Large Ray Packets for Real-time Whitted Ray Tracing

Ryan Overbeck[1] Ravi Ramamoorthi[1] William R. Mark[2,3]

[1]Columbia University
[2]Intel Corporation
[3]University of Texas at Austin

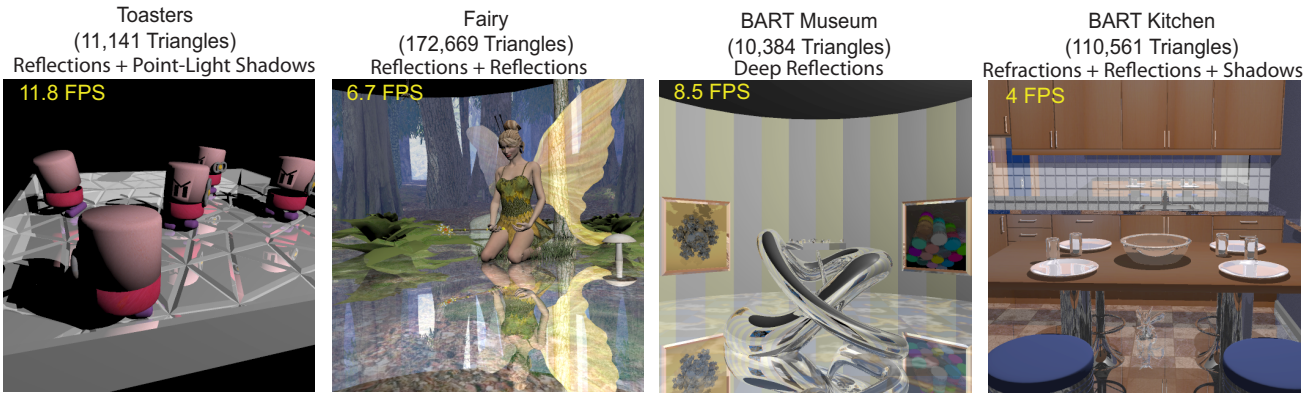

Figure 1: Real-time Whitted ray tracing on a single, affordable workstation is now possible. All images were rendered at 1024×1024 on a dual quad-core system (8 cores total) with 2.0 GHz Intel Xeon processors.

Abstract

In this paper, we explore large ray packet algorithms for acceleration structure traversal and frustum culling in the context of Whitted ray tracing, and examine how these methods respond to varying ray packet size, scene complexity, and ray recursion complexity. We offer a new algorithm for acceleration structure traversal which is robust to degrading coherence and a new method for generating frustum bounds around reflection and refraction ray packets. We compare, adjust, and finally compose the most effective algorithms into a real-time Whitted ray tracer. With the aid of multi-core CPU technology, our system renders complex scenes with reflections, refractions, and/or point-light shadows anywhere from 4–20 FPS.

1 Introduction

Real-time ray tracers use bundles of coherent rays, called ray packets, to achieve real-time performance. Recent approaches, such as [8], [10], and [12], allow algorithmic amortization across large packets of 16–256 rays by using new algorithms for scene traversal and bounding frusta to cull away expensive per-ray operations.

Ray packet tracing has proven very successful for primary visibility where neighboring rays are known to be regularly distributed and perspective parallel (ie. all rays meet at a point). Some success has also been demonstrated for point-light shadow rays, but there is very little published work on applying large ray packets to other ray traced effects.

We aim to employ large ray packet algorithms to achieve real-time Whitted ray tracing, but ray coherence is much less reliable in this domain. Beyond primary visibility, Whitted ray tracing requires secondary rays for point-light shadows, reflections, and refractions. According to [6], ray coherence degrades for these secondary effects, and we expect a corresponding drop in performance for large ray packet algorithms.

In this work, we study the two fundamental approaches for accelerating large ray packets: ray packet acceleration structure traversal algorithms and frustum culling. In Section 3, we will describe two old algorithms for traversing a Bounding Volume Hierarchy (BVH) as well as one new one. We call the two existing algorithms Masked traversal (introduced by [14] and used in [8]) and Ranged traversal (introduced by [10]). We introduce a new algorithm called Partition traversal which is robust to degradation in ray coherence. In Section 4, we review methods to generate frustum bounds around primary and shadow rays and introduce a new method to create frustum bounds around reflection and refraction rays. We believe this is the first published work to demonstrate frustum culling for reflections and refractions in a ray packet tracer.

In Section 5, we examine how the large ray packet algorithms for acceleration structure traversal and frustum culling respond to changing the three primary variables which affect ray casting performance: ray packet size, scene complexity, and ray recursion complexity. Larger ray packets provide more opportunity for algorithmic amortization, but can also lead to greater divergence during scene traversal. Scene complexity involves a combination of the gross number of scene triangles, relative triangle sizes and distribution, and surface variations. Ray recursion complexity is the degradation in ray coherence caused by secondary rays and increases from primary rays to shadow rays to refraction rays to reflection rays and also increases with the depth of reflection and refraction recursion. We summarize our conclusions in Table 1.

Based on the results of Section 5, Section 6 combines the best algorithms to create our fully real-time Whitted ray tracing system which consistently provides performance benefits of $3\times$–$6\times$ over 2×2 SIMD ray packets (see Table 2). With affordable multi-core CPU technology multiplying almost another order of magnitude, real-time Whitted ray tracing on commodity hardware using a single workstation is now fully realized. The images in Figure 1 were all generated with our system. From left to right, the first image has a reflective floor and point-light shadows and runs at 11.8 FPS. The fairy in the second image has refractive wings. Her eyes and the gold and jeweled components on her wand as well as the forest floor all reflect the scene. Even on this complex scene, we achieve

	Masked Traversal	Ranged Traversal	Partition Traversal	FrustumCulling
Introduced By	[14]	[10]	***New***	Primary+Shadows: [10] Reflections+Refractions: ***New***
Ray Packet Size	Only good for up to 4×4 packets.	Usually best for up to 8×8 packets and sometimes 16×16.	Good for all packet sizes.	Benefits tend to increase with packet size.
Scene Complexity	Only good for simple scenes.	Best for simple to moderate scenes.	Best for complex scenes.	Benefits decrease with scene complexity.
Ray Recursion Complexity	Bad for secondary effects.	Best for primary and shadow rays. Okay for low recursion. Bad for deep recursion.	Best for deep reflections and refractions.	Up to $2\times$ performance benefit for primary and shadow rays and $1.2\times$–$1.3\times$ for reflections and refractions.
Summary	Superseded by Ranged and Partition traversal.	Best for packet sizes $\leq 8 \times 8$, simple to moderate scenes, and moderate ray recursion complexity.	Best for large packet sizes $\geq 16 \times 16$, complex scenes, and high recursion complexity.	Best for primary rays and shadow rays. Helpful for reflection and refraction rays.

