Packet-based Whitted and Distribution Ray Tracing

Solomon Boulos† Dave Edwards† J. Dylan Lacewell† Joe Kniss‡ Jan Kautz⋆ Peter Shirley† Ingo Wald†
†University of Utah ‡University of New Mexico ⋆University College London

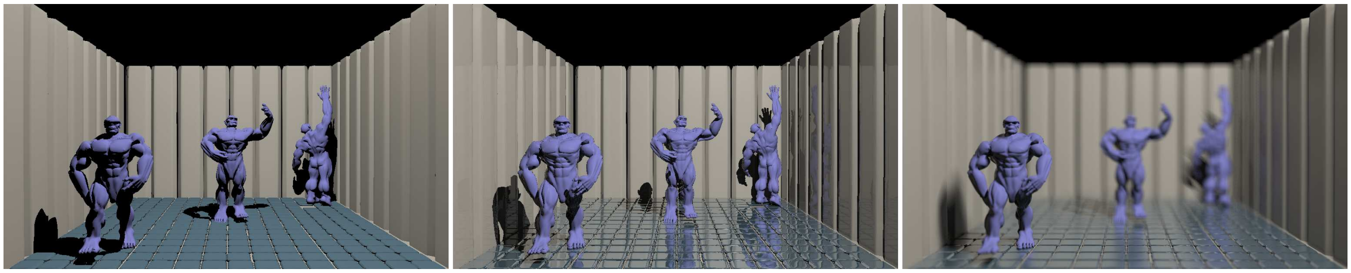

Figure 1: Left: ray casting with shadows (RCS). Middle: Whitted-style ray tracing (WRT). Right: distribution ray tracing (DRT) with 64 samples per pixel. This paper investigates interactive WRT on current hardware and the prospects for interactive DRT on future hardware.

Abstract

Much progress has been made toward interactive ray tracing, but most research has focused specifically on ray casting. A common approach is to use "packets" of rays to amortize cost across sets of rays. Little is known about how well packet-based techniques will work for reflection and refraction rays, which do not share common origins, and often have less directional coherence than viewing and shadow rays. Since the primary advantage of ray tracing over rasterization is the computation of global effects, such as accurate reflection and refraction, this lack of knowledge should be corrected. Our ultimate goal is to achieve interactive distribution ray tracing with randomized rays for glossy reflections, soft shadows, motion blur and depth of field. But it is not clear that the randomization would not further erode the effectiveness of techniques used to accelerate ray casting. This paper addresses the question of whether packet-based ray algorithms can be effectively used for more than visibility computation. It is shown that with the appropriate choice of data structure and packet assembly algorithm, useful algorithms for ray casting do indeed extend to both Whitted-style and distribution ray tracing programs.

1 Introduction

While some researchers predict that ray tracing will replace rasterization as the underlying algorithm for desktop graphics, others believe this will not happen in our lifetime [SIGGRAPH 2002]. Ray tracing has a number of advantages over rasterization, including automatic visibility culling, time complexity sub-linear in the number of objects, and ability to take advantage of multi-core architectures. Perhaps the most important advantage of ray tracing over rasterization is that it offers higher-quality images when "secondary" rays (e.g., for reflection and refraction) are used. The main drawback of ray tracing is that it is currently slower than hardware-based rasterization for most scenes. In this work, we investigate the practicality of interactive ray tracing with secondary rays, such as reflection and refraction. We also explore the future practicality of interactive distribution ray tracing.

One problem with discussing interactive ray tracing is that *ray tracing* is an overloaded term. In this paper, we use the term *ray casting* to refer to the use of ray tracing for visibility computations only (RCS, Figure 1, left). By adding direct lighting, reflection, and refraction to ray casting, we can implement Whitted's [1980] well-known algorithm; hence we refer to such a method as *Whitted-style ray tracing* (WRT, Figure 1, middle). The next step beyond WRT is *distribution ray tracing* (DRT, Figure 1, right), developed by Cook [1984]. A DRT renderer uses multiple primary rays per pixel to render non-singular effects such as depth of field, glossy reflection, motion blur, and soft shadows.

Recently, interactive ray tracing has been a popular topic for research. There are several current systems that can perform interactive ray casting; some of these implement simple shading by computing direct lighting from point sources. However, very few of these programs implement full WRT. One reason for this limitation is that most interactive ray casters trace packets of rays with shared ray origins, and reflection and refraction rays cannot be placed in such packets. The little evidence that exists about the performance of secondary ray packets is not encouraging [Reshetov 2006], so WRT may not be able to take advantage of the techniques that have proven so effective for ray casting.

Despite the uncertain outlook for interactive WRT performance, we believe that rendering with WRT rather than simple ray casting is an important goal. Although ray casting is faster than WRT, it is inferior to current GPU graphics in both performance and image quality. To get out of this "worst of both worlds" situation, interactive ray tracing will need to add the features of full WRT. In this paper, we discuss a new method for interactive WRT using a combination of generalized ray packets and a bounding volume hierarchy to improve efficiency, and show that our system can run at interactive rates on current high-end computers. We also examine the overall impact of reflection and refraction rays on rendering performance, and extend these measurements to the types of secondary rays associated with full DRT. Based on our findings, we believe that a WRT-based renderer can provide interactive performance now, and an interactive DRT-based renderer will be possible within a decade on general purpose hardware, and much sooner if ray tracing hardware is built.

