics hardware does not have a local stack on each element (vertex or pixel). Emulating a local stack on graphics hardware using the programmable functionalities has been experimented by Ernst et al. [2004], but it is not efficient for ray tracing because of the overhead of emulation.

In order to avoid using a stack in the ray traversal, Foley et al. proposed two new stackless kD-tree traversal methods [2005] called *kD-restart* and *kD-backtrack*. Figure 11 describes the overall algorithms of the two methods. In both methods, we maintain the interval of ray during ray traversal.

The kD-restart method simply restarts the ray traversal from the root when the ray exits the current leaf node. Note that a ray does not revisit the same node because the ray interval has been reduced when the ray exits the node. If n is the number of nodes, the worst

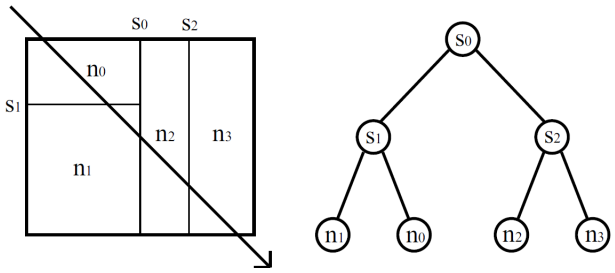

Figure 9: *Example of 2D kD-tree [Foley and Sugerman 2005]. S_0, S_1, S_2 are splitting planes and n_0, n_1, n_2, n_3 are leaf nodes. The tree on right shows the hierarchy of this kD-tree.*

case cost becomes $O(n \log(n))$, in comparison to $O(n)$ with the stack-based kD-tree traversal. The extra $O(\log(n))$ comes from the fact that we can expect the number of leaf nodes visited by a ray is $O(\log(n))$.

The kD-backtrack method reduces this additional cost by using backtracking into the ancestor node with an additional parent pointer per node. Foley et al. pointed out that the nodes that are pushed on the stack are always child nodes of the ancestor nodes in the stack-based kD-tree traversal. Therefore, by backtracking to the ancestor nodes, we can obtain all the nodes in stack without explicitly storing them in stack. The kD-backtrack method maintains the worst case cost to $O(n)$. Note, however, that backtracking to the ancestor nodes causes additional overhead. Moreover, each node should store the pointer to its parent node, which increases the memory consumption.

Regardless of the additional overhead from the standard kD-tree traversal, the resulting performance is up to 8 times faster than the uniform grid method with scenes like *teapot in a stadium*. Figure 12 shows some statistics of the performance.

Horn et al. later extended the kD-restart method to improve its performance by the *push-down* traversal [Horn et al. 2007]. Push-down traversal is based on the observation that a ray tends to intersect with only a small subtree of kD-tree. Horn et al. pointed out that the kD-restart algorithm is ignoring this property because it always restarts the ray traversal from the root. This results in repeating the redundant computation for nodes that will never intersect with the ray. Instead of restarting from the root, the push-down traversal restarts from the node that encloses the interval. Such node can be easily found by looking at a node that has not been visited during the traversal. In other words, the push-down traversal performs backtracking to find such node, instead of just restarting from the root. They also introduced *short-stack* to combine with the stack-based traversal. If the stack overflows, this method switches to the stackless traversal with the push-down traversal. Figure 13 shows the improvement with these modifications. In order to reduce the required bandwidth, Horn et al. also implemented the entire method as a single kernel on recent graphics hardware.

Horn et al. pointed out that kD-backtrack requires larger bandwidth for loading the parent pointers. This is the main reason why they modified the kD-restart method, despite its worst-case cost. They also noted that traversing packets of rays is easier with the kD-restart method. The idea of using packets of rays was previously proposed by Wald et al. [2001] in the context of real-time ray tracing on CPUs, which significantly amortizes the cost of ray traversal by performing the traversal by the bounding frustum or cone of rays. "Packets" in Figure 13 shows the improvement by this optimization. It assumes that packets of rays are coherent (i.e., they

```
kd-search( tree, ray )
   (global-tmin, global-tmax) = intersect( tree.bounds, ray )
   search-node( tree.root, ray, global-tmin, global-tmax )

search-node( node, ray, tmin, tmax )
   if( node.is-leaf )
      search-leaf( node, ray, tmin, tmax )
   else
      search-split( node, ray, tmin, tmax )

search-split( split, ray, tmin, tmax )
   a = split.axis
   thit = ( split.value - ray.origin[a] ) / ray.direction[a]
   (first, second) = order( ray.direction[a], split.left, split.right )

   if( thit >= tmax or thit < 0 )
      search-node( first, ray, tmin, tmax )
   else if( thit <= tmin )
      search-node( second, ray, tmin, tmax )
   else
      stack.push( second, thit, tmax )
      search-node( first, ray, tmin, thit )

search-leaf( leaf, ray, tmin, tmax )
   // search for a hit in this leaf
   if( found-hit and hit.t < tmax )
      succeed( hit )
   else
      continue-search( leaf, ray, tmin, tmax )

continue-search( leaf, ray, tmin, tmax )
   if( stack.is-empty )
      fail()
   else
      (n, tmin, tmax) = stack.pop()
      search-node( n, ray, tmin, tmax )
```

Figure 10: *Standard kD-tree traversal pseudocode [Foley and Sugerman 2005]. Note that we need to use a local stack.*

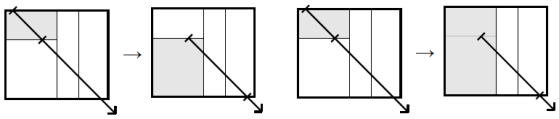

Figure 11: *Left: The* kD-restart *advances the interval of rays forward if there is no intersection at a leaf node before. The down traversal of the tree find a new leaf node based on this new interval. Right: The* kD-backtrack *resumes the search at the first ancestor node that intersect with the new interval. We perform up traversal of the tree to find such ancestor node.*

have similar directions and origins). Therefore, the performance goes down if rays diverge after reflections on a complex surface.

Popov et al. [2007] proposed another stackless kD-tree traversal by adding auxiliary links between the neighboring nodes in a kD-tree. The same idea had been proposed as the *rope-tree* in the context of ray tracing [Havran et al. 1998] on CPUs, but it was not efficient on CPUs. A rope-tree is a modification of kD-tree. Each node in a rope-tree has pointers called *ropes* which stores the neighboring nodes of each face of the node (i.e., a 3D node have 6 ropes). Figure 15 illustrates the example of rope-tree. Using ropes, the ray traversal is significantly simplified and does not require a local stack. The ray traversal of rope-tree simply chooses a rope based on the face of the node that a ray intersects, and steps into the next node based on the selected rope. Figure 16 shows the absolute performance of their implementation.

