Interactive k-D Tree GPU Raytracing

Daniel Reiter Horn   Jeremy Sugerman   Mike Houston   Pat Hanrahan

Stanford University *

Abstract

Over the past few years, the powerful computation rates and high memory bandwidth of GPUs have attracted efforts to run raytracing on GPUs. Our work extends Foley et al.'s GPU k-d tree research. We port their *kd-restart* algorithm from multi-pass, using CPU load balancing, to single pass, using current GPUs' branching and looping abilities. We introduce three optimizations: a packetized formulation, a technique for restarting partially down the tree instead of at the root, and a small, fixed-size stack that is checked before resorting to restart. Our optimized implementation achieves 15 - 18 million primary rays per second and 16 - 27 million shadow rays per second on our test scenes.

Our system also takes advantage of GPUs' strengths at rasterization and shading to offer a mode where rasterization replaces eye ray scene intersection, and primary hits and local shading are produced with standard Direct3D code. For 1024x1024 renderings of our scenes with shadows and Phong shading, we achieve 12-18 frames per second. Finally, we investigate the efficiency of our implementation relative to the computational resources of our GPUs and also compare it against conventional CPUs and the Cell processor, which both have been shown to raytrace well.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors I.2.2 [Computer Graphics]: Raytracing—Systems

Keywords: Programmable Graphics Hardware, Data Parallel Computing, Stream Computing, GPU Computing, Brook

1 Introduction

Raytracing is a workload with heavy computational requirements, but abundant parallelism. As a result, researchers have explored many novel custom and emerging architectures in pursuit of interactivity [Purcell et al. 2002; Woop et al. 2005; Benthin et al. 2006]. Our work extends the work of Foley et al. [2005] with algorithmic improvements and modifications for the new capabilities of current GPUs.

In their work, Foley et al. adopted k-d tree acceleration structures for GPU raytracing by introducing two stack-free traversal algorithms, but identify their performance as uncompetitive with optimized CPU implementations. Our implementation improves theirs by replacing their multi-pass approach with a single pass approach relying on the recent addition of looping and branching functionality to GPUs. We add three major enhancements to their *kd-restart* algorithm: packetization, *push-down*, and *short-stack*. Packetization combines CPU ray packet ideas [Wald et al. 2001] with restart, and takes advantage of the GPU's higher efficiency running four-wide SIMD code. *push-down* localizes rays to the sub-tree they overlap instead of always restarting from the tree root. *short-stack* adds a small, fixed-size stack to maintain the last N pushes, and falls back to restarting on underflow. The net impact is more than an order of magnitude performance improvement over Foley et al. and a traversal algorithm that in our scenes visits fewer than 3% more nodes as compared with the conventional stack-based approach.

Our system also seeks to exploit the GPU's natural strengths by generating primary hits via rasterization and computing local shading using standard programmable fragment shading, and traces rays for secondary effects such as shadows and specular bounces. For comparison with prior work and other architectures, we will also present results from casting primary rays with our system. Our system is implemented with a combination of Brook [Buck et al. 2004] for the computational kernels and Direct3D 9 [Microsoft 2003] for rasterization, shading, and display. We use ATI's CTM [ATI 2006b] toolkit to work around driver compiler bugs and gather statistics, but all of our GPU shader code is standard pixel-shader 3.0. On an ATI X1900 XTX [ATI 2006a], our 1024x1024 scenes with shadows and Phong shading render at 12-18 frames per second.

Finally, we analyze the performance of our implementation and compare it with work on x86 CPUs and the Sony/Toshiba/IBM Cell Broadband Engine Processor™ (Cell) [Benthin et al. 2006]. We identify the primary bottlenecks and structural inefficiencies of our environment as well as how architectural differences change some of the algorithmic options.

2 Related Work

There has been a significant interest in studying raytracing on parallel architectures. Benthin et al. [2006] designed a bounding volume hierarchy (BVH) raytracer on the Cell architecture using a software-managed cache to save bandwidth and software threads to hide the latency on a cache miss. Their raytracer performs nearly as well per core and per clock as a commodity x86, resulting in 60 million primary rays per second and 21 million primary plus shadow rays per second on the conference scene. This gives them clock-for-clock a 6.5x performance advantage over a single core x86. Sugerman et al. [2006] implemented a k-d tree raytracer on the Cell with a software managed cache, but without frustum culling or software threads. This version achieves 17.4 million primary rays per second on the Robots test scene, the same performance as the GPU raytracer we present here.

There also has been research on custom raytracing architectures. A recent example is the SaarCOR processor [Schmittler et al. 2002]. This chip features dedicated k-d tree

*{danielrh, yoel, mhouston, hanrahan}@graphics.stanford.edu

traversal and triangle intersection units and a fixed function shading unit. The RPU [Woop et al. 2005] improved upon the SaarCOR by enabling programmable shading and demonstrating an FPGA implementation. The initial RPU prototype operating at 66MHz provides comparable performance to an Intel P4 running at 2.66GHz. Unfortunately, this is not a commodity processor, nor is a full speed ASIC version available.