Table 1: Conclusions from the study of large ray packet algorithms in Section 5.

6.7 FPS. The frame from the BART museum in the third image is an example of deep reflections using 3 bounces of reflection at 8.5 FPS. Lastly, the frame from the kitchen scene at the far right puts it all together, using 1-bounce reflections, 4-deep refractions, and point-light shadows at an interactive rate of 4 FPS.

2 Background

We introduce the algorithms we will be studying in Sections 3 and 4. First, we review previous work and provide some background for understanding large ray packet algorithms.

2.1 SIMD Ray Packets

SIMD ray packets were first used by [14] and allow 4 rays to traverse the scene as if they were one by taking advantage of the Single Instruction Multiple Data units available on modern CPUs. 4-wide SIMD ray packets consistently provide a $2\times$–$3\times$ performance improvement over single rays.

In our system, we use 2×2 SIMD ray packets as our smallest ray primitive and will refer to a 2×2 ray packet as an individual SIMD ray to emphasize this fact.

2.2 Large Ray Packets

The work of [14] also demonstrates that tracing multiple rays together offers benefits beyond the extra floating point computation performance afforded by SIMD. Indeed larger ray packets of $n \times n$ with $n = 8$ or $n = 16$ provide up to an order of magnitude improvement over SIMD rays when used for primary visibility. However, the acceleration structure traversal algorithms must change in order to allow for such large packets.

Large ray packets have been demonstrated on kd-trees [8], grids [12], and BVHs [10]. See the recent STAR report [13] for an overview of the build and traversal algorithms for these structures and others. Our work focuses on BVHs because they currently exhibit the best combination of build and ray casting performance. However, traversal and frustum culling algorithms are similar between structures, and we believe our results can benefit these other structures as well.

All of these works and their performance numbers target primary visibility, treating point-light shadows as an added bonus. None of them provide in-depth performance comparisons for ray traced reflections or refractions.

2.3 Frustum Bounds for Ray Packets

Tight bounding frusta around coherent ray packets can cull away many ray–Axis-Aligned Bounding Box (AABB) and ray–triangle intersection queries. Frustum culling has been demonstrated for primary rays in [8] and for both primary rays and point-light shadows in [10] and [12]. To our knowledge, ours is the first ray tracer to use frustum culling for reflection and refraction rays.

2.4 Whitted Ray Tracing using Ray Packets

We know of two other works which apply large ray packets to Whitted ray traced effects: [1] and [6]. [1] only study the singular combination of traversal and culling algorithms used in [10]. [6] restrict their study to 4×4 ray packets using only the traversal algorithm from [8]. We explore a broader range of traversal and culling algorithms and up to 32×32 packets.

Both works demonstrate a $2\times$–$3\times$ hardware performance benefit by using SIMD rays. [1] only achieve about a $1.5\times$ performance benefit by using large ray packets of 8×8–16×16 over SIMD rays in a Whitted ray tracer. Our analysis in Section 5 demonstrates that their traversal algorithm can be significantly slower for some scenes when using multiple bounce reflections and refractions. Moreover, we often see $3\times$–$6\times$ performance benefits when using our combination of large ray packet traversal and culling algorithms over SIMD rays.

[6] explores the possibility of regrouping rays based on various measures of coherence. Their results demonstrate that it is difficult to impossible to efficiently collect more ray coherence beyond what is provided by the image raster. As such, we always group ray packets into $n \times n$ groups as determined by the screen space coordinates of their ancestral camera rays.

3 Traversal Algorithms for Large Ray Packets

In this section, we introduce the large ray packet BVH traversal algorithms that we will compare in Section 5. We review two existing algorithms, Masked traversal in Subsection 3.1 and Ranged traversal in Subsection 3.2, and introduce our new algorithm, Partition traversal, in Subsection 3.3. The Appendix is provided as a supplement to this section and provides expanded pseudocode for the Ranged and Partition traversal algorithms.

In the descriptions that follow, we use $R = (r_0, r_1, r_2, ..., r_n)$ to denote the set of all SIMD rays in the large ray packet. Using an index i, we can retrieve the ith ray: $R[i] == r_i$. We also differentiate between the concepts of a ray being *active* and *alive*. At any step in the BVH traversal, a ray is *active* if the ray traversal algorithm assumes that the ray overlaps the cell's AABB. A ray is *alive* if it actually does overlap the AABB. All *active* rays are tested against the triangles at the BVH leaves whether or not they are *alive*.

3.1 Masked Traversal

We call the first large ray packet algorithm Masked traversal because it uses an array of boolean values to mask out dead rays at the BVH leaves. It is the first and simplest large ray packet algorithm used by [14] and [8] for traversing kd-trees.