The main issue of rope-tree is that it increases the size of kD-tree significantly. As shown in Figure 17, adding the ropes increased the size of tree to about 3 times larger than the original kD-tree. Since

	Cornell		Kitchen		Robots		Conference	
	Primary	Shadow	Primary	Shadow	Primary	Shadow	Primary	Shadow
Plain Restart	38.3	34.8	8.6	17.1	7.7	9.5	9.1	15.2
Packets	17.5	35.3	13.3	21.1	14.0	13.5	13.6	15.9
Push-Down	88.8	74.7	12.5	21.4	14.7	12.8	13.9	16.1
Short-Stack	91.3	121.3	16.3	27.3	17.9	16.2	15.2	18.8

Figure 13: *The performance of the number of millions of rays per second in the resolution of* 1024^2 *pixels on ATI Radeon X1900 XTX [Horn et al. 2007]. Each line includes the modifications above it in the table (i.e., the last line includes all modifications). All scenes are the same as in [Foley and Sugerman 2005] except for the conference scene which consists of 282,801 triangles. "Primary" traced the rays that find visible points from camera and Shadow traced the rays for computing occlusion from a light source.*

scene	OpenRT primary rays frustum	CPU: Stackless primary rays only		GPU: Stackless primary rays only		GPU: Stackless with 2ndary rays	
		single	packet	single	packet	single	packet
SHIRLEY6	6.6	3.80	3.49	10.6	36.0	4.8	12.7
BUNNY	—	2.16	1.71	8.9	12.7	4.9	5.9
FAIRYFOREST	3.6	1.57	1.27	5.0	10.6	2.5	4.0
CONFERENCE	3.9	2.14	1.78	6.1	16.7	2.7	6.7

Figure 16: *Absolute performance for a single ray and packet traversal [Popov et al. 2007]. The CPU implementation uses the same algorithm on Opteron 2.6GHz. The GPU performance is measured on GeForce 8800 GTX. The performance is the number of rendered images per second with resolution of* 1024^2.

	Brute	Grid	Restart	Backtrack
Cornell Box	23	63	80	84
Bunny	4620	357	701	690
Robots	4770	8344	968	946
Kitchen	7350	2687	992	857

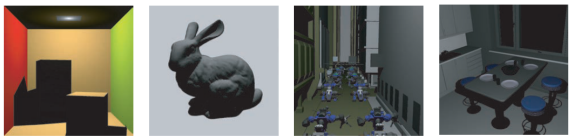

Figure 12: *Top: rendering time in millisecond for* 512^2 *pixels for primary rays (rays that find visible points from the camera). Brute used the naive all ray-triangle intersections. Grid used uniform grid based [Purcell et al. 2002]: Scenes used for the measurement. From left to right: Cornell Box - 32 triangles,* 2^3 *grid. Bunny - 69,451 triangles,* 50^3 *grid. Robots - 71,708 triangles,* 50^3 *grid. Kitchen: 110,561 triangles,* 50^3 *grid. All results from [Foley and Sugerman 2005].*

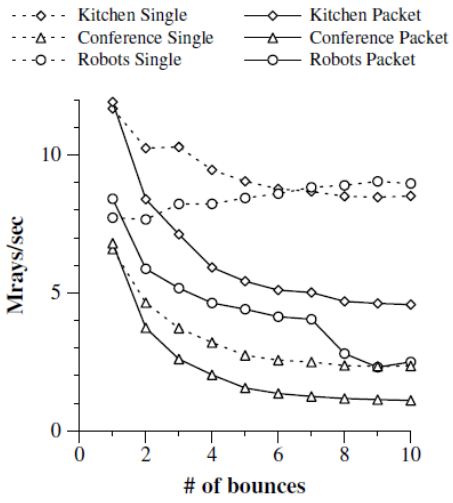

Figure 14: *The performance for specular reflections with/without packets [Horn et al. 2007]. 'Single' does not use the packets optimization. Note that the packet optimization actually decreases the performance as we increase the number of bounces*

graphics hardware still has a smaller amount of memory than CPUs, this severely limits the size of scene that can be handled. Note that all the data needs to be in the memory since there is no memory paging on current graphics hardware. Additional data loading due to the rope is also problematic because it increases the required number of registers and bandwidth. Due to its overhead, the authors calculated that the number of cycles per ray is about 10,000 in their implementation, which is still significantly large compared to 1,000 of the highly optimized CPU ray tracing [Reshetov et al. 2005].

More recently, Zhou et al. implemented the stack-based kD-tree traversal, as well as a kD-tree construction using NVIDIA CUDA [Zhou et al. 2008]. The performance is only comparable to optimized CPU ray tracing methods. Budge et al. [2008] also demonstrated the implementation of mono-ray (i.e., non-packet) stack-based traversal using CUDA with the similar performance. Although the comparable performance may sound reasonable, we have to take into account that graphics hardware has larger raw computational power than CPUs. For example, ATI Radeon HD 4800 achieves about 1.2TFLOPs, whereas Intel Core2 Quad QX9770 has 51GFLOPs. If the efficiency of ray tracing implementation is the same on both graphics hardware and CPUs, ray tracing on graphics hardware needs to be significantly faster than CPU ray tracing. Therefore, the comparable performance means that ray tracing on graphics hardware is very inefficient than CPU ray tracing.

		kd-tree properties					
scene	#tris	#leaves	#empty leaves	references	size	size with ropes	rope overhead
SHIRLEY6	804	3,923	1,021	1.86	82.4 kB	266.3 kB	3.23
BUNNY	69,451	349,833	183,853	2.53	6.9 MB	23.0 MB	3.30
FAIRYFOREST	174,117	721,083	382,345	3.01	14.9 MB	47.9 MB	3.22
CONFERENCE	282,641	1,249,748	515,970	3.13	27.8 MB	85.0 MB	3.06

Figure 17: *The size overhead caused by ropes [Popov et al. 2007]. Adding ropes increases the size of kD-tree by about 3 times regardless of the number of triangles.*

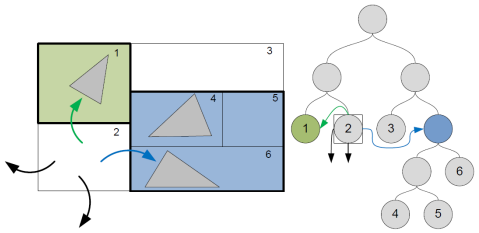

Figure 15: *A kD-tree with ropes [Popov et al. 2007]. Each rope in a face of leaf node points the smallest node that encloses all adjacent nodes of the face.*

3.5 Bounding Volume Hierarchy

Both uniform grid and kD-tree subdivide space into smaller subspaces. We can alternatively subdivide a list of triangles into smaller subsets instead of subdividing space. Bounding Volume Hierarchy (BVH) uses this approach to construct the tree of bounding volumes. Bounding volume is defined as an enclosing shape of all the triangles in a set. Although the choice of the bounding volume shape is arbitrary, Axis Aligned Bounding Box (AABB) is commonly used due to its efficiency of the ray-AABB intersection computation. Figure 18 shows the example of BVH.