Raytracing on graphics hardware has already drawn significant research interest. Purcell et al. [2002] designed the first raytracer on a GPU to utilize an acceleration structure, in their case a uniform grid. Foley et al. [2005] applied a k-d tree acceleration structure to GPU raytracing and showed, that on graphics hardware, there are scenes for which a k-d tree yields far better performance than a uniform grid. Given the lack of addressable temporary storage within a fragment shader, the fastest programmable unit available on the GPU, Foley et al. had to develop two stackless k-d tree traversal methods. Likewise Thrane and Simonsen [2005] developed a fixed-order bounding-volume traversal method for ray intersection on the GPU to obviate the needs for a stack, and Carr et al. [2006] used a similar BVH structure to create a raytracer suited to dynamic geometry. However, none of the previous GPU raytracing efforts have been able to significantly outperform, ray for ray, the performance of comparable single-threaded CPU implementations for arbitrary test scenes. This has been due to short instruction limits forcing segmentation of program code, expensive memory transfers between those segments and lack of fine-grained conditional execution. As the architectures continue to mature, these limitations have slowly become less severe, which allows us to exploit new algorithmic changes.

3 Algorithm

A typical k-d tree traversal algorithm takes a k-d tree, a binary tree of axis aligned splitting planes that partition space, and a ray as input and returns the first triangle the ray intersects. Traversal walks the ray through the tree so that it visits leaf nodes, and hence triangles, in front to back order. Rather than run recursively, optimized implementations maintain a stack data structure. When a ray passes through both sides of a splitting plane, the "far" subtree is pushed onto the stack and the algorithm first traverses the "close" subtree. Wald's thesis [Wald 2004] has a detailed description of the process, summarized in psuedocode below:

```
stack.push(root,sceneMin,sceneMax)
tHit=infinity
while (not stack.empty()):
   (node,tMin,tMax)=stack.pop()
   while (not node.isLeaf()):
      a = node.axis
      tSplit = ( node.value - ray.origin[a] ) / ray.direction[a]
      (first, sec) = order(ray.direction[a], node.left, node.right)
      if( tSplit ≥ tMax or tSplit < 0)
         node=first
      else if( tSplit ≤ tMin)
         node=second
      else
         stack.push( sec, tSplit, tMax)
         node=first
         tMax=tSplit
   for tri in node.triangles():
      tHit=min(tHit,tri.Intersect(ray))
      if tHit<tMax:
         return tHit //early exit
return tHit
```

As explained in previous work by Foley et al. [2005], the stack requirement of the standard k-d tree algorithm adapts poorly to GPUs. The recently added ability for GPU's to perform discontinuous writes to memory bypasses any caching and is vastly slower than cached writes on a CPU. Even if a stack were fast enough, its resource requirements would be impractical: enough memory for the deepest ray's needs multiplied by the total number of rays traversing in parallel. It is possible to render a subset of the rays at a time and reuse a stack buffer, but GPUs only run efficiently with thousands of rays traversing at once.

Instead, Foley et al. introduce two stackless traversal algorithms: *kd-restart* and *kd-backtrack*. Backtracking has tighter asymptotic bounds, but requires large auxiliary data structures (a six word bounding box per node in addition to the single word splitting plane) and correspondingly bandwidth hungry backtrack logic during traversal. Additionally there is no known method to add packets to the backtracking algorithm. On the other hand, *kd-restart* actually requires less code and no extra data structures. As shown in Appendix A.2, whenever a ray emerges from a leaf having hit nothing, the algorithm just steps the current ray forward by updating tMin, the variable marking the origin of the ray, to its previous endpoint, tMax. Then it sets the endpoint, tMax, to the boundary of the scene and *restarts* traversal from the top of the tree. Because of its simplicity and the potential to packetize it, we focused on improving *kd-restart*. Our improvements avoid visiting most of the extra nodes that restart would visit when compared to a standard k-d tree algorithm.

3.1 Packets

One of the most significant innovations in interactive raytracing was the introduction of packets [Wald et al. 2001]. In their simplest form, packets trace four rays at a time to take advantage of the four-wide SIMD instructions on modern CPUs. Beyond that, however, adjusting the packet size allows the programmer to trade-off amortizing the control and bookkeeping of traversal against the extra nodes visited by forcing all the rays in each packet to travel together. Additionally, further optimizations such as frustum culling even save ray-triangle intersections and become possible when using packets [Reshetov et al. 2005].