2 Background

Ray Casting Interactive ray casting has been extensively studied. Different ray casting software designs have targeted shared memory computers [Muuss 1987; Parker et al. 1999; Bigler et al. 2006], clusters [Wald 2004; DeMarle et al. 2003], traditional GPUs [Purcell et al. 2002; Foley and Sugerman 2005], and Cell processors [Benthin et al. 2006]. Another approach has been to abandon rasterization-based GPUs and implement ray casting directly in hardware. FPGA and ASIC designs for such ray casting hardware have been presented [Schmittler et al. 2002; Fender and Rose 2003; Woop et al. 2005; Woop et al. 2006]. One common technique for accelerating ray casting is grouping rays into packets to take advantage of coherence. Ray packets allow further efficiency through the use of SIMD instructions as well as packet-based culling using interval arithmetic or frustum tests [Wald et al. 2001; Reshetov et al. 2005; Boulos et al. 2006b; Wald et al. 2006a; Wald et al. 2006b; Lauterbach et al. 2006].

Dynamic Scenes Some work has also addressed ray casting for dynamic scenes, which is required if ray tracing is ever to be practical for games. Most of this work is intimately tied to the type of spatial acceleration structure used. For grids, both incremental [Reinhard et al. 2000] and complete [Ize et al. 2006] rebuilding strategies have been proposed. Motion decomposition may be used to build good *k*d trees for dynamic models if the full space of poses is known in advance [Günther et al. 2006]. Bounding volume hierarchies have been improved using incremental rebuilding schemes borrowed from collision detection [Larsson and Akenine-Möller 2003; Wald et al. 2006a; Lauterbach et al. 2006].

Shadow Rays Simple shading is often added to ray casting by computing direct lighting from point light sources. This is a simple extension of ray casting, since determining direct lighting from a point source is analogous to computing visibility from a pinhole camera. Shadow rays can therefore be traced from the light source using exactly the same techniques as primary rays from the camera [Wald et al. 2006a]. A similar argument applies to computing soft shadows at a single point, for example, by sending 16 shadow rays per primary ray [Parker et al. 1999].

Whitted-style Ray Tracing As with ray casters, most interactive Whitted-style ray tracers use ray packets to improve performance of visibility rays. When reflection and refraction rays are added to a packet-based ray tracer, it is not clear how packets of secondary rays should be constructed, nor whether such packets will even provide performance benefits. While some interactive ray tracers support reflection [Parker et al. 1999; Wald 2004; Lauterbach et al. 2006; Reshetov 2006] and a few have added refraction [Parker et al. 1999; Wald et al. 2002; Bigler et al. 2006], there is little detail in the literature about the impact of adding such secondary rays on rendering performance. Most of these systems also abandon the use of packets for such secondary rays, presumably because secondary rays lack a shared origin. There are three exceptions to this in the literature. The special purpose hardware from the University of Saarland [Schmittler et al. 2002; Woop et al. 2005; Woop et al. 2006] uses packets of four rays for both primary and secondary rays, but statistics are not provided for the the performance of secondary rays. Bigler et al. [2006] use packets for all secondary rays, but they provide no results on the performance of such packets. Reshetov [2006] uses packets for reflection rays, and concludes that when using *k*d trees, packets may not help performance.

Distribution Ray Tracing While DRT is currently a batch algorithm, reducing the number of samples per pixel has been a focus since its invention [Cook et al. 1984]. Low sampling rates can be achieved using interleaved sampling, which tries to replace low-frequency artifacts with dithering-like structured error in screen space [Keller and Heidrich 2001; Kollig and Keller 2002]. Although interactive DRT is a worthy goal, it is inherently slower than WRT for two reasons. First, to render non-singular effects, DRT requires more samples per pixel than WRT. Second, the rays generated in a DRT renderer are less coherent (i.e., they have a greater range in both ray origin and direction) than WRT rays, again due to non-singular effects such as depth of field and glossy reflection. This reduced ray coherence could imply DRT rays are intrinsically more expensive than coherent WRT rays.

Summary Most research indicates that packets are very useful for interactive ray casting, but the little quantitative evidence that has been gathered for reflection and refraction rays makes it unclear that secondary ray packets are helpful. It is not known how much overhead is required when ray casting is replaced with WRT, nor whether packets can be useful for reflection and refraction rays. Finally, even if packets can be useful for WRT, it is not known whether the same will be true for the less coherent rays in DRT. These unknowns are the main topic of this paper.

3 Interactive Whitted-style Ray Tracing

In this section we describe our new algorithms for interactive packet-based WRT. First, we explain how ray packets in WRT are different from packets in ray casting. Next, we outline an algorithm for efficient packet-based traversal of a bounding volume hierarchy. Finally, we compare several different methods for assembling packets of secondary rays, including a method which yields interactive performance in our tests.

3.1 General Packets

To implement WRT with a packet-based system, the code must be updated to handle general packets of rays. By "general packets," we mean packets in which the ray origins and directions are not constrained in any way (Figure 2). In ray casting, the pinhole camera model allows us to assume that all rays within a packet share a common origin; in a general packet, each ray may have a unique origin. In WRT, general packets may be created by reflection and refraction. For example, when a packet of primary rays hits an object, each reflected ray will have a different origin, due to the fact that the incident rays hit the object at different locations.