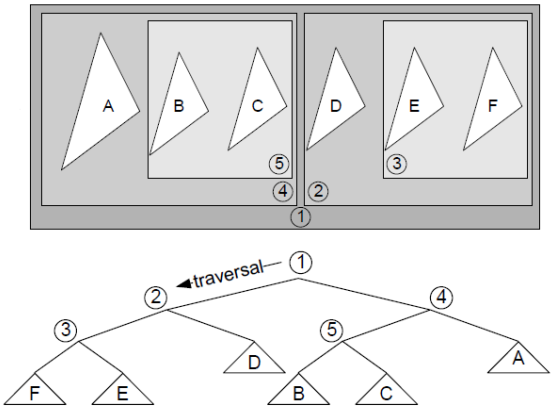

Figure 18: *BVH using AABB [Thrane and Simonsen 2005]. Each node is an AABB of its child nodes, and the root node is the AABB of entire scene. Leaf nodes contain a list of triangles.*

Similar to kD-tree, the ray traversal of BVH on CPUs uses a local stack. Therefore, directly porting the ray traversal of BVH on CPUs to graphics hardware is difficult. In order to avoid this issue, Thrane and Simonsen used *threading* of BVH [Thrane and Simonsen 2005]. Threading connects a node with a parent and sibling node that indicate the next node during ray traversal. These connections are chosen during ray traversal (depending on hit/miss on the node); thus they are called hit and miss links. For example, if a ray missed the left node, the sibling node (i.e., the right node) is used to continue the ray traversal. Figure 19 shows the example of threaded BVH. Note that a similar idea is used by the stackless traversal of kD-tree [Popov et al. 2007] as ropes. The traversal algorithm does not use a stack because the next node is always determined by hit/miss link. Thrane and Simonsen reported that the performance of their implementation is better than both kD-tree and uniform grid (see Figure 20 for some statistics).

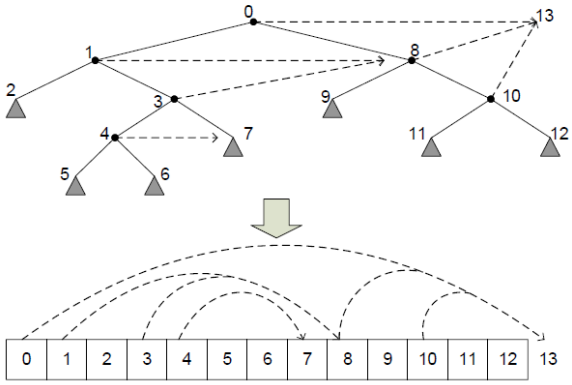

Figure 19: *Threading of BVH [Thrane and Simonsen 2005]. The dotted lines show miss link. In this representation, a hit link is represented as the pointer to next node in the linear array shown below.*

	Cows	Robots	Kitchen	Bunny	Cornell
Uniform grid	770	4626	3269	667	205
Kd-tree (Restart)	728	2771	1961	1299	203
Kd-tree (Backtrack)	567	2619	1908	913	220
BVH (Kay/Kajiya)	195	1017	556	257	143
BVH (Goldsmith/Salmon)	183	825	476	275	103

Figure 20: *Absolute performance comparison between different data structures including BVH [Thrane and Simonsen 2005]. The table shows rendering times in milli seconds for rendering an image with the resolution of 512^2 on NVIDIA GeForce 6800 Ultra. Two BVHs at the bottom only differs at the construction of BVH. kD-tree traversal is from [Foley and Sugerman 2005] and the scenes included in this table are exactly the same as they used.*

Carr et al. used the same threading approach for ray tracing of dynamic scenes [Carr et al. 2006]. The difference from previous work [Thrane and Simonsen 2005] is that scene geometry is represented as a geometry image [Gu et al. 2002], which is a 2D parameterization of a triangle mesh. In geometry images, each triangle

is formed by connecting neighboring three pixels that store vertex positions and normals as pixel data. The construction of BVH over geometry images is simplified to the construction of hierarchical tree over pixels, which is efficiently done on graphics hardware (see Figure 21 for example). Since this process is very fast, this algorithm can handle dynamic geometry. If the some vertices moved, the BVH can be quickly rebuilt only by updating the size of bounding boxes efficiently. Note that the tree structure of BVH nodes, including threading, does not change because of fixed topological links between pixels on geometry images. Unfortunately, using a single geometry image is restricted to handling a single connected mesh. For example, a car with rotating wheels needs to use 5 geometry images (1 for a car body and 4 for wheels). Since using 5 geometry images requires 5 independent ray tracing processes, this method is costly for disconnected meshes.

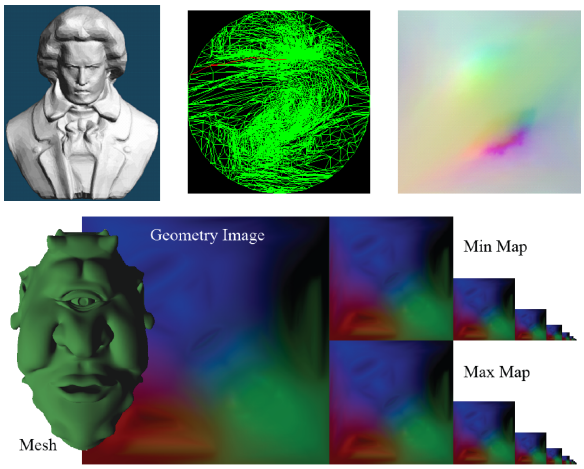

Figure 21: *Geometry images and the BVH construction. Top row: Given a triangle mesh (left), we can parametrize the mesh onto 2D by providing sufficient cuts (middle). By recoding vertex positions and normals in each pixel, we get the 2D image that represents the 3D mesh (right). Bottom row [Carr et al. 2006]: Based on the geometry image, we can easily compute the AABB of BVH at each level by taking minimum and maximum of positions over all pixels in different resolutions.*

scene	[PGSS07] primary	[PGSS07] 2ndary	our GPU ray tracer primary	our GPU ray tracer +shadow
FAIRYFOREST	10.6	4.0	13.2 (14.6)	4.8
CONFERENCE	16.7	6.7	16 (19)	6.1
SODA HALL	—	—	13.6 (16.2)	5.7
POWER PLANT	—	—	6.4	2.9

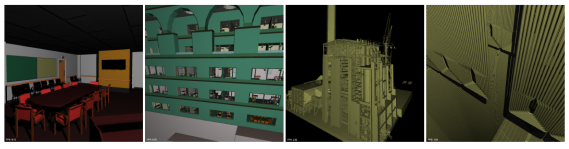

Figure 22: *Top row: the comparison between stackless kD-tree traversal [Popov et al. 2007] and the stack-based BVH traversal on GPU [Günther et al. 2007]. The performance is measured by the number of frames per second (FPS) for a 1024^2 rays on NVIDIA GeForce 8800 GTX. Bottom row: The rendered images. Conference (282,641 triangles), Soda Hall (2,169,132 triangles), Power Plant (12,748,510 triangles) and the inside of Power Plant.*