Our first *kd-restart* modification is the introduction of packets. The significant complication beyond standard packetization is that restart requirements constrain us from freely modifying tMax when individual rays go inactive. To step the ray forward at the restart point, we set the new tMin to the previous step's tMax. With packets, we add a liveness mask to each ray so that the mask is always refined based on whether that ray should traverse a given child. Only upon restarting is the liveness mask reinitialized from tMin and tMax. Thus it is straightforward to add packetization to the standard *kd-restart* algorithm. Just like conventional implementations, packets naturally increase our utilization of math units and dilute the per-ray impact of the conditional logic.

3.2 Push-Down

Often the intersection of a ray with the scene volume only passes through a subtree of the entire k-d tree. Instead of restarting at the root, such rays only need to back up to the node that is the root of the lowest subtree enclosing them. Thus, as a ray descends from the root, so long as it only

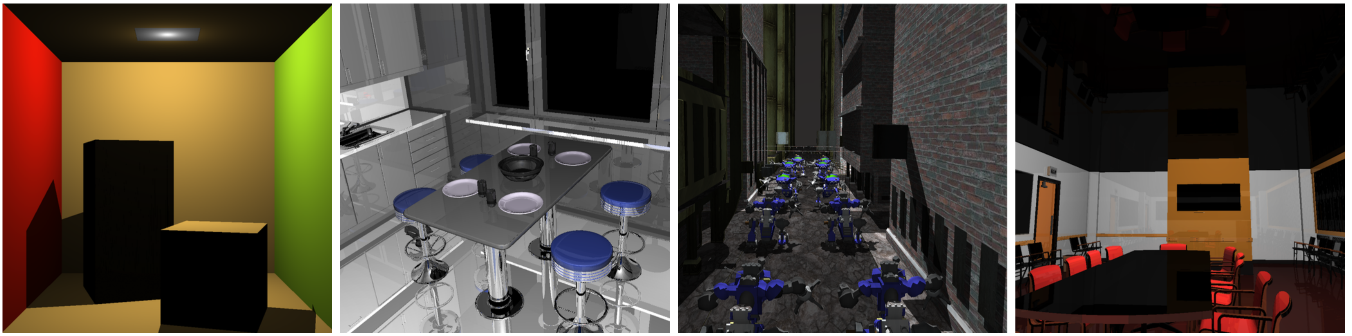

Figure 1: Our test scenes: Cornell Box, Robots, Kitchen, Conference Room.

encounters nodes whose splitting planes it does not cross (i.e. it remains strictly in the near subtree or strictly in the far subtree), it is safe to continue moving the restart location down. Because the traversal algorithm already has separate cases to handle rays strictly in one child and rays that span both children, implementing this optimization, which we call *push-down*, is straightforward.

The major limitation of this optimization is that it is no longer beneficial as soon as traversal encounters the first node a ray could conceivably span, regardless of how unlikely the ray is to make it to the far side before terminating. That makes the benefits of *push-down* somewhat fragile. Rays from a camera or light whose frustum happens to cross one of the early splitting planes will get no benefit and pay a slight penalty for the extra bookkeeping. Additionally, it only eliminates traversal steps near the top of the tree, which are the ones most shared and coherent across neighboring rays.

3.3 Short-Stack

Foley et al. [2005] assert that a stack requires storage proportional to the maximum stack depth times the number of rays. However, we observe that it is possible to use a stack of bounded size, and fall back to a stackless algorithm if that stack should underflow. Thus, *short-stack* introduces a small, fixed-size stack whose manipulation is modified in two ways. Pushing a new node when the stack is full is acceptable, and discards the bottom-most entry. Popping from an empty stack no longer terminates traversal, but instead triggers a restart. Effectively, the stack serves as a cache that can be used to trade-off the amount of per-ray state versus frequency of restarts.

short-stack is complementary to *push-down*, but it is more robust and useful. The stack only degrades once it fills without the sensitivity to ray directions that plagues *push-down*. Also, restarting has the most overhead when a ray is near the bottom of the tree and must restart to access a neighbor node. *short-stack* eliminates the overhead of any restarts that would have happened within N pushes of the bottom of the tree. *push-down* advances the restart point to the node that would have been the first entry on a complete stack. So, by the time a restart occurs with *push-down*, the number of extra nodes traversed is relatively small. Finally, converting traversal from single rays to ray packets often prompts a few extra traversals near the leaves as the packet spreads out relative to node size. The small stack captures these extra traversals well, whereas the standard restart algorithm would visit many extra nodes.

Appendix A shows the step by step modifications beginning with the original k-d tree code (A.1) and progressing to the version with all our optimizations applied (A.5).