3.2 Bounding Volume Hierarchy

We use a bounding volume hierarchy [Rubin and Whitted 1980] for the spatial acceleration structure in our renderer. We have chosen a bounding volume hierarchy (BVH) instead of a *k*d tree or grid for two main reasons. First, the BVH performs well for the types of deformable models used in many interactive programs such as games [Wald et al. 2006a]. Second, it is relatively easy to write packet-based BVH traversal code that is efficient for general packets. While new research may change our decision in the future, we think the BVH is currently the best choice for our needs.

Just as with the grid or *k*d tree, a culling test can indicate when an entire ray packet misses a bounding box during BVH traversal. We adopt the interval arithmetic-based culling approach presented by Wald et al. [2006a]. That method does not handle general packets, but it is straightforward to modify it to do so, as follows. The classic ray-box test computes three intersection intervals in ray parameter space; the ray hits the box if and only if the set intersection of the three intervals is non-empty. For general packets, each packet stores six intervals that encompass all ray origin and direction values over each Cartesian coordinate. We can then perform the classic ray-box test, using intervals in place of single coordinates. This yields three

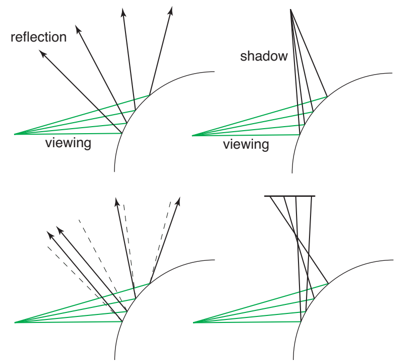

Figure 2: The top row shows WRT viewing rays in green and packets for reflection (left) and shadows (right). The bottom row shows similar configurations in DRT. In both WRT and DRT, reflection rays form "general" packets that lack a common origin. If depth of field were added, the DRT viewing rays would also lack a common origin, since each ray would be emitted from a different location on the camera lens. Finally, note that the random perturbations in DRT can create packets with less coherence than in WRT, with respect to ray origins and directions.

conservative intersection intervals that encompass the intersection intervals between the box and all individual rays in the packet. For some packets, the intersection of these three conservative intervals will still be empty, and the traversal of the entire packet can be terminated. Further details of this method are available in a technical report by Boulos et al. [2006b].

3.3 Assembling packets for secondary rays

In a packet of primary rays, some or all of the viewing rays will hit surfaces that may or may not share object ID, material properties, geometric proximity, or surface orientation. Any of these properties may be used in deciding how to create packets of secondary rays. For example, in a checkerboard with alternating brushed metal and glass squares, it is not clear whether a single packet of secondary rays should contain rays reflected from both metal and glass. Figure 3 illustrates the complexity of assembling secondary rays into packets. For the 16 primary rays shown, 6 shadow rays are generated, 12 specular reflection rays are generated, and 6 specular refraction rays are generated. Among the many options for generating secondary ray packets, we think the following methods are potentially useful approaches:

No packets: Each of the secondary rays are sent separately.

Runs: Secondary rays are traced in the same packet if they have some property in common (e.g., intersected material type), and their corresponding primary rays are numerically adjacent to each other.

Groups: All secondary rays with some common property (e.g., intersected material type) are grouped in a packet.

Ray Types: Three packets are generated: one containing all shadow rays, one containing all reflection rays, and the third containing all refraction rays.

Blind: One packet is generated, containing all secondary rays.

The two most complete interactive WRT systems described in the literature do not use packets for secondary rays [Parker et al. 1999; Wald 2004]. Bigler et al. [2006] use the runs method, but they

Figure 3: Sixteen packeted rays hit various objects and materials. Rays 1, 2, 3, and 4 hit diffuse surfaces and only generate shadow rays. Rays 5 and 6 hit the floor and generate both specular reflection and shadow rays. Rays 7, 8, 11, and 12 hit the metal teapot and generate only specular reflection rays. All other rays hit glass objects and generate both specular reflection and refraction rays.

do not provide examples with reflection or refraction. The University of Saarland hardware prototypes use the ray types method with packets of four rays each [Schmittler et al. 2002; Woop et al. 2005; Woop et al. 2006]. None of these systems have reported detailed performance statistics for either reflection or refraction rays in isolation. Reshetov [2006] groups all reflection rays into a packet and gives detailed statistics for these rays, but his results are not encouraging.

For the example in Figure 3, the following secondary packets are sent for the runs, groups and ray types method. We assume that intersected material type is the common property used to group rays in the runs and groups methods.

Runs: Two packets of shadow rays are generated, containing rays $(1,2,3,4)$ and $(5,6)$, respectively. Five packets of reflection rays are traced: $(5,6)$, $(7,8)$, $(9,10)$, $(11,12)$, and $(13,14,15,16)$. Finally, two refraction packets are generated: $(9,10)$ and $(13,14,15,16)$.

Groups: The packets are similar to the Runs method, except rays in the same packet need not be adjacent. Once again, two shadow ray packets are traced: $(1,2,3,4)$ and $(5,6)$. However, only three reflection packets are generated: $(5,6)$, $(7,8,11,12)$, and $(9,10,13,14,15,16)$, along with a single refraction packet: $(9,10,13,14,15,16)$.