Guenther et al. implemented a packet-based ray tracing using BVH on graphics hardware [Günther et al. 2007]. The algorithm utilizes a stack instead of the threading technique by using NVIDIA CUDA on GeForce 8 series. Instead of having a local stack per ray, they used a shared local stack between several rays using packet ray tracing [Wald et al. 2001]. By sharing a stack with several rays, the computation overhead and memory cost of stack are amortized over multiple rays in a packet. Moreover, since a stack-based traversal does not require any additional data on an acceleration data structure, their method can handle the very large number of triangles (see the bottom row of Figure 22 for example). The top row of Figure 22 shows that their implementation outperforms the stackless kD-tree traversal [Popov et al. 2007]. The disadvantage of all the stack-based algorithms is that it reduces the amount of parallelism. We discuss this issue later in detail.

3.6 Ray-Space Hierarchy

Since the goal of acceleration data structure is to decrease the number of ray-triangle combinations, we can use the spatial subdivision of rays instead of the spatial subdivision of geometries to achieve the same goal. Szécsi extended Ray Engine [Carr et al. 2002] as *Hierarchical Ray Engine* by using tiles of rays, instead of individual rays [Szécsi 2006]. In this approach, we feed all the triangles to graphics hardware, but the actual ray-triangle computation is performed only if the enclosing cone of the set of rays (tile) intersects the triangle. Since we do not need to construct an acceleration data structure of objects (which is often costly than ray tracing itself), this approach is well-suited for dynamic scenes. Figure 23 shows the block diagram of this algorithm.

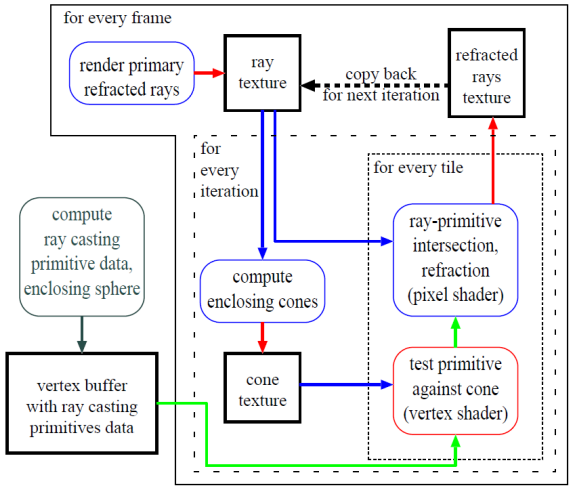

Figure 23: *The block diagram of Hierarchical Ray Engine [Szécsi 2006]. A triangle is first tested against the enclosing cone of set of rays. The ray-triangle intersection computations are executed only if the cone intersects with a triangle.*

Roger et al. [2007] proposed a similar idea for ray tracing of dynamic scenes. They developed a *ray hierarchy* which uses hierarchy of ray tiles, instead of uniformly-sized tiles as in [Szécsi 2006]. The construction of ray hierarchy is efficiently done on graphics hardware because creating the upper level of tiles is operationally similar to image resizing (Figure 24). After constructing the tree, the list of candidate rays for *each triangle* is obtained by traversing the ray hierarchy. Note that this is dual to ordinary ray tracing, where we obtain the list of candidate triangles for *each ray* by traversing the triangle hierarchy. The ray traversal is started from the root node, and it outputs child nodes that intersect with a given triangle. The output of intersecting child nodes needs a random

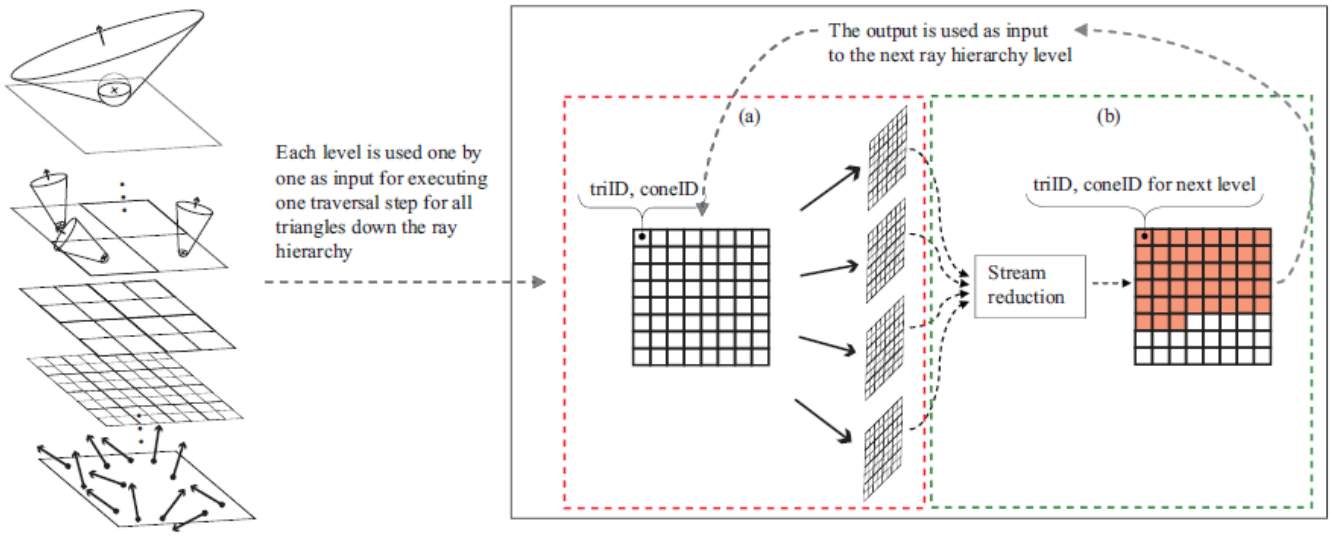

Figure 25: *The traversal of the ray-space hierarchy [Roger et al. 2007]. We first initialize each pixel by storing the root node index and all triangle index. We then perform cone-triangle intersections for each combination and write out the results if a cone intersects with the triangle into 4 separate buffers (since we have 4 child nodes per node). The stream reduction compacts the 4 buffers into one buffer by ignoring a cone that did not intersect with the triangle. Resulting buffer again feeds as the input until we reach leaf nodes.*