4 Implementation

We implemented the optimizations from Section 3 in a renderer for an ATI X1900XTX GPU. While the renderer can trace rays, it can also rasterize and shade like a traditional 3D GPU application. Our implementation uses the Brook [2004] runtime and stock Direct3D 9.0c, and the GPU fragment programs for raytracing are written in Brook or pixel-shader 3.0 assembly and run with version 6.7 of the ATI Catalyst drivers. For the non-packetized ray-scene intersection, each GPU fragment traces a single ray through the scene. For the packetized case, each fragment traces four rays at once. Unfortunately, every driver version tried miscompiles some of our raytracing fragment programs when translating them to the board's native assembly, and no version produced a working *short-stack* program for this GPU. As a workaround, we used ATI's CTM toolkit to compile offline and hand-patched the assembly and implemented our system with an option to pass the altered shaders directly to the GPU.

We selected the parameters of our implementation to best suit our hardware. The k-d tree is built using a surface area heuristic [Havran and Bittner 2002], but with an estimated cost of triangle intersection equal to the estimated cost of traversing one node. This is because the estimated cost of intersection is equal to the cost of traversal based on instruction counts and timing numbers of our implementation. Direct3D 9.0c limits the numbers of available resources per fragment to 32 four-wide floating point registers, so we can only fit four rays per packet with a three entry stack.

Our system renders as follows: we construct a vertex buffer and texture containing the scene geometry, and textures with the k-d tree nodes and material properties of the primitives. We rasterize the scene and produce a buffer of hits using a shader that pulls the (x, y, z) location from the rasterizer and depth buffer and computes the incoming direction from the camera parameters. Alternatively, we can rasterize a single rectangle the size of the requested image with programs bound to generate and intersect eye rays to produce raytraced initial hits for comparison. From this point, the renderer generates and traces shadow or specular rays as appropriate. For shadow rays, it loops through each light and runs multiple passes that generates a shadow ray for each hit, intersect that ray against the k-d tree, and computes local Phong shading for the hits not in shadow. For specular rays, it runs passes which generate bounce rays, in-

	Shadow Single	Shadow Packet	1 Bounce Single	1 Bounce Packet
Cornell Box	18.7	28.1	19.6	27.5
Kitchen	14.3	18.3	8.3	8.6
Robots	10.0	12.1	5.2	6.6
Conference	12.1	14.2	6.0	6.1

Table 1: Frame rates (fps) rendering our scenes with shadows or with a specular bounce at 1024x1024. Primary hits are generated via rasterization of triangles.

tersects them against the k-d tree, and produces a new buffer of hits. These hits can be Phong shaded and accumulated, with weights, into the frame buffer location that generated the initial bounce, and they can be fed back into the same sequence of shaders to generate further rays.

Currently, our implementation is built to favor pluggability and simplicity over end to end performance, and could be optimized in a number of ways that we address in Section 6. However, it is important to emphasize that local shading is intentionally kept distinct from ray generation or intersection. This allows local shaders to be written very similarly to existing GPU shaders and run correspondingly efficiently.

5 Performance

We evaluated our system with an ATI Radeon X1900 XTX 512MB [ATI 2006a] running with a 2.4 GHz Core2 Duo [Intel 2006]. We configured the board to run with Catalyst 6.7 drivers and the full 650 MHz ALU clock and 750 MHz memory clock it uses for fullscreen applications. We tested with 1024x1024 images of the four scenes shown in Figure 1: a trivial Cornell Box with 32 triangles, the robots and kitchen scenes from Foley et al. [2005] with 71,708 and 110,561 triangles respectively, and the conference room scene with 282,801 triangles from Benthin et al. [2006].

5.1 End-to-End Performance

Table 1 lists the speed of our implementation at rendering complete images including either shadows or specular bounces. While our shaders for copying rasterizer output into ray tracing and shading ray hits are unoptimized and designed for flexibility, ray-scene intersection still dominates rendering time. The packet tracer does significantly better for the shadow images, but packets are nearly performance neutral for one specular bounce.

5.1.1 Primary and Shadow Rays

In order to evaluate our improvements separately and compare against prior work, we rendered using primary rays instead of rasterization and measured the raw ray-scene intersection rate as we applied our optimizations. The results are laid out in Table 2 and follow roughly the same behavior for primary and shadow rays.

As predicted in Foley et al. [2005], replacing the multipass implementation with a single pass looping implementation provided a large increase in performance. Our GPU has 3.75x the computational resources of theirs, but our unoptimized, single-pass restart implementation is more than *25 times* as fast for the same scenes. Our algorithmic optimizations do not deliver comparable improvements, but still improve our performance by a combined 65 - 130 percent.