The pseudo-code for Masked traversal in Figure 2 provides a basis for introducing all of the large ray packet algorithms in this Section. At each step, all rays are tested against the current cell. If *any* ray hits at line 10, then all rays continue through the tree together. At a leaf cell, lines 21– 27, we check if the bounding

```
 1: // Traverse a Ray packet, R, through theBVH
 2: void traverseBVH( Rays R, Frustum F, BVH theBVH )
 3:     BVHCell curCell = theBVH.root;
 4:     Stack<StackNode> traversalStack;
 5:     bool rayMasks[size( R )];
 6:     while ( true )
 7:       bool anyHit = false ;
 8:       if ( frustumIntersectsAABB( F, curCell.AABB() ))
 9:         for ( Index i=0; i < size( R ); ++i )
10:           rayMasks[i] = rayIntersectsAABB( R[i], curCell.AABB() ))
11:           if ( rayMasks[i] )
12:             anyHit = true ;
13:             if ( isInner( curCell )) break ;
14:       if ( anyHit )
15:         if ( isInner( curCell ))
16:           StackNode node;
17:           node.cell = curCell.farChild( R );
18:           traversalStack.pushBack( node );
19:           curCell = curCell.nearChild( R );
20:           continue ;
21:         else // isLeaf( curCell ) == true
22:           Triangles T = curCell.triangles();
23:           for ( Index j = 0; j < size( T ); ++j )
24:             if ( frustumIntersectsTriangle( F, T[j] ))
25:               for ( Index i = 0; i < size( R ); ++i )
26:                 if ( rayMasks[i] )
27:                   rayIntersectTriangle( R[i], T[j] );
28:       // END if ( anyHit )
29:       if ( traversalStack.empty() )
30:         break ;
31:       StackNode node = traversalStack.pop();
32:       curCell = node.cell;
33:     // END while ( true )...
34: // END void traverseBVH(...
```

Figure 2: Pseudo-code for Masked BVH traversal.

frustum culls the triangle at line 24, and if not, all alive (unmasked) rays are tested against the triangle. This algorithm can have many extra ray–AABB tests especially deep in the tree.

3.2 Ranged Traversal

Ranged traversal, introduced for use with BVHs in [10], attempts to avoid many of the ray–AABB tests in Masked traversal by tracking the first alive SIMD ray in R. Let i_a be the index to that SIMD ray. At a BVH cell, we find i_a using the getFirstHit() function:

```
Index getFirstHit( Rays R, Frustum F, AABB B, Index i_a )
    if ( rayIntersectsAABB( R[i_a], B )) return i_a;
    if ( !frustumIntersectsAABB( F, B )) return size( R );
    for ( Index i = i_a + 1; i < size( R ); ++i )
      if ( rayIntersectsAABB( R[i], B ))
        return i;
    return size( R );
```

A call to getFirstHit() replaces the AABB tests on lines 8– 13 in Figure 2.

We track i_a by adding it to the traversal stack's nodes:

```
struct StackNode
    BVHCell cell;
    Index i_a; // Index to the first alive ray
```

At the BVH leaves, we perform the reverse operation and find i_e, the index to the last alive ray in R:

```
Index getLastHit( Rays R, AABB B, Index i_a )
    for ( Index i_e = size( R )-1; i_e > i_a; --i_e )
      if ( rayIntersectsAABB( R[I[i_e]], B ))
        return i_e + 1;
    return i_a + 1;
```

We place a call to getLastHit() right after line 21 in Figure 2. All rays in the interval $[i_a,i_e)$ are active and are tested against the triangles in the leaf cell.

By tracking i_a and finding i_e at the leaves, we avoid many ray–AABB intersection tests at the inner cells, but add more ray–triangle intersections at the leaves. For coherent rays, this is acceptable and is a large improvement over Masked, but Ranged traversal can still end up with many extra active rays deep in the BVH leading to potential overhead.

3.3 Partition Traversal

Our new traversal algorithm, which we call Partition traversal, partitions the rays into strictly alive and dead subsets. Our approach is similar to the algorithm in [11], who aim to increase SIMD utilization for wide (> 4) SIMD units. Distinct from their method, our algorithm is real-time, using an efficient approach to filter out dead rays, and treats the SIMD ray as the smallest ray primitive.

We maintain a separate set of SIMD ray indices, $I = (i_0, i_1, i_2, ..., i_n)$ with $n = \text{size}(R)$. This list is initialized to $I = (0, 1, 2, ..., n)$ at the start of traversal, and instead of tracking the *first* active SIMD ray as we did in Ranged traversal, we use i_a to track one past the last active SIMD ray. We filter out SIMD rays which miss the current BVH cell's AABB with a call to partRays():

```
1: Index partRays( Rays R, Frustum F, AABB B, Indices I, Index i_a )
2:   if ( !frustumIntersectsAABB( F, B )) return size( R );
3:   Index i_e = 0 ;
4:   for ( Index i = 0; i < i_a; ++i )
5:     if ( rayIntersectsAABB( R[I[i]], B ))
6:       swap( I[i_e++], I[i] );
7:   return i_e;
```

A call to partRays() replaces the AABB tests on lines 8– 13 in Figure 2.

partRays() performs the frustum–AABB test first, then loops through the indices in I, testing each indexed SIMD ray against the cell's AABB. By swapping the elements in I and incrementing i_e at line 6, I is split in-place into two subsets with the indices to the alive SIMD rays in front of i_e. By the end of partRays(), i_e is one past the index to the last alive SIMD ray, and the rest of the SIMD rays indexed by $I[i_e$: size(I)-1] are inactive.

We store the result of partRays() in i_a, and, just as for Ranged traversal, we need to add only this one integer to the traversal stack's nodes. As the ray packet traverses down the tree, the list of alive rays gets smaller. As it pops back up the tree, the SIMD ray ids in I will be re-ordered, but i_a will still point to the end of the alive SIMD ray indices.

In order to test only the alive rays against the triangles at the BVH leaves, we replace the more expensive mask branches in the the loop at lines 25– 27 in Figure 2 with a simple indirection:

```
22:  for ( Index i = 0; i < i_a; ++i )
23:    rayIntersectTriangle( R[I[i]], T[j] );
```

Partition traversal is designed to gracefully handle degradation in ray coherence, and there is nothing limiting the ray packet's size beyond memory bandwidth. However, if the rays in R are truly coherent, then Ranged traversal may avoid more ray–AABB intersection tests.