Ray Types: Three packets are traced, one containing all shadow rays $(1,2,\ldots,6)$, one containing all reflection rays: $(5,6,\ldots,16)$, and one containing all refraction rays: $(9,10,13,14,15,16)$.

The runs method requires little state, and packets can be scheduled as soon as the run is interrupted. The ray types method also requires minimal state, since no more than three new packets are generated. We believe that both of these methods are useful; we have tested both and found that generating packets with the ray types method yields the best performance.

The grouping method is problematic for several reasons. The most obvious reason is that in the general case, the number of groups is bounded only by the number of outgoing rays. For example, if we group outgoing rays into packets such that rays within a small angle θ of each other are assembled into a single packet, it is possible to choose θ small enough so that all outgoing rays end up in different groups. Thus general grouping requires an impractical amount of

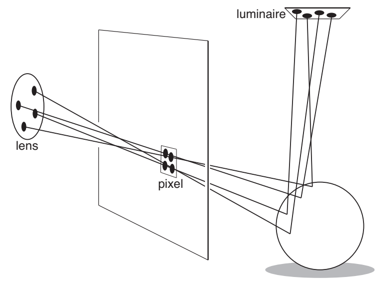

Figure 4: Distribution Ray Tracing

state on the stack, since each ray packet is statically allocated with space for 64 rays.

However, it is possible to bound the total number of groups (and hence outgoing packets) allowed. For example, directional binning could generate eight packets corresponding to the 8 possible direction octants. In fact, the ray type method described above is a particular form of grouping that generates at most three outgoing packets: one for shadow rays, one for reflection and one for refraction. We prefer the ray type grouping method, as it has low overhead and is less sensitive to ray ordering than the runs method. We also tested the octant-based directional grouping method, but it suffers from large space requirements; up to 16 groups are required, 8 for shadow rays and 8 for rays requiring recursive shading.

The blind method is obviously a poor choice, since grouping secondary rays of all types will yield packets with poor coherence. This case is made worse by transparent objects, since reflection and refraction rays are placed in the same packet, although the two types of rays will tend to go in very different directions. Furthermore, it is often more efficient to trace shadow rays separately from recursive shading rays; shadow rays only require occlusion tests, which may be more efficient than the full intersection tests needed for other ray types. Unfortunately, the blind method does not allow even this simple separation.

Due to the disadvantages of general grouping and blind assembly, we conclude that only ray types and runs are viable options. A thorough comparison can be found in the results section.

4 Distribution Ray Tracing

Distribution ray tracing differs from single-sample WRT in several important ways. The major difference between WRT and DRT is the ability of DRT to handle non-specular effects such as depth of field, motion blur, soft shadows, and glossy reflections (see Figure 4). In DRT, multiple primary rays are traced through each pixel, and each primary ray originates from a different position on the camera lens, at a different time. Rays that intersect a surface send shadow rays to different positions on area light sources. Rays that hit glossy surfaces send reflection rays that are perturbed from the ideal reflection direction.

The main concern for interactive DRT is that ray packets will not exhibit enough coherence to make ray packets worthwhile. In the Results section of this paper, we quantitatively examine the cost of ray packets in DRT. In the remainder of this section, we describe the main differences between our WRT and DRT implementations. More details may be found in [Boulos et al. 2006a].

4.1 Ray Branching

In single-sample WRT implementations, rays that hit a dielectric surface must branch into a reflection and refraction ray, since both of those components must be computed. One benefit of the multisampling in DRT is that ray branching is not as vital as it is in single-sample renderers. For example, if a packet of 64 rays hits a surface that it 25% reflective and 75% refractive, instead of tracing N reflection and N refraction rays, we would trace only N rays total, 25% of which are reflection rays, and 75% of which are refraction rays. This cuts down on the branching factor in the ray tree and uses multisampling to average the combined effects of reflection and refraction. Since ray types are chosen probabilistically, we avoid scintillation artifacts by choosing samples consistently over time (see below).

4.2 Motion-blurred Primitive Intersection

The only aspect of primitive intersection not already handled by our WRT renderer is motion blur. Each frame in a Whitted-style ray tracer occurs at one exact point in time, and all triangles have a fixed, well-defined position, even in an animated scene. In distribution ray tracing, however, each frame corresponds to a continuous interval of time, so moving primitives actually have an entire range of possible locations. As each ray has a fixed time stamp, different rays may potentially see the same triangle at different positions.

Therefore, we cannot use the fast projective triangle test proposed by Wald et al. [2001], since this depends on precomputing data for static triangles. Instead, we revert to a barycentric triangle test similar to one optimized for general packets of rays by Kensler and Shirley [2006].

4.2.1 Impact of Motion Blur on the BVH

To implement motion blur, we must also enlarge the BVH's bounding volumes to encompass all of a triangle's potential positions in the rendered frame. Thus, the bounding volumes in a DRT BVH are looser than those in WRT. The performance impact of looser bounding volumes varies with several parameters. If triangles move faster, their bounding volumes grow, and rendering cost increases. Similarly, finely tessellated scenes cause a larger *relative* increase in the bounding volumes' surface area, even if the *absolute* amount of movement is the same. On the other hand, higher frame rates lead to less motion per frame, suggesting that at 50-100Hz (as in games) the overhead due to motion blur would be lower than with our current slower frame rates. Similarly, if future ray tracers will use higher-order surfaces (e.g., subdivision surfaces) instead of triangles, the relative motion blur overhead would decrease, as these primitives are larger than individual triangles.