Four-wide ray packets per fragment are as relevant on this GPU as on CPUs, as both architectures provide 4-way SIMD

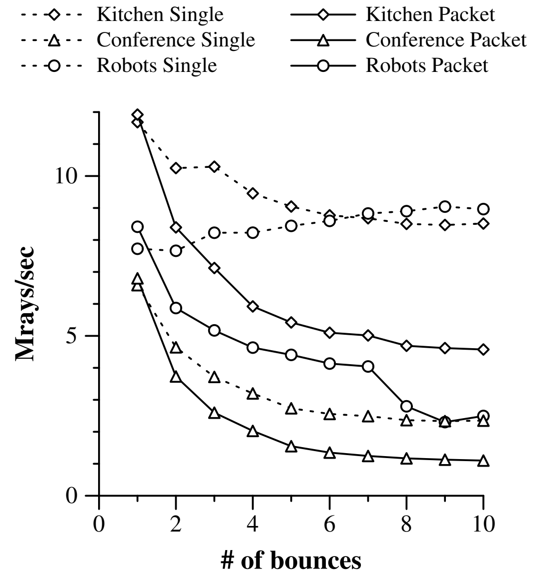

Figure 2: Performance of specular bounces, with and without packets. As the number of bounces increases, the rays diverge in execution and performance decreases. Eventually, maximal divergence causes performance to become nearly constant.

instructions. While the X1900 can pair two scalar operations per pipe per clock, it is still only half as efficient as having a complete four-wide instruction.

push-down offers a small boost to three of the scenes, but not for the kitchen. In the kitchen, the camera is nearly lined up with one of the early splits in the k-d tree. That prevents any appreciably smaller subtree from bounding the portion of the scene that overlaps most of the viewing frustum. *short-stack*, on the other hand, helps all of our scenes. We instrumented the code to count the number of nodes visited with each setup. For the three complex scenes *push-down* visited 3 - 22% fewer nodes than *kd-restart* while *short-stack* reduced counts an *additional* 48 - 52% demonstrating its superior robustness and effectiveness. We ran a simulation of *short-stack* that ignored register limits and used a deep enough stack to capture all activity. The actual *short-stack* traversals visited fewer than 3% more nodes than the unlimited stack for our test scenes.

5.1.2 Specular Rays

Figure 2 shows a graph of the performance of specular rays as a function of the number of bounces. Their performance is significantly worse than shadow and primary rays. The most important difference is that with bounces, no logical frustum rooted at camera or light sources encloses all the rays. Thus, the rays diverge more across the scene. This affects both the locality of their access to scene data and the divergence of execution among fragments representing nearby image pixels.

CPUs suffer from similar effects with secondary rays, but the impacts are amplified on GPUs. While GPUs do support conditional branching, they only support it efficiently when neighboring fragments branch in large groups [Buck 2005; Stanford University 2006]. So, while divergence lowers the number of active rays per packet on CPUs, it also lowers the number of active fragments across a much wider group on GPUs. That leads to the non-packet performance near-

	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

Table 2: Rates for rays, in millions per second, rendering our test scenes at 1024x1024 with successive optimizations enabled. Each line includes the optimizations above it.

ing packet performance for the first bounce and overtaking it for subsequent bounces. When each fragment represents four rays, the divergence across the fragments to be scheduled at any given time covers four times as many rays, and is correspondingly more divergent. Additionally, the GPU performs many times better with coherent than with random texture access [Stanford University 2006], and as the rays bounce their memory references become more incoherent.

5.2 Hardware Performance

5.2.1 GPU Efficiency

We also analyzed how efficiently our raytracer used the GPU's resources. While the rates in Section 5.1.1 are many times faster than prior results, we are below peak rates available on the hardware. The X1900 XTX has 16 pixel pipelines clocked at 650 MHz, each with 3 four-wide ALUs, giving it an execution rate of 31.2 GInstructions/second (4-wide SIMD) when fully utilized. When we instrumented our code and determined its instruction issue rate, we found that for the robots and conference scenes the non-packet tracer achieves 66 - 75% of the board's instruction rate and the packet tracer achieves 35 - 42%. Note that when packet tracing each instruction performs four operations, so packet tracing can still trace rays in less time, even with a lower instruction issue rate.

Our code is clearly stalling and bottlenecked on something other than compute power. We identified four possibilities for why our code does not obtain peak instruction issue rate:

- Insufficient fragments / independent instructions to cover dependent math
- Insufficient fragments to cover the latency of fetching values from textures / texture caches
- Insufficient bandwidth to fetch all the texture data for all the fragments
- Insufficient fragments executing the same instruction to fill the board

There are enough fragments (rays or packets) and independent instructions that we believe stalls due to dependent math are highly unlikely. We eliminated both bandwidth and latency effects by repeating our rendering with the memory clock reduced from 750 MHz to 500 MHz. Despite the 1/3 reduction in memory speed, system performance dropped by only a few percent, so the board is successfully hiding memory latencies and is not sensitive to bandwidth.

That left divergent execution as the remaining candidate performance bottleneck. Recall from Section 5.1.2 that GPUs only support branches efficiently when large groups of fragments run with identical instruction pointers. That is, they branch the same way at the same time. We know that while nearby rays follow similar paths, they are not identical and divergence is fairly common. This motivated the z-buffer based culling in both Foley et al.[2005] and Purcell et al.[2002]. Also recall that toggling from single rays to packets widens the bounding frustum of a fixed size group of fragments and therefore increases the probability of divergence.