4 Frustum Bounds for Large Ray Packets

In this section, we first review frustum culling basics, and then describe the construction of tight bounding frusta for primary rays in Subsection 4.1 and shadow rays in Subsection 4.2. We end with our new approach for bounding reflection and refraction rays in Subsection 4.3.

A bounding frustum culls AABBs and triangles using either its 4 bounding corner rays, its 4 side planes, or both. The corner rays

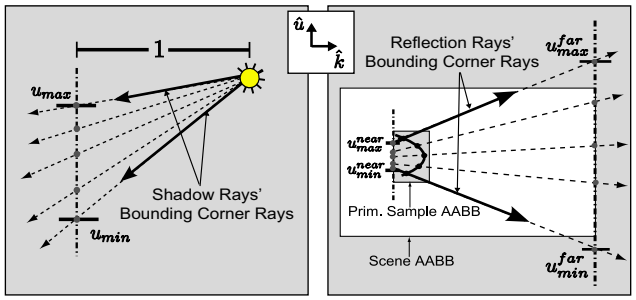

Figure 3: Finding bounding frustum corner rays for point-light shadow rays (left) and reflection rays (right).

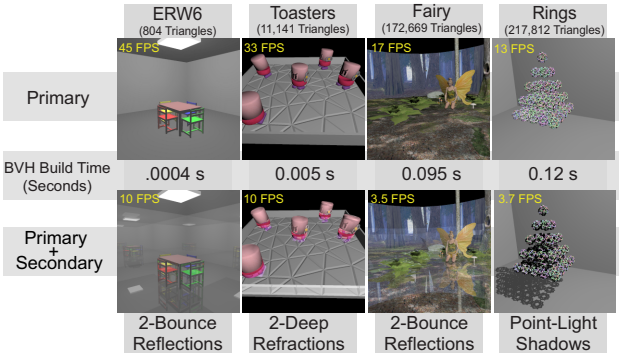

Figure 4: The scenes used for evaluating the traversal algorithms from Section 3 and the frustum culling algorithms from Section 4. All images were rendered at 1024×1024.

cull a triangle when all 4 rays lie outside of the same triangle edge, and they cull an AABB if all 4 rays are separated by the same slab (see [8] and [7]).

The side planes cull any convex polyhedron (either a triangle or AABB) when all of the polyhedron's vertices lie outside of the same side plane. Let $\vec{n}_i$ and b_i with $0 \leq i < 4$ be the plane normals and offsets for the 4 bounding frustum plane equations, and let p_k be the polyhedron's vertices, then:

$$H_i = \vec{n}_i \cdot \vec{p}_k - b_i. \tag{1}$$

If $H_i > 0$ then p_k is outside the plane defined by $(\vec{n}_i, b_i)$, and if all p_k are outside the same plane, then this plane culls the polyhedron. [8] shows a version of this test optimized for AABBs using SSE instructions.

Given 4 corner rays, we easily find the frustum's side planes. Let $\vec{o}_i$ and $\vec{d}_i$ with $0 \leq i < 4$ be the corner rays' origins and directions respectively, then:

$$\vec{n}_i = \vec{d}_i \times \vec{d}_{(i+1)\%4} \tag{2}$$

$$b_i = \vec{o}_i \cdot \vec{n}_i \tag{3}$$

4.1 Frustum Bounds for Primary Rays

The task of generating a bounding frustum starts with finding the frustum's 4 corner rays. For primary rays through a pinhole camera, the corner rays are simply the rays at the corners of the $n \times n$ ray packet, and the frustum planes are retrieved directly from Equations 2 and 3.

4.2 Frustum Bounds for Point-Light Shadow Rays

For shadow rays, the 4 corner rays defined by the raster are no longer guaranteed to bound the ray volume. Instead, we use an alternate approach as described by [2] and illustrate this method on the left of Figure 3.

We first choose a dominant axis for the ray directions which we call $\hat{k}$. We use the sum $\vec{d}^s = \sum_i^{n \times n} \vec{d}_i$ and take the axis of the max component: $\hat{k} = AxisOf(\max(d^s_x, d^s_y, d^s_z))$. Let the other two axes be $\hat{u}$ and $\hat{v}$. We place an imaginary plane orthogonal to $\hat{k}$ at a distance of 1 in front of the rays' origin. This plane is the vertical dashed line in Figure 3. We find the (u,v)–coordinates of the intersection between the ray and the plane which are simply $(u = d_u d_k^{-1}, v = d_v d_k^{-1})$ (we multiply by d_k^{-1} instead of dividing by d_k to emphasize the fact that d_k^{-1} is usually precomputed for each ray for efficient BVH traversal).

Let u_{min} and u_{max} be the minimum and maximum u–coordinates, and accordingly v_{min} and v_{max} the minimum and maximum v–coordinates. The (u,v)–coordinates of directions of the bounding corner rays will then be (u_{min}, v_{min}), (u_{max}, v_{min}), (u_{max}, v_{max}), and (u_{min}, v_{max}), with a 1 or -1 for the k–coordinate, and the origin for all 4 corner rays is simply the location of the point-light. Both the computation of the frustum planes in Equations 2 and 3 as well as the frustum–AABB and frustum–triangle culling tests based on Equation 1 simplify given that the k–coordinate will be 1 or -1, and the common (u,v) values between the neighboring corner ray directions. See [2] for details.

4.3 Frustum Bounds for Reflections and Refractions

The approach in Subsection 4.2 only works for rays that meet at a point and so doesn't apply to reflection or refraction rays. Here we introduce a new method which extends to general ray packets, and illustrate our algorithm on the right of Figure 3 which shows reflection rays bouncing off of a curved surface.

We start by choosing a dominant axis, $\hat{k}$, exactly as we did in Subsection 4.2, but this time, instead of a single imaginary plane, we pick two planes. In order to provide conservative bounds, these planes must bound the paths of all rays in the packet. Therefore, we choose a *far* plane at k^{far} from the scene's AABB in the $+\hat{k}$ direction and a *near* plane at k^{near} in the $-\hat{k}$ direction from the AABB bounding the ray origins. We then find the (u,v)–coordinates of the rays' intersections with both planes, resulting in the intervals $[u^{near}_{min}, u^{near}_{max}]$, $[v^{near}_{min}, v^{near}_{max}]$ at k^{near} and $[u^{far}_{min}, u^{far}_{max}]$, $[v^{far}_{min}, v^{far}_{max}]$ at k^{far}.