Note that motion blur due to a moving *camera* also has an adverse effect due to reduced coherence in primary packets (similar to depth of field), but this does not affect the tightness of the BVH.

4.2.2 Amortizing Normal Vector Calculations

In DRT, each primary ray in a pixel is assigned a unique time value. Because each triangle may be rotating over time, the triangle's normal may not be identical for each ray in a packet. Here we give the details of our method that allows us to cheaply compute the per-ray normal with constant setup per packet and quadratic interpolation per ray.

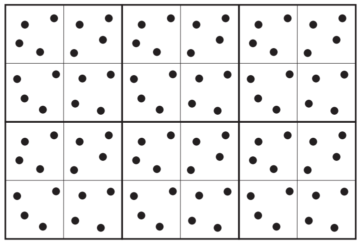

Figure 5: We use interleaved sampling in our renderer. A static set of samples is generated on a multi-pixel tile, and this pattern is repeated across the image.

If we denote the vertices of a static triangle as $\vec{p}_i$, then the vertices of a moving triangle (linearly interpolated between frames) can be written as $\vec{p}_i(t) = (1-t)\vec{p}_{i_s} + t\vec{p}_{i_e}$, where $\vec{p}_{i_s}$ and $\vec{p}_{i_e}$ denote vertex positions at the start and end of the frame, respectively. Here we assume that the parameter t varies from 0 at the start of the frame to 1 at the end. The edges of the triangle can be written similarly as: $\vec{e}_i(t) = (1-t)\vec{e}_{i_s} + t\vec{e}_{i_e}$.

Computing the normal of a static triangle is straightforward: $\vec{N} = \vec{e}_0 \times \vec{e}_1$. In the case of a moving triangle, the expression is the same and produces the following quadratic form:

$$\begin{aligned} \vec{N}(t) &= \vec{e}_0(t) \times \vec{e}_1(t) \\ &= [(1-t)\vec{e}_{0_s} + t\vec{e}_{0_e}] \times [(1-t)\vec{e}_{1_s} + t\vec{e}_{1_e}] \\ &= (1-t)^2\vec{N}_s + t(1-t)\,[\vec{e}_{0_s} \times \vec{e}_{1_e} + \vec{e}_{0_e} \times \vec{e}_{1_s}] + t^2\vec{N}_e \end{aligned}$$

In this form, we can determine the normal for all rays in the packet by first computing four cross products and then quadratically interpolating these values for each ray. For large ray packets (we use 64 rays per packet) this reduces the number of cross products required by factor of 16. It would also be possible to compute these vectors once per frame, but we have not explored this option.

We have implemented this method, and for many architectures, performance actually degrades relative to naive computation of the normal vector on-the-fly for every ray. For example, on an x86 processor, our renderer ran 10% slower using the amortized method. We believe this is due to the increased register pressure required to keep the precomputed vectors resident. Our method may be useful on architectures such as the Cell that have a larger register file, thus we have included it here.

4.3 Generating Random Samples

To make DRT interactive, it is imperative to use few samples per pixel. For the amount of computational power that will be available in the foreseeable future, this implies that the number of samples will be smaller than that needed for convergence, and visible error will be present. An intelligent sampling method such as Keller and Heidrich's [2001] interleaved sampling (see Figure 5) can be used to decrease the perceptible error for a given sampling rate. Although this makes the sampling code somewhat more complicated than a simple jittering-based DRT implementation, it does not negatively impact performance. Another benefit of using a static sample set such as the one presented in Keller and Heidrich is that it removes the temporal scintillation artifacts that arise when using different random samples in every frame.

We use a sampling scheme based on the Sudoku game. Details may be found in [Boulos et al. 2006a]. Although different sampling methods lead to varying amounts of visible noise (for a given number of samples per pixel), rendering performance is not highly dependent on the sample set used. The one exception is that the mapping from random samples to scene space must be done intelligently. For example, when sampling a Phong lobe for glossy reflection, certain mappings transform the origin in sample space to a reflected ray that is perfectly tangent to the object's surface. These tangent rays generate many false positives when performing the BVH traversal, and adversely affect performance.

	RCS fps	WRT fps	DRT fps
conference	8.0	2.8	0.02
rtrt	16.5	9.3	0.13
poolhall	12.0	6.1	0.07

Table 1: Performance in frames per second (fps) on a shared memory machine with four dual core 2GHz Opteron 870s for ray casting with shadows (RCS), Whitted ray tracing (WRT), and distribution ray tracing (DRT). RCS and WRT have one sample per pixel, and DRT has 64 samples per pixel. Image sizes are 1024 by 1024 pixels.

5 Results

Our system is implemented in C++ with SIMD extensions, and is modified from the system used for ray casting with shadows (RCS) by Wald et al. [2006a]. The most important extensions to that code was adding support for general packets, reflection, refraction, and interleaved sampling. To establish a baseline, we first use ray casting with shadows. As noted by Reshetov et al. [2005], just adding support for normalized viewing rays, local shading, and display significantly slows down ray casting. We have noted the same phenomenon in our code, and it is similar to the factor of two noted by Reshetov et al. Our baseline includes normalized viewing rays, local shading with shadows, and display. For all tests we start with viewing ray packets of 64 rays, and use *ray types* to build secondary packets. We ran all of our tests with different packet sizes (2×2, 4×4, 8×8 and 16×16) and found as in [Wald et al. 2006a] that the 8×8 packets perform best.