To test this hypothesis, we modified our original fragment shader assembly via CTM, and replaced the body of the intersection and traversal loops, including ALU operations and memory fetches, with an equivalent number of "MOV" instructions. We then fed the kernel the same sequence of control logic so the modified shader would execute the same number of instructions and take the same branching pattern as the original, without the effects of memory read patterns. The execution rate of this test very closely matched the execution rate of the operational raytracing kernel. This leads us to conclude that incoherent branching is therefore our current performance bottleneck.

5.2.2 CPU & Cell Comparison

Raytracing has been shown to run efficiently on conventional CPUs [Wald 2004] and on the Cell processor [Benthin et al. 2006]. The Cell system from [Benthin et al. 2006] can issue 19.2 four-wide GInstr/s, about 62% the rate of our X1900 XTX, but casts 57.2 million primary rays per second compared to our 15.2 for similar renderings of the conference room scene. Their single 2.4 GHz Opteron reaches 8.7 million primary rays per second. Rasterizing primary hits and using GPU shading increases our end-to-end performance, and our render rates for full images with only shadows are 60% of the Cell and four times the speed of the Opteron, but there are clearly opportunities for making the central raytracing more competitive on GPUs.

There are two potential avenues for improvement: enabling better implementations of our current algorithms and enabling better algorithms. Direct3D 10 capable GPUs will deliver one straightforward improvement: integer types and instructions [Blythe 2006]. The control logic and bit operations manipulating our k-d tree nodes are bulky, implemented as floating point operations, and increase the cost of our inner loop. Of course, the biggest boost to the current algorithm would be a GPU less affected by execution divergence, but this would potentially require a large change in architecture design. Nevertheless, on both CPUs and Cell, each hardware thread executes independently of the others whereas our GPU requires every fragment that executes in a given clock to have the same instruction pointer.

One significant algorithmic improvement used on other architectures is larger packets. Cell, the x86, and the X1900 XTX all have four-wide math units, but as described in Section 3, the advantages of packets extend beyond occupying each SIMD lane. As a result, [Benthin et al. 2006] operated on 64 ray packets on Cell and [Sugerman et al. 2006]

employed 16 ray packets on both an x86 and Cell. However, larger packets are problematic on the GPU for two reasons. We are already near the Direct3D 9 limit on registers per fragment program. Because the GPU hides latency by running large numbers of fragments, increasing register availability for fragment programs either requires a large increase in the on-board register file or reduces the number of fragments that can be executed at once for covering latencies. And, while we might be able to grow the packet size by shrinking the stack, not only would is impair the effectiveness of *short-stack*, it would most likely make the code more divergent branch limited just as current packets are more limited than single rays.

6 Discussion

Our implementation obtains a major speedup over previous work when tracing rays on a GPU, sustaining over 15 million rays per second on our test scenes. We were able to demonstrate interactive performance at high resolutions with complex scenes on a GPU. We introduce optimizations to the *kd-restart* algorithm that limit it to nearly the same number of traversal steps as the conventional stack implementation, while still keeping the per-fragment state small enough to fit in registers. We combine a looping single pass implementation with packets to remove the bandwidth recirculation problems of [Foley and Sugerman 2005], exploit the four-wide math units of our GPU, and produce a GPU raytracer that is no longer memory bound.

In addition to these algorithmic improvements, which makes our implementation competitive to other highly tuned raytracers, we believe significant performance speedups can be realized with improved software engineering. For example, the rasterization pass uses a single monolithic vertex buffer and depth test without any sorting, culling, or other strategies often employed in interactive 3D applications. Also, all of the ray generation runs in separate passes from intersection. Shadow rays use the full intersection kernel instead of terminating after the first hit and a set of passes per light rather than a loop that handles all the lights. Without packets, and with a statically predetermined configuration of shadow and specular rays, all the hits for local shading and weighted accumulation could be produced with a single (long) shader bound during rasterization. Packets always need one intermediate pass to combine independently rasterized pixels into packet-sized chunks.

For ray-scene intersection, our current limiting performance constraint is execution coherence. There is an effective 'SIMD width' across fragments that leaves functional units idle unless there are enough fragments (48 on the X1900XTX) whose execution is at the same point. While raytracing is coherent among nearby rays, there is still divergence between fragments, especially with secondary rays or when each fragment is working on a packet of rays at a time. We can compensate by building a k-d tree whose cost metric encourages larger leaves (traversal and intersection are closer to the same cost on GPUs than CPUs already), but execution divergence still limits performance to around 40% of the board's issue rate.