The corner ray origins are the extremal intersection points with the near plane: $(u^{near}_{min}, v^{near}_{min}, k^{near})$, $(u^{near}_{max}, v^{near}_{min}, k^{near})$, $(u^{near}_{max}, v^{near}_{max}, k^{near})$, and $(u^{near}_{min}, v^{near}_{max}, k^{near})$. The corner ray directions are the difference between the extremal intersection points with the far plane and these origins. As in Subsection 4.2, the frustum planes come from the corner rays using simplified versions of Equations 2 and 3, and the culling tests are based on simplified extensions of Equation 1. Our algorithm generates frusta with equivalent characteristics to the frusta used in [7] to cull triangles at acceleration structure leaf cells. See [7] for details on optimizing construction and intersection tests using this form of frustum bounds.

5 Results–Comparison

In this Section, we analyze the performance characteristics of the ray packet traversal algorithms from Section 3 and frustum culling algorithms using frusta generated by the methods from Section 4. We first give an overview of the comparison setup in Subsection 5.1, we then study traversal algorithms in Subsection 5.2 and frustum culling in Subsection 5.3. We examine how these algorithms respond to changes in scene complexity, ray recursion complexity, and ray packet size and summarize the results in Subsection 5.4. We use the best combination of algorithms to create our real-time Whitted ray tracer in Section 6.

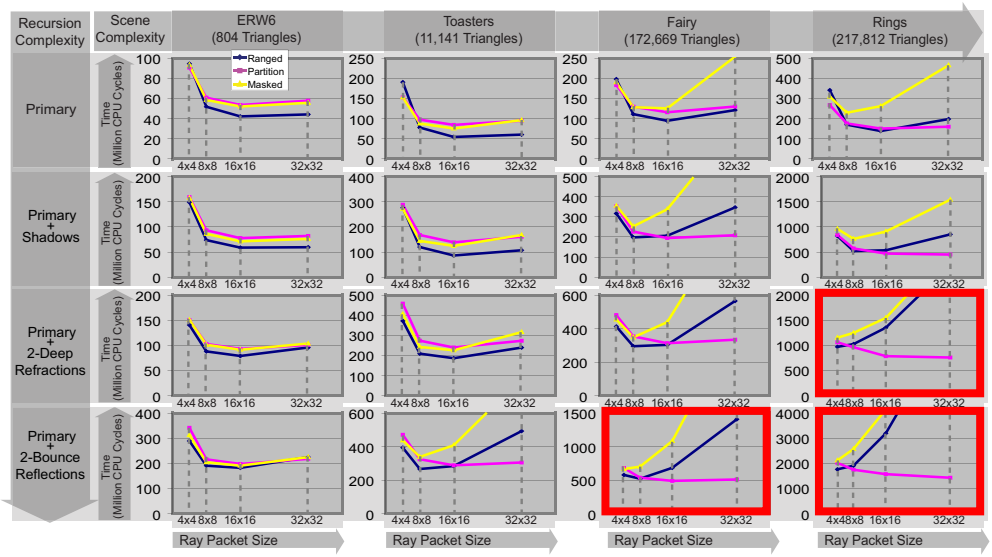

Figure 5: Plots of Masked, Ranged, and Partition traversal times for rendering one 1024×1024 image with varying ray packet size, scene complexity, and ray recursion complexity. Masked and Ranged traversal degrade relative to Partition traversal as any of scene complexity, ray recursion complexity, and/or ray packet size increase.

5.1 Comparison Setup

Hardware Configuration: All tests in this Section (except where otherwise noted) generate images at 1024×1024 resolution on a dual quad-core system (for a total of eight cores) with 2.0GHz Intel Xeon processors. While faster processors and more cores are available, our system is an example of an affordable hardware package. As of the time of writing this paper, such a system was commonly available for around $2000. The timings include all costs related to ray casting and shading. We leave out time to send the image to the graphics card as this adds anywhere from 10% more CPU cycles for the slower renders to 50% for the faster renders. For multi-threaded ray casting, we use the standard approach of tiling the image canvas and dealing out tiles to each thread.

Scenes: We use the scenes shown in Figure 4, each of which was chosen for specific qualities. The ERW6 scene is extremely simple, having only 804 triangles and all flat surfaces and presents an ideal environment for a coherent ray packet tracer. The Toasters scene, with 11,141 triangles, targets the lower end of the level of visible complexity in a typical video game scene. The Fairy scene has 172,669 triangles. It is a realistic, high complexity scene, perhaps a future video game scene with both large and tiny objects. The grass at the fairy's knees and the base of the tree is particularly difficult for a packet tracer. The Rings scene from the SPD [3] is specifically intended as a worst case scenario for reflection and refraction rays. The small and tangled rings serve to disperse secondary rays in all directions.

Acceleration Structure: As previously noted, we use a BVH as our acceleration structure. We use a single-threaded binned SAH build which has previously been shown in [9] to be interactive to real-time even for complex scenes, but is about an order of magnitude slower than state-of-the-art BVH builds using either a grid pre-build as in [9] or a pre-existing scene hierarchy as in [4] or [15]. While we focus on ray casting performance, we also include the time to build our BVH from scratch in Figure 4 to demonstrate that our acceleration structures are of interactive quality.

Whitted Effects: In order to best evaluate performance for reflection and refraction rays, we set *all* scene materials to be reflective or refractive. This makes for some very difficult situations for our ray packet tracer. The detailed geometry in the Fairy and Rings scenes create some highly incoherent ray packets. We investigate some more reasonable rendering configurations in Section 6.