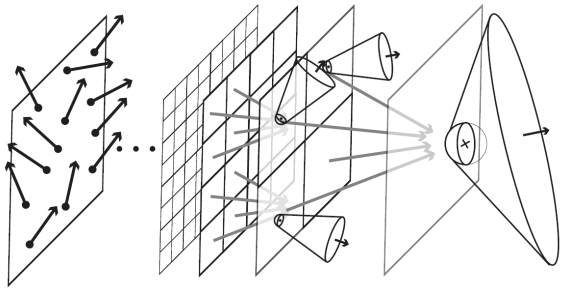

Figure 24: *The hierarchy of cones [Roger et al. 2007]. The last level of hierarchy starts from individual rays (left), and the hierarchy is constructed by grouping 4 neighboring cones into the enclosing cone by the bottom up approach.*

write which is inefficient on graphics hardware. Therefore, all the 4 child nodes are written regardless of the results of intersections with additional valid/invalid flags (i.e., valid if there is an intersection). As a result, some nodes are stored with invalid flags even if corresponding child nodes do not intersect with the triangle. In order to remove those invalid nodes and compact the array of nodes in parallel, Roger et al. developed a new algorithm called *streaming reduction*. Figure 25 summarizes the algorithm. The performance of this method is barely comparable with the kD-tree ray tracing by Horn et al. [2007], which is not comparable with the fastest ray tracing on graphics hardware such as the stackless kD-tree [Popov et al. 2007].

4 Discussion

In this section, we compare and discuss the methods we surveyed in previous section. First, we compare the absolute performance of different methods. We then discuss latest ray tracing methods on CPUs briefly in order to compare it with ray tracing on graphics hardware. We present several reasons behind why graphics hardware is less efficient than CPU ray tracing and propose some solutions later in this section. We also discuss ray tracing other than graphics hardware and CPUs to highlight benefits of ray tracing on graphics hardware. Table 1 summarizes the comparison between different methods in the survey.

4.1 Absolute Performance

Figure 26 compares performance of different methods. Since each method uses different graphics hardware, we scaled the performance based on the bottleneck. For example, if the bottleneck is bandwidth, we scaled the performance by the ratio of bandwidth. This is likely to be a valid scaling because graphics hardware is fully pipelined (i.e., the bottleneck completely determines the entire performance). However, note that this comparison is not very accurate because the performance also depends on several factors such as scene setting (e.g., the camera angle and complexity of the objects) and different architecture of graphics hardware. In addition, some of the methods should not be directly compared because they have different functionalities. For example, ray hierarchy [Roger et al. 2007] is capable of handling dynamic scenes whereas ray tracing using object hierarchy cannot change the shape of objects during runtime. Therefore, this comparison should be considered as a rough estimate of the absolute performance of each method. The first thing we noticed is that packet ray tracing methods generally outperform mono-ray tracing methods (compare [Günther et al. 2007] and [Thrane and Simonsen 2005] for example). This is because packet ray tracing amortizes the cost (memory read/write, computations) of traversal by sharing the same traversal between several rays. Although the original algorithm of packet ray tracing was developed for CPUs, the performance comparison shows that packet ray tracing is also effective on graphics hardware as well. We also noticed that both stackless methods and stack-based methods have similar performance in this comparison. It is contradicting to the fact that almost all stackless methods add the overhead to the

stack-based version. We discuss this point later in this section.

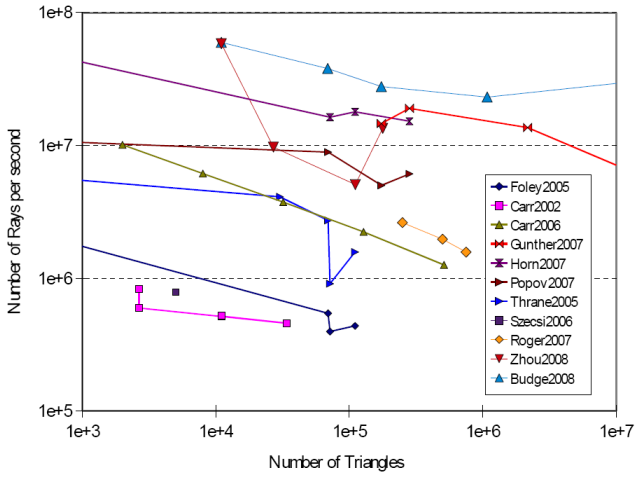

Figure 26: *Absolute performance comparison of different methods. The numbes are scaled according to the specifications of NVIDIA GeForce 8800.*

4.2 Comparison with CPU Ray Tracing

In order to discuss the efficiency of ray tracing on graphics hardware, we describe the latest ray tracing methods on CPUs. One of the significant steps toward high performance ray tracing is the work by Wald et al. [2001]. Their key idea is to trace a coherent set of rays at the same time in order to amortize the cost of ray traversal, which significantly improves the performance of ray tracing. This speedup is because ray traversal is the bottleneck of ray tracing. Note that the performance of ray-triangle intersection does not affect the overall performance of ray tracing. For example, the number of ray-triangle intersections performed is often 2 to 4 whereas the number of ray traversal steps is 30 to 50 [Reshetov et al. 2005].

The idea of using coherent rays is currently the main focus of high performance ray tracing on CPUs. Wald et al. applied the same idea for uniform grid [Wald et al. 2006] and bounding volume hierarchy [Wald et al. 2007a] as well. Recently, Reshetov proposed *vertex culling* [Reshetov 2007], which reduces redundant ray-triangle computations of packet ray tracing using the frustum of rays. Other than the basic idea of using packets of rays, recent work on packet ray tracing has not been used on graphics hardware. We consider that testing recent work of packet tracing on graphics hardware is interesting future work.

Reshetov extended the idea of packet tracing to multiple levels of ray packets (i.e., hierarchy of ray packets) to further improve the performance [Reshetov et al. 2005]. This work is considered to be the fastest ray tracing on CPUs so far. The performance is 109.8 million rays per second for a scene with 804 triangles, 19.5 million rays per second for the conference scene (274K triangles), and 35.5 million rays per second for the soda hall scene (2195K triangles) on Intel Pentium4 3.2GHz using 2 threads with hyperthreading.

The fact that we can obtain this performance on CPUs indicates that ray tracing on CPUs is significantly efficient than ray tracing on graphics hardware. To be concrete, ray tracing on Intel Pentium4 3.2GHz with 6.4 GFLOPS is comparable with ray tracing on NVIDIA GeForce 8800 GTX with 345.6 GFLOPS in terms of the ray tracing performance. For example, the fastest BVH packet tracing [Günther et al. 2007] achieves 16.2 million rays per second for the *soda hall scene* on GeForce 8800 GTX, in comparison to 35.5 million rays per second on two CPU cores [Reshetov et al. 2005]. If we simply scale the performance by FLOPS, GeForce 8800 GTX has to achieve 1.887 billion rays per second for this scene. Although this estimation is not accurate because we compared different methods and these numbers for FLOPS are theoretical values, this performance loss is indicating that ray tracing on graphics hardware is inefficient compared to ray tracing on CPUs. Another possibility is that the performance of ray tracing on graphics hardware is bounded by some factors other than computation since it did not scale by FLOPS. However, almost all methods in this survey are computation-bounded, which performance should be scaled by FLOPS.