We have identified two architectural obstacles that limit the efficiency of raytracing on current GPUs. As described above, GPUs only realize their computation potential when their workloads are not only parallel, but significantly SIMD. Both the x86 and Cell processor require four-wide instructions for peak utilization, but the X1900 XTX is not only four-wide SIMD per execution unit, but also 48 wide across execution units per clock, and other GPUs have similar large branching coherence requirements. Also, where CPUs hide memory latency with caches and Cell with an explicit DMA engine, GPUs employ parallelism that keeps hundreds of fragments in flight at a time. This restricts each fragment to a small footprint for all state. Our restart optimizations compensate for the resulting unsuitability of per-fragment stacks, but we have not found a way to fit the 16 - 64 rays per packet that effectively amortize intersection costs on other architectures. Even if larger packets fit, they would exacerbate the execution divergence problem.

The most significant upcoming change to GPU programming is the transition to Direct3D 10 [Blythe 2006] and correspondingly capable hardware. A few of its new features offer potential for minor enhancements. As mentioned in Section 5.2.2, integer support saves a few frequently executed instructions needed to handle control logic and bit manipulation in floating point. The larger indexable constant banks could potentially cache the top of the k-d tree or a small portion of scene data and might prove faster than texture memory. The larger register limits, in theory ease the restrictions on stack depth and packet size, but in practice their implementation is virtual and causes shaders that consume more registers to be proportionally limited in parallelism and therefore performance. Overall, these changes do not enable any obvious major improvements nor any easing of our most significant barrier: the divergence penalty.

A key next step in our system is to incorporate realistic surface shading. Shading is a dominant component of modern rendering, be it rasterized or raytraced. By design, GPUs excel at texturing, bump mapping, and other shading effects pervasive in modern 3D applications. As shading becomes more complex, GPUs become compelling architectures for complete raytracing systems because of their ability to use rasterization for primary rays, raytracing for secondary effects, and the shading horsepower of modern GPUs.

7 Acknowledgments

We would like to thank Tim Foley for his ideas and previous GPU raytracing work. Additionally we would like to thank Ian Buck, Mark Segal and Derek Gerstmann for developing and supporting the GPU abstractions underneath our implementation. This research was supported by the US Department of Energy (contract B554874-2), the Rambus Stanford Graduate Fellowship, the ATI Fellowship Program, and the Intel PhD Fellowship Program.

References

ATI, 2006. Radeon X1900 product site. http://ati.amd.com/products/radeonx1900/index.html.

ATI, 2006. Researcher CTM documentation. http://ati.amd.com/companyinfo/researcher/documents.html.

Benthin, C., Wald, I., Scherbaum, M., and Friedrich, H. 2006. Ray Tracing on the CELL Processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*.

Blythe, D. 2006. The Direct3D 10 system. *ACM Trans. Graph. 25*, 3, 724–734.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. 2004. Brook for GPUs: Stream computing on graphics hardware. In *Proceedings of ACM SIGGRAPH 2004*.

BUCK, I. 2005. GPU computation strategies. In *GPGPU Course Notes - SIGGRAPH 2005*.

CARR, N. A., HOBEROCK, J., CRANE, K., AND HART, J. C. 2006. Fast gpu ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006*, Canadian Information Processing Society.

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 Press, New York, NY, USA, 15–22.

HAVRAN, V., AND BITTNER, J. 2002. On improving kd-trees for ray shooting. In *Proceedings of WSCG'2002 conference*, 209–217.

INTEL, 2006. Intel Core2 Duo Processor. http://www.intel.com/products/processor/core2duo .

MICROSOFT, 2003. Directx home page. http://www.microsoft.com/windows/directx/default.asp .

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

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. *ACM Trans. Graph. 24*, 3, 1176–1185.

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.

STANFORD UNIVERSITY, 2006. GPUBench. http://graphics.stanford.edu/projects/gpubench .

SUGERMAN, J., FOLEY, T., YOSHIOKA, S., AND HANRAHAN, P., 2006. Ray tracing on a cell processor with software caching. Poster at The 2006 IEEE Symposium on Interactive Ray Tracing. http://www.sci.utah.edu/RT06/full_compendium.pdf.

THRANE, N., AND SIMONSEN, L. O. 2005. *A comparison of acceleration structures for GPU assisted ray tracing.* M.S. thesis, University of Aarhus, Aarhus, Denmark.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3, 153–164.

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph. 24*, 3, 434–444.

A Code for Restart Optimizations

Figures A.1 through A.5 display the progressive modifications to standard k-d tree traversal as the optimizations from Section 3 are applied. Unlike the text, packets are applied last to make the differences easiest to follow.