5.2 Masked vs. Ranged vs. Partition Traversal

We compare Masked, Ranged, and Partition traversal in the collection of plots in Figure 5. Each plot shows time to render one image in millions of CPU cycles (lower values on the y-axis mean faster render times) versus ray packet size. Along the x-axis, we use 4×4, 8×8, 16×16, and 32×32 ray packets. The plots themselves are organized in a table with scene complexity increasing along the x-axis, and ray recursion complexity along the y-axis. From top to bottom, we display ray recursion complexity using primary visibility, primary visibility with point-light shadows, 2-deep refractions, and 2-bounce reflections. We use frustum culling for all results in this Subsection.

Masked traversal, in yellow (or light gray for gray-scale prints), is consistently slower than Ranged traversal, and is slower than Partition in most cases. It was found to work consistently only up to 4×4 ray packets in [14] and [8], and our results agree with these earlier findings. Render times explode with increased packet size for higher scene complexities to the right and deeper levels of ray recursion to the bottom.

As noted in Section 3.1, Masked traversal keeps all rays active at the inner cells which can lead to large overheads. If even one packet ray decides to visit a BVH leaf, then all packet rays will be tested against the leaf's AABB. Less coherent packets will have many extra ray–AABB intersection tests leading to poorer performance with increased scene complexity, ray recursion complexity, and packet size.

Ranged traversal, in dark blue (dark gray), behaves significantly better than Masked traversal, and provides the best results on up to 16×16 ray packets for most cases. The most notable exception to this are the three high-lighted plots in the lower right corner with higher scene complexity and higher ray recursion complexity. While Ranged traversal performs much better for the other configurations, a downward trend is clearly visible as we increase scene complexity, ray recursion complexity, and ray packet size.

Partition traversal, in magenta (medium gray), is the most robust to the degrading coherence for higher scene complexity, reflection and refraction rays, and larger ray packets. Both Masked

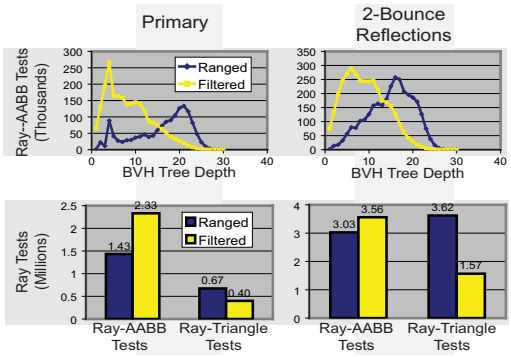

Figure 6: Histograms (top) and bar charts (bottom) counting the number of ray–AABB tests and ray–triangle tests required for rendering the Fairy scene at 512×512 using 16×16 ray packets. Ranged traversal is better for primary rays, but Partition traversal is better for reflection rays.

and Ranged traversal reach a breaking point as ray packet size increases. Partition traversal, on the other hand, consistently improves with increased packet size regardless of scene and ray recursion complexity, and is limited only by memory bandwidth and cache coherence which degrades slightly for 32×32 packets.

5.2.1 Partition vs. Ranged traversal: A Closer Look

Ranged traversal and Partition traversal each have their own strengths and weaknesses, and Figure 6 shows why. These graphs compare the number of ray–AABB tests and ray–triangle tests because these dominate the ray casting time in our system.

Partition traversal, in yellow (light gray), generates a peak in both histograms higher in the tree, closer to the root, while Ranged traversal, in blue (dark gray), is more peaked at the deeper cells. As described in Subsection 3.3, Partition traversal always tests every alive ray against every BVH cell which leads to more ray–AABB tests higher in the tree. Deeper in the tree, most of the rays have been filtered out, so there are fewer ray–AABB tests.

Ranged Traversal, as described in Subsection 3.2, can avoid many ray–AABB tests by only testing rays until it finds i_a, the index to the first alive packet ray. The key to success is the probability that most active rays after i_a are also alive deeper in the BVH, particularly at the BVH leaves. This tends to happen for primary visibility resulting in fewer total ray–AABB in the bottom left of Figure 6 and faster render times in the top row of Figure 5.

However, if the active rays after i_a aren't truly alive, Ranged traversal may suffer big overheads. An incoherent ray packet may avoid some ray–AABB tests higher in the tree only to have to perform them deep in the tree where there are exponentially more BVH cells, causing the higher peak at deeper BVH cells in the histograms in Figure 6. Even worse, if dead rays reach the leaves, there will be many more expensive ray–triangle tests as shown at the bottom of Figure 6. For the more coherent primary ray packets on the left, these extra ray–triangle tests are acceptable since ray–AABB tests dominate ray casting time, but the less coherent secondary ray packets on the right lead to many extra ray–triangle tests and slower render times in the highlighted plots in Figure 5.

5.3 Frustum Culling for Whitted Ray Tracing

In this Subsection, we evaluate frustum culling for all ray types in a Whitted ray tracer using the algorithms in Section 4 to construct tight bounding frusta. All results in this Subsection were generated using Partition traversal since this presents the most stable baseline.

Figure 7 demonstrates the benefits of frustum culling. As in Subsection 5.2, each plot shows time to render one image versus ray packet size, and the plots are organized left to right in order of increasing scene complexity and top to bottom in order of increasing ray recursion complexity.

In general, frustum culling works best for primary visibility and point-light shadows (the top two rows in Figure 7). There is some benefit for frustum culling on reflection and refraction rays for the relatively simple ERW6 and Toasters scene, but barely any noticeable benefit for the more complex Fairy and Rings scene. We will see in Section 6 that the results for the Toasters scene is more representative of most rendering configurations where not all surfaces are reflecting and/or refracting.

We expect good results for primary rays from a pinhole camera, but it is less clear why culling for point-light shadow rays is more effective than for reflection and refraction rays. Point-light shadow rays converge at the light source which leads to tighter ray packets and hence tighter ray bounds. Reflections and refractions, on the other hand, tend to diverge making it significantly more difficult to generate tight bounding frusta.