4.3 Performance Gap

As we already discussed above, ray tracing on graphics hardware is inefficient compared to ray tracing on CPUs. In this section, we will describe three issues that cause inefficiency. We will also discuss possible solutions for these issues in order to provide the idea for more detailed analysis in future work.

Data retention Since the computation model on graphics hardware is essentially the same as a streaming processor, no data is retained after each computation. For example, if the kernel of ray-triangle intersection performs several ray-triangle intersections for the same ray, we need to reload the ray data each time the kernel is invoked. This wastes bandwidth because the ray data needs to be loaded from memory many times even if it does not change during the computation. In CPUs, such ray data will likely be in the registers or the local cache because the cache algorithm on CPUs will capture frequent accesses to the same data. Although recent graphics hardware has the cache for textures (i.e., array of elements), the cache algorithm exploits the spatially coherent accesses over neighboring elements, not the repeated usage of the same element. In the ray traversal, it is usually rare that such spatial locality exists because neighboring rays will go through a different part of acceleration data structure. We consider that ray tracing on graphics hardware should take into account this difference. We think that sorting rays based on their directions and origins will amortize this issue because similar rays will access the same data, and putting them close to each other facilitates the use of cache on graphics hardware.

Wide-SIMD width Although graphics hardware can exploit higher degree of parallelism in comparison to CPUs, graphics hardware still cannot process each single pixel independently. This means that graphics hardware needs to perform the same computation over several pixels. For example, ATI X1900 XTX requires 48 pixels to do the same computation in order to avoid any idling of processing units [Horn et al. 2007]. This is related to the fact that graphics hardware hides the memory latency with the context switch over several identical computations. We noticed that this is essentially the same as having wider SIMD width because many data share the same computation. Because of this hardware constraint, the computation on graphics hardware becomes most efficient if the flow of computation is uniform over all data (vertices/pixels). However, having the uniform computation flow is especially difficult for ray traversal because each ray tends to have different routes of computations depending on acceleration data structure. Most of ray traversal methods surveyed in this paper have not considered this issue. One of the solutions is packet ray tracing. As we already discussed, packet ray tracing amortizes the cost of ray traversal significantly by sharing the ray traversal over similar rays. However, since packets tracing limits generality of ray tracing, it is not necessarily the optimal solution. We consider that effectively

	Aceeleration Data Structure	Relative Performance	Dynamic Geometry	Packet Tracing	Stack/Stackless	Implementation
[Purcell et al. 2002]	Uniform Grid	N/A	N	N/A	Stackless	N/A
[Carr et al. 2002]	N/A	1.0	N	N/A	Stackless	Shader
[Thrane and Simonsen 2005]	BVH (threaded)	3.6	N	Mono	Stackless	Shader
[Foley and Sugerman 2005]	kD-tree	1.1	N	Mono	Stackless	Shader
[Carr et al. 2006]	BVH (threaded)	5.1	Y	Mono	Stackless	Shader
[Szécsi 2006]	Ray-Space (tile)	N/A	Y	N/A	Stackless	Shader
[Roger et al. 2007]	Ray-Space (hierarchy)	6.2	Y	N/A	Stackless	Shader
[Günther et al. 2007]	BVH	42.3	N	Packet	Stack-based	Streaming
[Horn et al. 2007]	kD-tree	40.9	N	Packet	Stack-based	Shader
[Popov et al. 2007]	kD-tree (rope)	17.2	N	Mono/Packet	Stackless	Streaming
[Budge et al. 2008]	kD-tree	68.6	N	Mono	Stack-based	Streaming
[Zhou et al. 2008]	kD-tree	26.2	Y	Mono	Stack-based	Streaming

Table 1: *Comparison between different methods. The relative performance column shows the rough estimated speedup from [Carr et al. 2002] for 10k triangles. The implementation column shows whether the actual implementation is based on programmable shader (Shader) or streaming processing language (Streaming).*

using wider SIMD width is the key to improve the performance of ray tracing on graphics hardware. We expect that the idea of sorting rays based on their directions and origins will be applicable as well. However, this idea merely amortizes the issue. We need to solve the fact that ray tracing is not suited for wide SIMD architecture. A general solution to this issue is currently the main focus of high performance ray tracing (for example, refer to [Wald et al. 2007b]).

Stackless and Stack-based method Unlike stack-based methods, stackless traversal methods require more computational resources [Horn et al. 2007]. In theory, the performance of stack-based methods should be better than the performance of stackless methods. However, we have shown that this is not necessarily the case, based on the performance comparison between the different methods. We claim that this is because stack-based algorithms express smaller degree of parallelism than stackless algorithms on graphics hardware. Stack-based algorithms require a local stack for each ray. The amount of required memory for stack could be prohibitively large, depending on the number of rays processed in parallel. Therefore, the number of parallel processes is usually restricted by the total size of local stack. For example, the standard BVH traversal usually requires 256 byte per ray for the local stack. Since NVIDIA G80 has 16K byte of memory that can be used for the stack, 64 rays can be processed in parallel using the 256 byte local stack per ray. This is not fully utilizing the parallelism on graphics hardware because NVIDIA G80 can process up to 128 processes in parallel. Based on this viewpoint, we consider that stackless methods are likely to scale well in the future as the number of parallel processing units will increase. Although most recent work is using a stack-based method graphics hardware, further investigation on stackless traversal is necessary.

4.4 Other Related Work

Schmitteler et al. developed a ray tracing hardware architecture, called *SaarCOR* [Schmittler et al. 2002], in order to accelerate ray-triangle intersections. Although ray-triangle intersections are not the bottleneck of ray tracing, SaarCOR significantly outperformed ray tracing on CPUs and graphics hardware due to the efficiency of hardware implementation. Later, Woop et al. extended SaarCOR into *RPU* [Sven Woop and Slusallek 2005] using FPGA, which has programmable functionalities and accelerates both ray-triangle intersection and ray traversal. They reported that the performance is 50-100 times faster than ray tracing on graphics hardware. These results indicate that having special functionalities for ray tracing significantly improves the performance. However, note that hardware ray traversal functionality means that we cannot alter acceleration data structure. This is not desirable property because it cannot incorporate new acceleration data structure such as BIH [Waechter and Keller 2006].

Benthin et al. implemented ray tracing on Cell processor [Benthin et al. 2006]. Cell processor is a heterogeneous multi-core CPU consisting of one general processor called PPE and several specialized vector processors called SPE. PPE is a modified IBM PowerPC, which is not different from a recent single core CPU except for its ability to manage SPEs. SPEs are SIMD processors with a small, fast scratch pad memory called LS (local store) that acts as user-managed cache. The distinct limitation of SPEs is that they cannot directly access the main memory. In order to access data on main memory, it is required to issue DMA transfers explicitly from PPE to SPE's LS. Since the latency of DMA transfer is larger than arithmetic instructions, decreasing the number of DMA transfers is important to avoid idling of SPEs.