A.1 Standard stack k-d tree traversal

```
stack.push(root,sceneMin,sceneMax)
tHit=infinity
while (not stack.empty()):
   (node,tMin,tMax)=stack.pop()
   while (not node.isLeaf()):
      a = node.axis
      tSplit = ( node.value - ray.origin[a] ) / ray.direction[a]
      (first, sec) = order(ray.direction[a], node.left, node.right)
      if( tSplit ≥ tMax or tSplit < 0)
         node=first
      else if( tSplit ≤ tMin)
         node=second
      else
         stack.push( sec, tSplit, tMax)
         node=first
         tMax=tSplit
   for tri in node.triangles():
      tHit=min(tHit,tri.Intersect(ray))
      if tHit<tMax:
         return tHit //early exit
return tHit
```

A.2 Modifications for *kd-restart* traversal

```
tMin=tMax=sceneMin
tHit=infinity
while (tMax<sceneMax):
   node=root
   tMin=tMax
   tMax=sceneMax
   while (not node.isLeaf()):
      a = node.axis
      tSplit = ( node.value - ray.origin[a] ) / ray.direction[a]
      (first, sec) = order(ray.direction[a], node.left, node.right)
      if( tSplit ≥ tMax or tSplit < 0)
         node=first
      else if( tSplit ≤ tMin)
         node=second
      else
         /* Remove:stack.push(sec, tSplit, tMax) */
         node=first
         tMax=tSplit
   for tri in node.triangles():
      tHit=min(tHit,tri.Intersect(ray))
      if tHit<tMax:
         return tHit //early exit
return tHit
```

A.3 Modifications for *push-down* traversal

```
tMin=tMax=sceneMin tHit=infinity
while (tMax<sceneMax):
   node=root
   tMin=tMax
   tMax=sceneMax
   pushdown=True
   while (not node.isLeaf()):
      a = node.axis
      tSplit = ( node.value - ray.origin[a] ) / ray.direction[a]
      (first, sec) = order(ray.direction[a], node.left, node.right)
      if( tSplit ≥ tMax or tSplit < 0)
         node=first
      else if( tSplit ≤ tMin)
         node=second
      else
         node=first
         tMax=tSplit
         pushdown=False
      if pushdown:
         root=node
   for tri in node.triangles():
      tHit=min(tHit,tri.Intersect(ray))
      if tHit<tMax:
         return tHit //early exit
return tHit
```

A.4 Modifications for *short-stack* traversal

```
tMin=tMax=sceneMin tHit=infinity
while (tMax<sceneMax):
   if stack.empty():
      node=root
      tMin=tMax
      tMax=sceneMax
      pushdown=True
   else:
      (node,tMin,tMax)=stack.pop()
      pushdown=False
   while (not node.isLeaf()):
      a = node.axis
      tSplit = ( node.value - ray.origin[a] ) / ray.direction[a]
      (first, sec) = order(ray.direction[a], node.left, node.right)
      if( tSplit ≥ tMax or tSplit < 0)
         node=first
      else if( tSplit ≤ tMin)
         node=second
      else
         stack.push(sec,tSplit,tMax)
         node=first
         tMax=tSplit
         pushdown=False
      if pushdown:
         root=node
   for tri in node.triangles():
      tHit=min(tHit,tri.Intersect(ray))
      if tHit<tMax:
         return tHit //early exit
return tHit
```

A.5 Modifications for packetized *short-stack* traversal

```
tMin_v=sceneMin
tMax_v=sceneMax
tHit_v=infinity
done_v=False
didstack=False
while ((didstack or not all(done_v)) and not all(tHit_v<tMax_v)):
   node=root
   if not stack.empty():
      didstack=True
      (node_v,tMin_v,tMax_v,live_v)=stack.pop()
      pushdown=False
   else:
      didstack=False
      node=root
      live=tMin_v<=tMax_v and not done_v
      pushdown=True
   while (not node.isLeaf()):
      a = node.axis
      tSplit_v = ( node.value - ray_v.org[a] ) / ray_v.dir[a]
      (first, sec) = order(ray_0.dir[a], node.left, node.right)
      wantNear_v=tSplit_v>tMin_v and live_v
      wantFar_v=tSplit_v ≤tMax_v and live_v
      if(all(wantNear_v or not live_v) and not any (wantFar_v))
         node=first
      else if(all(wantFar_v or not live_v) and not any(wantNear_v))
         node=second
      else
         pushdown=False
         node=first
         top_live_v=live_v and wantFar_v
         top_tMin_v=top_live_v?max(tMin_v,tSplit_v):tMin_v
         stack.push(second,top_tMin_v,tMax_v,top_live_v)
         live_v=wantNear_v
         tMax_v=wantNear_v ?  min(tSplit_v,tMax_v) :  tMax_v
      if pushdown:
         root=node
   for tri in node.triangles():
      tHit_v=min(tHit_v,tri.Intersect(ray))
      if all(tHit_v<tMax_v)
         return tHit_v //early exit
   done_v=done_v or (tHit_v<=tMax_v)
   tMin_v=tMax_v
   tMax_v=sceneMax
return tHit_v
```