5.4 Packet Traversal and Frustum Culling: Summary

We summarize our conclusions for ray packet traversal and frustum culling in Table 1. Masked traversal is superseded by Ranged and Partition traversal. Ranged traversal is the best for primary visibility on packets of up to 16×16. For secondary rays, Ranged traversal tends to be the best on packets of up to 8×8, but runs the risk of falling to the pressure of increased scene and ray recursion complexity. Partition traversal should be used for secondary rays in systems that require large ray packets, complex scenes, or where ray–geometry intersection tests dominate ray casting time. Alternatively, Partition traversal can be used for *all* secondary rays as a conservative measure to avoid the pitfalls of Ranged traversal.

Frustum culling works well for primary visibility and point-light shadows providing up to $2\times$ benefit. The results for reflection and refraction rays are less impressive, speeding up ray casting times mostly for relatively flat and smooth surfaces. Frustum culling does help, generally by about $1.2\times$–$1.3\times$, but it should not be relied upon to achieve real-time performance.

6 Results–Performance

In this Section, we construct our real-time Whitted ray tracer and evaluate its performance. We find that the analysis in Section 5 leads to robust real-time performance, and large ray packets offer significant benefits over SIMD rays for Whitted ray tracing.

Based on the recommendations from Section 5 and Table 1, our ray tracer uses 16×16 ray packets with Ranged traversal for primary rays, and we choose between Partition and Ranged traversal for reflection, refraction, and shadow rays based on the scene and ray recursion complexity. We use frustum culling for all ray packets. All hardware configurations used to generate results in this Section are the same as in Section 5.

Figure 1 shows several scenes and ray recursion configurations rendered with our real-time Whitted ray tracer. These images tend to render significantly faster than those from Section 5 because only select surfaces are set to be reflective or refractive. The Toasters scene is the same scene from Section 5, but we have set 1-bounce reflections for the floor and turned on point-light shadows. The Fairy scene is also used in Section 5, but we have set 1-deep refractions on the wings, and 1-bounce reflections on the forest floor making it appear as if the fairy is sitting on water. While it isn't particularly noticeable from this view, reflections are turned on for the fairy's eyes as well as the gold and jewels on her wand.

Figure 1 also includes two scenes from the BART [5] collection. We render only the first keyframe from these sequences. The museum image demonstrates deep reflections with 3-bounces. The kitchen scene uses 1-bounce reflections, 4-deep refractions, and point-light shadows from one point-light. Notice the refractions through the bowls and glasses on the table as well as the dragon model under the table. Light even refracts through the dragon's reflection.

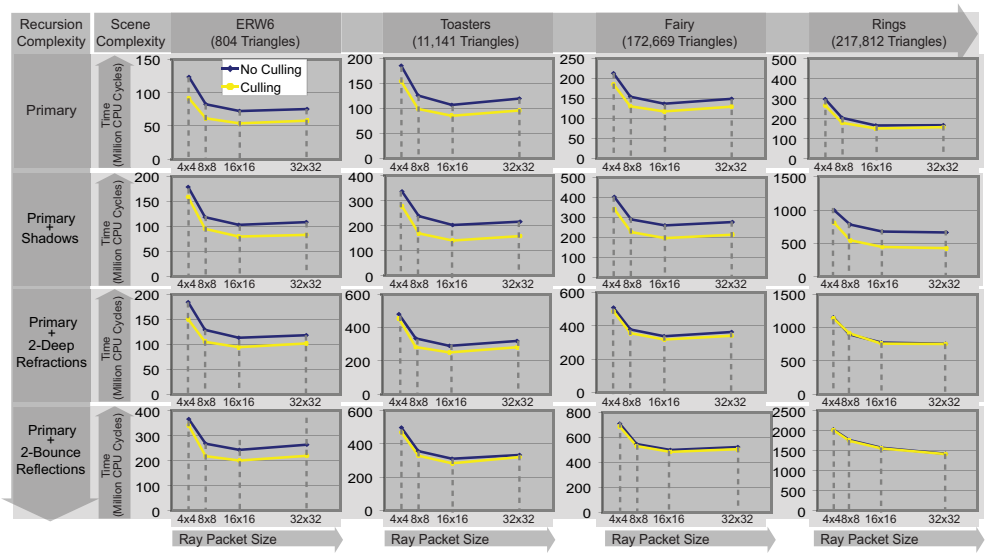

Figure 7: Times for rendering one 1024×1024 image with and without frustum culling with varying packet size, scene complexity, and ray recursion complexity. Frustum culling works best for primary rays and point-light shadow rays, and mostly helps reflections and refractions off of flat and smooth surfaces.

	Toasters	Fairy	BART Museum	BART Kitchen
2×2 SIMD Ray Packets	1.9 FPS	2.1 FPS	2.4 FPS	1.2 FPS
16×16 Ray Packets	11.8 FPS	6.7 FPS	8.5 FPS	4 FPS
Performance Benefit	**6.1** ×	**3.2**×	**3.5**×	**3.3**×
Reflect+Refract Traversal	Ranged	Partition	Partition	Partition
Culling Benefit	1.35×	1.19×	1.18×	1.19×

Table 2: Comparison of our large ray packet tracer using 16×16 packets against 2×2 SIMD ray packets for rendering the images in Figure 1.

We compare to 2×2 SIMD ray packets in Table 2. This table presents times for rendering the images in Figure 1 for SIMD rays and our large ray packet tracer and whether secondary rays use Partition or Ranged traversal. In all examples, large ray packets provide at least 3× faster render times over SIMD Rays and up to 6× for the simpler Toaster scene.

In the last row of Table 2, we include the performance benefit due solely to frustum culling. For these configurations, frustum culling improves performance by about 18%–35% which is significantly more than reported in Section 5 for the Fairy and Rings scene with reflection and refraction rays. As is usually the case in Whitted ray traced scenes, the reflective and refractive surfaces used in this Section tend to be significantly flatter and smoother leading to better culling performance.

7 Conclusion

This paper introduces a fully real-time CPU-based Whitted large ray packet tracer. Entering this new domain required serious analysis of large ray packet algorithms for scene traversal and frustum culling. It also required the new Partition traversal algorithm and a new approach for generating frustum bounds around reflection and refraction rays. The result is a real-time Whitted large ray packet tracing system which is robust to degrading coherence.