We ran our system on three scenes (see Figure 6) using camera paths for each scene (the path for the conference scene was originally used by Reshetov[Reshetov 2006]). The default maximum depth allowed is 50, but ray tree attenuation usually keeps the trees much shallower. The first columns of Tables 1 and 2 show our system performance for a one megapixel image on an eight core system for RCS. When we add reflection and refraction (WRT), as shown in the second column of Table 1, there is a slowdown of 2-3 times in framerate from RCS. The majority of this slowdown is because more rays are sent, however, reflection and refraction rays are more expensive than primary and shadow rays (see Figure 7). This difference in cost is also represented in the difference in total rays traced per second (see Table 2). It is important to note that WRT is only about 30% slower per ray than RCS.

With DRT at 64 samples per pixel, we are factors of hundreds or thousands from interactive performance. However, the rays per second achieved is about one half that for WRT on the conference scene, and only about 30% worse for the other two models. This is partly an artifact of the conference scene material parameters: all surfaces are reflective with a fairly reasonable exponent (mimicking the test from Reshetov[Reshetov 2006]). One interesting question is how much it slows down a WRT just to add the sampling infrastructure for DRT even if it is not used (i.e., the reflection rays are perturbed by zero degrees and remain ideal in behavior). Our tests have shown a consistent slowdown of around 12%, so this is not the main source of the slowdown of DRT.

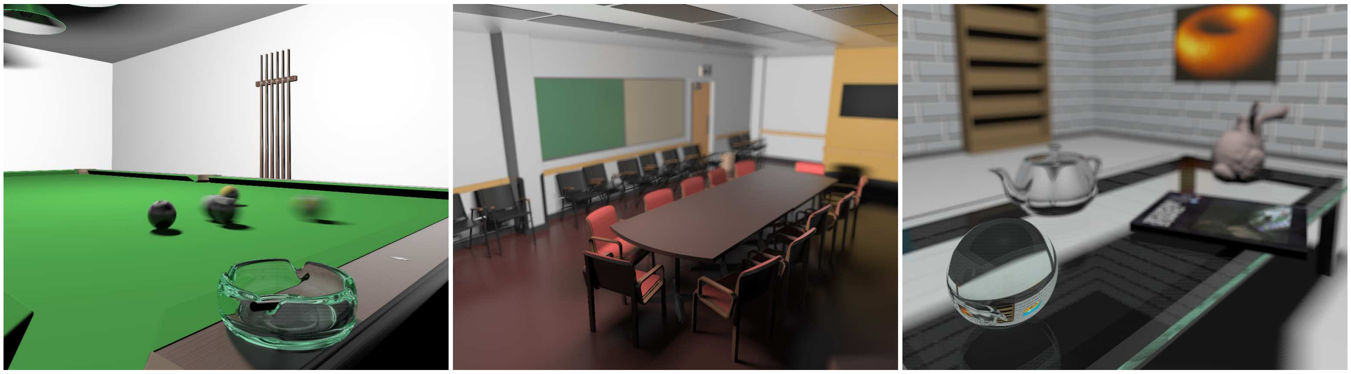

Figure 6: Our three test scenes. Left: pool hall (305,314 triangles). Middle: conference (282,664 triangles). Right: rtrt (83,844 triangles).

	RCS rps	WRT rps	DRT rps
conference	16.2M	12.3M	4.5M
rtrt	33.1M	28.4M	20.2M
poolhall	24.8M	17.2M	10.8M

Table 2: The number of rays traced per second (rps) for the same configurations as in Table 1.

5.1 From WRT to DRT

In our tests, most DRT features display fairly small percentage differences across different values and the different scenes. For example, in changing the diameter of the lens aperture from 0 (effectively a pinhole camera) to twice a reasonable size we only see percentage differences at each step (see Figure 7). Light source size only affects the shadow rays as these rays do not cast recursive rays. The exposure time has behaves similar to other variables for small values, but is surprisingly non-linear. When the exposure time is low, primitives in the scene expand the bounding boxes less and to the packet of rays "look like less primitives". As this exposure time gets higher, this effectively creates many more primitives for the rays to intersect.

Our experiments reveal that in terms of overall performance, both the ray type and shader id runs assembly algorithms produce substantial performance increases over the no packets assembly algorithm (around 2-3x speedup for our scenes). Compared to each other, the ray type assembly is usually 10-20% faster for a given scene over the full animation path. While this seems like a small improvement, it is important to understand where this improvement comes from.

The difference in performance between ray type and runs assembly can be seen from looking at the behavior of packets of rays as the bounce depth increases (see Figure 8). Both methods perform fairly similarly at first and the difference in overall performance is only around 10%. At higher bounce depths, however, the runs assembly demonstrates usually demonstrates between 15-25% more primitive and box tests. This doesn't show up as a big penalty overall, but this is an artifact of ray tree pruning (there are many fewer of these higher cost rays). As the performance gap between CPUs and memory increases, reducing box tests will reduce memory accesses and should widen the gap between ray type assembly and runs assembly [Reshetov 2006]. Similarly, if the number of rays traced at deeper bounces becomes more important (as for path tracing or caustics from long specular chains) the ray type assembly method will pull further ahead.