In order to avoid complex management of LS, Benthin et al. implemented software caching and software hyperthreading which systematically avoids unnecessary DMA transfers. The idea is to switch the current task to any task that performs arithmetic operations whenever DMA transfer is required. Their implementation achieved comparable performance with a single SPE to Intel Pentium4 2.8GHz. Since Cell processor has 8 SPEs, it is 8 times faster than Intel Pentium4 2.8GHz. One of the interesting observations they made is that a shading process of intersection points becomes the bottleneck in this system.

The shading process requires random accesses to the main memory which frequently invokes DMA transfers. Since graphics hardware has the specialized texture cache to handle such access patterns of shading processes, they mentioned that the implementation on PlayStation3 (which has Cell and graphics hardware in the same system) will improve the performance significantly. We claim that it indicates that using graphics hardware directly for ray tracing as well is probably the optimal solution because shading can be directly performed on graphics hardware without transferring data between another processor such as Cell processor.

5 Future Work

Construction of Acceleration Data Structure An open problem of ray tracing on graphics hardware is the efficient construction of acceleration data structure on graphics hardware. The construction of acceleration data structure can be avoided by using the ray-space hierarchy, but the ray tracing performance is often not comparable to object-space hierarchy (e.g., kD-tree). The main issue

here is how to extract the parallelism in the construction of an acceleration data structure. Shevtsov proposed the parallel version of kD-tree construction on a multi-core CPU [Shevtsov et al. 2007]. The basic idea is to perform construction of a subtree on each core in parallel. However, since degree of parallelism on a multi-core CPU is still significantly smaller than that on graphics hardware, exploiting fine-grained parallelism is still required.

Recently, Zhou et al. developed the algorithm to construct a kD-tree on graphics hardware by building a tree in breadth-first order [Zhou et al. 2008]. It is based on the observation that each node has many objects that can be processed in parallel during the first few subdivision of space. Similarly, after we finely subdivided the space, we can process each node in parallel because we have many nodes. In order to fully exploit parallelism, the algorithm uses breadth-first order for first several levels of subdivisions, and uses depth-first order later. Although their implementation is still barely comparable to the latest methods on CPUs, such as the above mentioned method by Shevtsov et al. [2007], we consider this algorithm as the promising starting point for the fully dynamic construction on graphics hardware.

More recently, Lauterbach et al. [2009] proposed the construction of BVH on graphics hardware. They proposed the following two methods: a method based on similar approach as Zhou et al. [Zhou et al. 2008], and a method using a space filling curve. The second method uses the fact that a space filling curve tends to cluster nearby objects in multidimensional space onto nearby 1D space on the curve. Using the 1D array of objects, the construction of BVH can be done by sorting objects along a space filling curve and clustering two nearby objects into one in a bottom up fashion. Since this method is very fast but tends to degrade the quality of the tree (i.e., the performance of ray traversal is inferior), Lauterbach et al. also proposed the hybrid method. The hybrid method uses the second method only for the upper level of the tree, where the construction cost is high, and uses the first method to maintain the quality of the tree. We consider that this approach could be further explored for different acceleration data structure such as kD-tree.

Specialized Acceleration Data Structure for GPUs In addition to using existing acceleration data structures on graphics hardware, we consider that finding a new acceleration data structure suitable for graphics hardware is interesting future work. As we have already seen in the survey, an acceleration data structure that is efficient on CPUs is not necessarily efficient on graphics hardware. For example, the kD-tree traversal on graphics hardware needs to be stackless, which causes additional overhead compared to the kD-tree traversal for CPUs.

Among different acceleration data structures, we have found that the ray classification could be more suitable for ray tracing on graphics hardware [Arvo and Kirk 1987]. The key idea is to use the spatial subdivision of the 5D space of rays (i.e., 3 for ray origin and 2 for ray direction) that stores the list of objects. This 5D space is first subdivided into subspaces using uniform grid. Any object that intersects at least one ray in each 5D subspace are added to the list. The ray traversal of the ray classification is very simple because each ray corresponds to a point in 5D space. Therefore, the ray traversal performs a single, constant-time query in 5D space and obtains the list of objects. Since it does not require stack, it might be more efficient on graphics hardware. Mortensen et al. recently proposed a similar approach called *visibility field* [2007]. They reduced the dimensionality of space into 4D space by considering each ray as a line rather than a half line. They reported that this method outperformed packet ray tracing by Wald et al. [2001]. This method has not been tested on graphics hardware as far as we know, so we consider it is interesting to see whether this is efficient on graphics hardware as well.

Utilizing Rasterization Another interesting future work is utilizing the rasterization functionality of graphics hardware. For example, rasterization has been used for accelerating eye-ray tracing from a camera, because the result of ray tracing eye rays is equivalent to rasterization. Another application of rasterization for ray tracing is ray tracing a set of parallel rays (called ray bundle [Szirmay-Kalos 1998]) by parallel projection [Hachisuka 2005]. Recent work even shows rendering complex refraction and caustics (e.g., light focusing pattern under a pool) using rasterization [Wyman 2005; Shah et al. 2007], which have been considered difficult with rasterization. Although these techniques are not exactly equivalent to ray tracing due to their approximations, resulting images are visually equivalent. We consider that it is possible to use these rasterization techniques as a hint for efficient ray tracing. For example, if we know the bound of intersection points using rasterization, we can use this bound to reduce the number of ray traversal steps.

Ray Tracing Hardware Architecture Based on the survey, we propose an "ideal" hardware architecture for the ray tracing algorithm. In general, we think that hardware implementations are preferable to software implementations using programmable functionality. The reason is that programmable functionality has the overhead of interpreting a user program compared to hardware implementations.

First, we think that the ray-triangle intersection algorithm should be implemented on hardware, not by the programmable functionality. This is because there are several mature algorithms for computing ray-triangle intersections [Kensler and Shirley 2006], which can be implemented directly in hardware without a loss of flexibility in the ray tracing algorithm.

However, the ray traversal algorithm should not be completely hardwired as in previous work [Sven Woop and Slusallek 2005]. The reason is that each acceleration data structure is suitable for a different type of scene. For example, a BVH is suitable for dynamic scenes, whereas a kD-tree results in the best performance for static scenes. Therefore, we would like to have the flexibility to change the acceleration data structure based on the scene type, while keeping the efficiency of a hardware implementation. One idea for balancing between flexibility and efficiency is to implement the tree traversal procedure in hardware. For example, we could have a hardware tree traversal algorithm with a programmable criterion. Switching between a kD-tree and a BVH would be performed by just changing the criterion (e.g., hit/miss for BVH traversal and a ray interval for kD-tree).

