2
Evolution of Ray Tracing

In this chapter we review several research results using either the CPU or the GPU to perform ray-tracing computations. Different acceleration structures have been developed on the CPU in order to render dynamic scenes. On the other hand, the graphics hardware has always impaired an efficient ray-tracing implementation that also supports object movements and deformations. Only a single ray-tracing technique uses state-of-the-art GPUs to support dynamic scenes. Its results prove that the graphics hardware is capable of outperforming even the fastest CPU implementations.

2.1
Dynamic Scenes on the CPU

Over the last few years, different spatial structures have been proposed to accelerate ray tracing. Amongst others, the *kd-tree* has usually shown best performance in the general case [Havran 2000, Wald 2004]. Its main advantage is a simple ray-traversal procedure that requires only one multiplication and one addition per node visited. Indeed, one of the fastest ray-tracing implementations have combined a technique called *inverse frustum culling* with a kd-tree to achieve impressive results [Reshetov et al. 2005].

The process of building a high-quality kd-tree is fairly complicated. The best known strategy involves minimizing a cost function, known as *Surface Area Heuristic* (SAH) [Wald and Havran 2006]. Even with an asymptotically optimal implementation, a modern CPU still spends several seconds to build a kd-tree for a non-trivial scene (a hundred thousand triangles, for instance). This elevated cost has limited kd-tree usage to static scenes, where the acceleration structure can be built in a pre-processing step.

In order to support dynamic scenes, compromises must be made. One can use an acceleration structure that is not so efficient but that can be quickly modified or rebuilt during the rendering process. Some authors have proposed a new structure, called the *Bounding Interval Hierarchy* (BIH) to achieve this goal [Wächter and Keller 2006]. It is similar to a kd-tree, but it uses two bounding planes instead of one split plane per node. The result is a hierarchy that can be quickly rebuilt each new animation frame. The BIH performed slightly worse than a kd-tree for static scenes but significantly better for dynamic ones.

Another technique is directed towards structured movement. In this case, all animation key-frames are known prior to rendering. This enables the construction of a *Bounding Volume Hierarchy* (BVH) that can be adapted (deformed) while objects move in the scene [Wald et al. 2007]. The main idea is to keep the hierarchy topology while only deforming the node bounding volumes. Results show interactive frame rates for the entire animation. However, object movement is restricted. In the event of unpredicted motion, there is a risk that the hierarchy topology becomes invalid and must be entirely reconstructed. When this happens, rendering performance is impacted severely.

There are two other ideas that aim to support the general case of dynamic scenes. Both propose the reconstruction of the entire acceleration structure each frame. The first technique uses a *Uniform Grid* [Wald et al. 2006]. Although not so efficient for ray traversal, its simplicity equates to a fast rebuild scheme. The second proposal implements a kd-tree construction procedure in parallel, tapping into the processing power of modern multi-core CPUs [Shevtsov et al. 2006]. The two approaches are able to achieve interactive rendering rates even for non-structured animations.

2.2 Research on the GPU

With the increasing programmability of commodity graphics processing units, it is possible to perform more than the specific graphics computations for which they were designed. GPUs are now capable coprocessors, and their high speed makes them useful for a variety of applications that can be implemented in parallel. Today, general purpose computing on the GPU (GPGPU) is an active research field with several applications including signal processing, pattern matching, physics simulations and advanced rendering techniques such as ray tracing.

In spite of that, the GPU hardware still presents several challenges for implementing efficient algorithms and data structures for ray tracing. For instance, hardware resources are limited: there is a maximum number of registers that can be assigned per thread, memory accesses must be rationalized to avoid saturation of memory bandwidth, and programming features such as function recursion are not available. These and other factors limit GPU occupancy and parallelism.

One of the first successful ray tracing performed on the GPU used a

Uniform Grid [Purcell et al. 2002]. Due to hardware restrictions at the time, the authors were forced to design a streaming ray tracing algorithm. Each step, such as generating primary rays, was done by a different fragment shader. This computational model, although effective, proved to be seriously bandwidth limited. Each successive draw call incurred a great overhead in setup and memory transfers. The performance achieved was of a couple hundred thousand rays per second, while existing CPU implementations were already at millions of rays per second.

Since then, kd-trees have become very popular on the CPU for its great performance in several different scenes. Thus, succeeding GPU implementations attempted to harness the power of this structure. The main difficulty with traversing a kd-tree is the need of a per-ray traversal stack, in order to ascend and descend the tree.

Due to hardware restrictions in memory allocation and scatter operations, the first successful kd-tree implementation on the GPU present two algorithms for ray traversal that run without a stack, named *kd-restart* and *kd-backtrack* [Foley and Sugerman 2005]. Although both algorithms are asymptotically optimal, they both present a significant overhead. Rendering performance was of only a few hundred thousand rays per second, still lagging behind the CPU.

Later on, advances in graphics hardware programmability made it possible to write the entire ray tracing algorithm in one single program, thus avoiding several limitations of previous work. It was not until the most recent years that ray tracing on the GPU began achieving better performance than state of the art CPU implementations.

Further improvements on kd-tree based traversal were proposed [Horn et al. 2007]. The authors accelerate the *kd-restart* and *kd-backtrack* algorithms by modifying it with *packetization*, *push-down* and *short-stack* techniques. For the first time, a GPU ray-tracing technique was able to rival similar CPU implementations.

Another concurrent work presents a new stackless traversal algorithm [Popov et al. 2007]. It combines packet traversal with *ropes* between neighboring leaf nodes to avoid a stack. This new traversal scheme shows similar performance than the previously mentioned kd-tree implementations. At a later time, the same authors present another ray tracing implementation on the GPU, but for the first time using a BVH [Günther et al. 2007]. The acceleration structure is built on the CPU, while ray tracing is done on the graphics card using a stackless algorithm. Although BVHs are usually slower than kd-trees for ray tracing, the results show comparable performance. One advantage of a BVH implementation is a lower memory footprint, which allowed for the visualization of large models (more than ten million triangles) on the GPU for the first time.

To our knowledge, only one related research has demonstrated ray-tracing dynamic scenes on the GPU. This work was based on an efficient kd-tree reconstruction algorithm, fully implemented inside the graphics hardware [Zhou et al. 2008]. The entire tree structure was rebuilt each frame, if necessary. The tree traversal procedure used a small per-ray stack, implemented inside the GPU using the CUDA programming model [Nvidia 2008]. When ray tracing dynamic scenes, performance obtained surpassed the best CPU implementations. For the first time it was possible to obtain interactive frame rates for dynamic scenes together with illumination effects such as reflections and refractions.

In this work we propose a technique with a similar goal. Our procedure fully rebuilds the entire acceleration structure inside the GPU, while performing ray-traversal and shading computations. However, we have taken a fundamentally different approach. This latest state-of-the-art result uses a kd-tree, which is a spatial structure that is highly efficient for ray-traversal but also requires a highly complex construction procedure. Our strategy consists in using an acceleration structure that is simpler to be rebuilt inside the graphics hardware. It may not be able to achieve optimal ray-traversal performance, but its effectiveness has already been proven in other related research [Purcell et al. 2002, Wald et al. 2006]. The work presented here investigates whether it is more effective to trade ray-traversal performance for a faster structure rebuild.