6 Conclusions and Discussion

We have demonstrated what we believe is the first interactive WRT system to support animated and dynamic scenes such as those used in games. We have shown that ray packets and reflection/refraction rays are not necessarily incompatible. We have also shown that DRT is not severely more expensive *per ray* than WRT, and that most of the cost difference is due to necessary multisampling. The following are a number of important questions we have not definitively answered, along with our best current answers. We believe all of these topics deserve further study.

What applications benefit from raytracing? The sub-linear time complexity of ray-triangle intersections is a primary advantage of raytracing, which allows us to interactively render densely tessellated surfaces with complex lighting and materials. The process of character posing often involves manipulating a bone rig and previewing the influence on a low-resolution mesh in a limited lighting environment. The reason for this is due in large part to polygon rasterization limits and the demands of complex lighting. Interactive DRT offers the potential for animation setups in a lighting environment that is closer to the final frame quality. DRT also holds promise for interactive games with substantially more detailed models and more general lighting. This approach avoids special case approximations such as environment maps and low quality shadow maps by directly and efficiently simulating reflection and visibility. The character model seen in Figure 1 can be posed in our system at interactive rates (at lower sample rates, 1-16 samples per pixel), allowing an animator to work in a more representative lighting environment.

Is DRT enough? Several extensions to WRT and DRT allow computation of global illumination effects; among these are path tracing [Kajiya 1986], bidirectional path tracing [Lafortune and Willems 1993], and photon mapping [Jensen 1996]. These are certainly useful for some applications, and if enough computational power becomes available, they are worth pursuing. However, we think DRT will be sufficient for many applications including most games, and simpler additions such as ambient occlusion will be almost as valuable as global illumination.

How many samples per pixel are needed? WRT benefits from multiple samples per pixel for antialiasing, and DRT requires multiple samples per pixel for acceptable image quality. The number of samples needed will depend on scene and display characteristics; in our experience, 16 to 64 samples per pixel are sufficient for high quality results.

Should rasterization be used for visibility? This is the trend in the computer-generated film industry because of the high number of procedural objects (e.g., displacement-mapped subdivision surfaces) that are used [Christensen et al. 2006]. However, we believe interactive applications will benefit more from enhanced lighting effects than from complex procedural geometry that requires on-the-fly computation. Since visibility computations are an inherent part of ray tracing, we believe that future interactive applications may simply use ray tracing alone, rather than computing visibility with rasterization.

Shouldn't GPUs be used for ray tracing? So far, GPU ray tracers are not as fast as CPU ray tracers. If reflection and refraction from curved surfaces are not needed then the accumulation buffer technique [Haeberli and Akeley 1990] could be used to great effect. However, we think that such reflections and refractions are desirable.

What hardware will WRT and DRT run on? WRT performs reasonably well on commodity CPUs, and with teraflop processors, WRT should run very fluidly on most scenes. DRT, on the other hand, may require special purpose hardware, especially if high-resolution images are needed.

Are ray packets a good idea? Ray packets trade software complexity for speed, and for secondary rays, the trade-off is not as clear as it is for ray casting. Although it is not often discussed in the graphics literature, most researchers are well aware that there is a high hidden cost for software complexity. More research into automatic ray scheduling in the spirit of Pharr et al. [1997] could combine the best qualities of packet-based and single-ray code.

What is the new bottleneck? Shading time is now competing with total tracing time as the bottle neck (a profile of our code reveals that they are nearly equal), as long as some reasonable packet grouping is used. This implies that one of the most important tasks for future work is to be able to group shading operations and perform common sub-expressions in a parallel manner. Alternatively, robust methods that amortize shading costs (similar to irradiance caching) may provide the same sort of improvement for shading cost that we have seen in tracing costs.

References

BENTHIN, C., WALD, I., SCHERBAUM, M., AND FRIEDRICH, H. 2006. Ray Tracing on the CELL Processor. In Wald and Parker [Wald and Parker 2006], 15–23.

BIGLER, J., STEPHENS, A., AND PARKER, S. G. 2006. Design for parallel interactive ray tracing systems. In Wald and Parker [Wald and Parker 2006], 187–195.

BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2006. Interactive distribution ray tracing. Tech. Rep. UUSCI-2006-022, SCI Institute, University of Utah, Salt Lake City, Utah, USA.

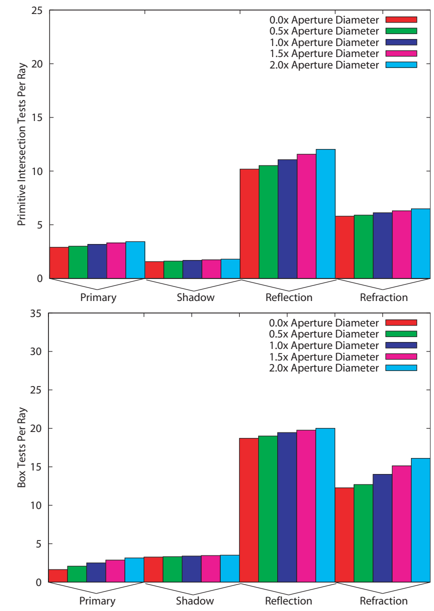

Figure 7: Effect of aperture on primitive tests (top) and box tests (bottom) per ray. As aperture diameter increases the effect on primitive intersections is fairly small.