Finally, we note that current graphics hardware architecture is well suited for the shading computation. Previous work on ray tracing on the Cell processor also suggests this is likely to be the case [Benthin et al. 2006]. Thus, we believe that an "ideal" hardware architecture for ray tracing combines current graphics hardware architecture with the specialized ray tracing hardware functions as discussed above.

6 Conclusion

In this paper, we surveyed several ray tracing algorithms on graphics hardware. Implementing ray tracing on graphics hardware is not just the matter of engineering effort. In order to implement efficient ray tracing, the algorithm should be modified so that it utilizes the parallelism on graphics hardware. We have seen that the current performance of ray tracing on graphics hardware is compa-

rable with a highly optimized CPU ray tracing method. However, we also pointed out that current algorithm of ray tracing on graphics hardware is significantly inefficient compared to CPU ray tracing, if we consider the difference between the raw computational power of CPUs and graphics hardware. In order to fully harness the power of graphics hardware for ray tracing, we need to take into account architectural differences between CPUs and graphics hardware. We discussed several issues that have not been well handled by existing methods. Since we consider that the parallel nature of programming model on graphics hardware will be the standard model for high performance computing in the future, current work should be applicable for other parallel processing platform, not only for graphics hardware.

Acknowledgments

I would like to acknowledge Henrik Wann Jensen for insightful suggestions for the points of discussion, Krystle de Mesa for helpful and thorough comments on the overall text, and Will Chang for well-directed comments on the ray tracing hardware architecture section.

References

ARVO, J., AND KIRK, D. 1987. Fast ray tracing by ray classification. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 55–64.

BENTHIN, C., WALD, I., SCHERBAUM, M., AND FRIEDRICH, H. 2006. Ray Tracing on the CELL Processor. *Technical Report, inTrace Realtime Ray Tracing GmbH, No inTrace-2006-001 (submitted for publication)*.

BUDGE, B. C., ANDERSON, J. C., GRATH, C., AND I., K. 2008. A hybrid cpu-gpu implementation for interactive ray-tracing of dynamic scenes. Tech. Rep. CSE-2008-9, University of California, Davis Computer Science.

CARR, N. A., HALL, J. D., AND HART, J. C. 2002. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 37–46.

CARR, N. A., HOBEROCK, J., CRANE, K., AND HART, J. C. 2006. Fast gpu ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of Graphics Interface 2006*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 203–209.

DUTRE, P., BALA, K., BEKAERT, P., AND SHIRLEY, P. 2006. *Advanced Global Illumination*. AK Peters Ltd.

ERNST, M., VOGELGSANG, C., AND GREINER, G. 2004. Stack implementation on programmable graphics hardware. 255–262.

FOLEY, T., AND SUGERMAN, J. 2005. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, New York, NY, USA, 15–22.

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry images. *ACM Trans. Graph. 21*, 3, 355–361.

GÜNTHER, J., POPOV, S., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, 113–118.

HACHISUKA, T. 2005. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, ch. High-Quality Global Illumination Rendering Using Rasterization, 613–635.

HAVRAN, V., BITTNER, J., AND SÁRA, J. 1998. Ray tracing with rope trees. In *14th Spring Conference on Computer Graphics*, L. S. Kalos, Ed., 130–140.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 167–174.

KARLSSON, F., AND LJUNGSTEDT, C. J. 2004. *Ray tracing fully implemented on programmable graphics hardware*. Master's thesis, Chalmers University of Technology.

KENSLER, A., AND SHIRLEY, P. 2006. Optimizing ray-triangle intersection via automated search. *rt 0*, 33–38.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA1, D., 2009. Fast bvh construction on gpus. To be appeared on Eurographics 2009.

MORTENSEN, J., KHANNA, P., YU, I., AND SLATER, M. 2007. A visibility field for ray tracing. In *CGIV '07: Proceedings of the Computer Graphics, Imaging and Visualisation*, IEEE Computer Society, Washington, DC, USA, 54–61.

OWENS, J. D., LUEBKE, D., NAGA GOVINDARAJU, M. H., KRÜGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1, 80?–113.

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 (Sept.), 415–424. (Proceedings of Eurographics).

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *ACM Trans. Graph. 21*, 3, 703–712.

PURCELL, T. J. 2004. *Ray tracing on a stream processor*. PhD thesis, Stanford, CA, USA. Adviser-Patrick M. Hanrahan.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, 1176–1185.

RESHETOV, A. 2007. Faster ray packets - triangle intersection through vertex culling. In *SIGGRAPH '07: ACM SIGGRAPH 2007 posters*, ACM, New York, NY, USA, 171.

ROGER, D., ASSARSSON, U., AND HOLZSCHUCH, N. 2007. Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the gpu. In *Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering)*, the Eurographics Association, J. Kautz and S. Pattanaik, Eds., Eurographics and ACM/SIGGRAPH, 99–110.

SCHMITTLER, J., WALD, I., AND SLUSALLEK, P. 2002. Saarcor: a hardware architecture for ray tracing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 27–36.

SHAH, M. A., KONTTINEN, J., AND PATTANAIK, S. 2007. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics 13*, 2, 272–280.

SHEVTSOV, M., SOUPIKOV, A., AND KAPUSTIN, A. 2007. Highly parallel fast kd-tree construction for interactive ray tracing. *Computer Graphics Forum 26*, 3, 305–404.

SVEN WOOP, J. S., AND SLUSALLEK, P. 2005. Rpu: A programmable ray processing unit for realtime ray tracing. In *Proceedings of ACM SIGGRAPH 2005*.

SZÉCSI, L. 2006. The hierarchical ray engine. In *WSCG '2006: Proceedings of the 14th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2006*, 249–256.

SZIRMAY-KALOS, L. 1998. Global ray-bundle tracing. Tech. Rep. TR-186-2-98-18, Vienna.

THRANE, N., AND SIMONSEN, L. O. 2005. *A comparison of acceleration structures for GPU assisted ray tracing*. Master's thesis, University of Aarhus.

WAECHTER, C., AND KELLER, A. 2006. Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006 (Proceedings of 17th Eurographics Symposium on Rendering*, 139–149.

WALD, I., BENTHIN, C., WAGNER, M., AND SLUSALLEK, P. 2001. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, Blackwell Publishers, Oxford, A. Chalmers and T.-M. Rhyne, Eds., vol. 20, 153–164. available at http://graphics.cs.uni-sb.de/ wald/Publications.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph. 25*, 3, 485–493.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1, 6.

WALD, I., GRIBBLE, C. P., BOULOS, S., AND KENSLER, A. 2007. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Tech. Rep. UUSCI-2007-012.

WYMAN, C. 2005. An approximate image-space approach for interactive refraction. *ACM Trans. Graph. 24*, 3, 1050–1053.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B., 2008. Real-time kd-tree construction on graphics hardware. ACM Transaction on Graphics (conditionally accepted).