We thoroughly evaluated ray packet algorithms for frustum culling and three BVH traversal algorithms in the context of real-time Whitted ray tracing. There are a large number of possible combinations of these algorithms, and in the process of this work, we evaluated many of them which are not presented. We found that the simple solutions work best and believe the algorithms presented here most concisely encompass the results of our research.

Distributed ray traced effects are also likely to benefit from our work. Here we focus on Whitted ray traced effects to push them into real-time, but real-time results remain out of reach for distributed ray tracing. Based on the results in this paper, we believe this class of effects requires new algorithms beyond ray coherence based techniques to join the interactive domain.

Acknowledgements

This work was supported in part by the NSF (grants CCF 03-05322, CCF 04-46916, CCF 07-01775), a Sloan Research Fellowship, and an ONR Young Investigator Award N00014-17-1-0900. We also acknowledge an Intel fellowship to Ryan Overbeck and related equipment donations from Intel and NVIDIA. Fairy scene provided by DAZ Productions via the Utah 3D Anim. Repo.. Thanks to the anonymous reviewers as well as Kevin Egan, Craig Donner, Sean Keely, and Warren Hunt for their helpful comments.

References

[1] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. Packet-based Whitted and Distribution Ray Tracing. In *Proc. Graphics Interface*, May 2007.

[2] S. Boulos, I. Wald, and P. Shirley. Geometric and Arithmetic Culling Methods for Entire Ray Packets. Technical Report UUCS-06-010, 2006.

[3] E. Haines. A proposal for standard graphics environments. *IEEE Computer Graphics & Applications*, 7(11), 1987.

[4] W. Hunt, W. R. Mark, and D. Fussell. Fast and lazy build of acceleration structures from scene hierarchies. In *IEEE Symp. on Interactive Ray Tracing*, 2007.

[5] J. Lext, U. Assarsson, and T. Moeller. Bart: A benchmark for animated ray tracing, 2000.

[6] E. Mansson, J. Munkberg, and T. Akenine-Moller. Deep coherent ray tracing. *IEEE Symp. on Interactive Ray Tracing*, 2007.

[7] A. Reshetov. Faster ray packets - triangle intersection through vertex culling. In *IEEE Symp. on Interactive Ray Tracing*, 2007.

[8] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM TOG SIGGRAPH 05*, 24(3), 2005.

[9] I. Wald. On fast Construction of SAH based Bounding Volume Hierarchies. In *IEEE Symp. on Interactive Ray Tracing*, 2007.

[10] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM TOG*, 26(1), 2007.

[11] I. Wald, C. P. Gribble, S. Boulos, and A. Kensler. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Technical Report UUSCI-2007-012, 2007.

[12] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM TOG SIGGRAPH 06*, 2006.

[13] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*, 2007.

[14] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3), 2001.

[15] S.-E. Yoon, S. Curtis, and D. Manocha. Ray tracing dynamic scenes using selective restructuring. In *EGSR*, 2007.

A Appendix

A.1 Ranged Traversal

```
1:  // Traverse a Ray packet, R, through theBVH using Ranged Traversal
2:  void rangedTraverseBVH( Rays R, Frustum F, BVH theBVH )
3:      BVHCell curCell = theBVH.root;
4:      Stack<StackNode> traversalStack;
5:      Index i_a = 0;
6:      while ( true )
7:        i_a = getFirstHit( R, F, curCell.AABB(), i_a );
8:        if ( i_a < size(R) )
9:          if ( isInner( curCell ))
10:           StackNode node;
11:           node.cell = curCell.farChild( R );
12:           node.i_a = i_a;
13:           traversalStack.pushBack( node );
14:           curCell = curCell.nearChild( R );
15:           continue ;
16:         else  // isLeaf( curCell ) == true
17:           Index i_e = getLastHit( R, curCell.AABB(), i_a );
18:           Triangles T = curCell.triangles();
19:           for ( Index j = 0; j < size( T ); ++j )
20:             if ( frustumIntersectsTriangle( F, T[j] ))
21:               for ( Index i = i_a; i < i_e ; ++i )
22:                 rayIntersectTriangle( R[i], T[j] );
23:       // END if ( i_a < size(R) )
24:       if ( traversalStack.empty() )
25:         break ;
26:       StackNode node = traversalStack.pop();
27:       curCell = node.cell;
28:       i_a = node.i_a;
29:     // END while ( true )...
30: // END void traverseBVH(...
```

A.2 Partition Traversal

```
1:  // Traverse a Ray packet, R, through theBVH using Partition Traversal
2:  void partitionTraverseBVH( Rays R, Frustum F, BVH theBVH )
3:      BVHCell curCell = theBVH.root;
4:      Stack<StackNode> traversalStack;
5:      Index I[size( R )];
6:      for ( Index i = 0; i < size(R); ++i ) I[i] = i;
7:      Index i_a = 0;
8:      while ( true )
9:        i_a = partRays( R, F, curCell.AABB(), I, i_a );
10:       if ( i_a > 0 )
11:         if ( isInner( curCell ))
12:           StackNode node;
13:           node.cell = curCell.farChild( R );
14:           node.i_a = i_a;
15:           traversalStack.pushBack( node );
16:           curCell = curCell.nearChild( R );
17:           continue ;
18:         else  // isLeaf( curCell ) == true
19:           Index i_e = getLastHit( R, curCell.AABB(), i_a );
20:           Triangles T = curCell.triangles();
21:           for ( Index j = 0; j < size( T ); ++j )
22:             if ( frustumIntersectsTriangle( F, T[j] ))
23:               for ( Index i = 0; i < i_a ; ++i )
24:                 rayIntersectTriangle( R[I[i]], T[j] );
25:       // END if ( i_a > 0 )
26:       if ( traversalStack.empty() )
27:         break ;
28:       StackNode node = traversalStack.pop();
29:       curCell = node.cell;
30:       i_a = node.i_a;
31:     // END while ( true )...
32: // END void traverseBVH(...
```