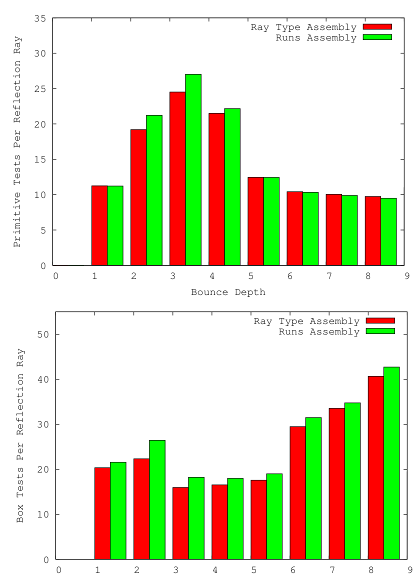

Figure 8: Comparison of ray type assembly and runs assembly for primitive tests (top) and box tests (bottom) with increasing bounce depth.

BOULOS, S., WALD, I., AND SHIRLEY, P. 2006. Geometric and Arithmetic Culling Methods for Entire Ray Packets. Tech. Rep. UUCS-06-010.

CHRISTENSEN, P. H., FONG, J., LAUR, D. M., AND BATALI, D. 2006. Ray tracing for the movie 'Cars'. In Wald and Parker [Wald and Parker 2006], 1–6.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH '84)*, vol. 18, 137–145.

DEMARLE, D. E., PARKER, S., HARTNER, M., GRIBBLE, C., AND HANSEN, C. 2003. Distributed interactive ray tracing for large volume visualization. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 87–94.

FENDER, J., AND ROSE, J. 2003. Estimating performance of a ray-tracing ASIC design. In *Proceedings of IEEE International Conference on FPT*, 7–14.

FOLEY, T., AND SUGERMAN, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of HWWS*, 15–22.

GÜNTHER, J., FRIEDRICH, H., WALD, I., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Ray Tracing Animated Scenes using Motion Decomposition. *Computer Graphics Forum*.

HAEBERLI, P., AND AKELEY, K. 1990. The accumulation buffer: hardware support for high-quality rendering. In *Proceedings of SIGGRAPH*, 309–318.

IZE, T., WALD, I., ROBERTSON, C., AND PARKER, S. G. 2006. An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. In Wald and Parker [Wald and Parker 2006], 47–55.

JENSEN, H. W. 1996. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, 21–30.

KAJIYA, J. T. 1986. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, ACM Press, 143–150.

KELLER, A., AND HEIDRICH, W. 2001. Interleaved sampling. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, 269–276.

KENSLER, A., AND SHIRLEY, P. 2006. Optimizing ray-triangle intersection via automated search. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*.

KOLLIG, T., AND KELLER, A. 2002. Efficient multidimensional sampling. *Computer Graphics Forum 21*, 3, 557–563.

LAFORTUNE, E. P., AND WILLEMS, Y. D. 1993. Bi-directional path tracing. In *Proceedings of Compugraphics '93*, 145–153.

LARSSON, T., AND AKENINE-MÖLLER, T. 2003. Strategies for Bounding Volume Hierarchy Updates for Ray Tracing of Deformable Models. Tech. Rep. MDH-MRTC-92/2003-1-SE, MRTC, February.

LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. 2006. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In Wald and Parker [Wald and Parker 2006], 39–45.

MUUSS, M. J. 1987. RT and REMRT - shared memory parallel and network distributed ray-tracing programs. In *Proceedings of the 4th Computer Graphics Workshop*, USENIX Association, Cambridge, MA, USA, 86–98.

PARKER, S., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive ray tracing. In *Symposium on Interactive 3D Graphics*, 119–126.

PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. 1997. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of SIGGRAPH*, 101–108.

PURCELL, T., BUCK, I., MARK, W., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *Proceedings of SIGGRAPH*, 703–712.

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering*, 299–306.

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

RESHETOV, A. 2006. Omnidirectional ray tracing traversal algorithm for kd-trees. In Wald and Parker [Wald and Parker 2006], 57–60.

RUBIN, S., AND WHITTED, T. 1980. A 3D representation for fast rendering of complex scenes. In *Proceedings of SIGGRAPH*, 110–116.

SCHMITTLER, J., WALD, I., AND SLUSALLEK, P. 2002. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, 27–36.

SIGGRAPH, 2002. Panel: Rays vs. rasters: When will ray-tracing replace rasterization? Kurt Akeley, Brad Grandtham, David Kirk, Tim Purcell, Larry Seiler, Philipp Slusallek (panelists).

WALD, I., AND PARKER, S. G., Eds. 2006. Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3, 153–164. (Proceedings of Eurographics).

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2002. OpenRT – a flexible and scalable rendering engine for interactive 3d graphics. Tech. Rep. TR-2002-01, Saarland University.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2006. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics (conditionally accepted)*. Available as SCI Institute, University of Utah Tech.Rep. UUSCI-2006-023.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics 25*, 3, 485–493. (Proceedings of ACM SIGGRAPH 2006).

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

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (June), 343–349.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: A programmable ray processing unit for realtime ray tracing. In *Proceedings of SIGGRAPH*, 434–444.

WOOP, S., BRUNVAND, E., AND SLUSALLEK, P. 2006. Estimating performance of a ray-tracing ASIC design. In Wald and Parker [Wald and Parker 2006], 7–14.