UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Ray Packet Tracing with Acceleration using AVX

A Thesis submitted in partial satisfaction of the requirements for the degree of Masters of Science

in

Computer Science

by

Franklin Wu

Committee in charge:

Professor Henrik Wann Jensen, Chair
Professor Scott B. Baden
Professor Dean M. Tullsen

2016

The Thesis of Franklin Wu is approved and it is acceptable in the quality and form for publication on microfilm and electronically:

---

---

---

Chair

University of California, San Diego

2016

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Henrik Wann Jensen for his support as chair of my committee. His guidance has proved to be invaluable to me.

ABSTRACT OF THE THESIS

Ray Packet Tracing with Acceleration using AVX

by

Franklin Wu

Master of Science in Computer Science

University of California, San Diego, 2016

Professor Henrik Wann Jensen, Chair

The most time consuming operation of ray tracing is the intersection of rays. Since many rays are traced, the use of single instruction multiple data, or SIMD, instructions (such as the 4-wide SSE) to trace multiple rays simultaneously have been used. However, the incoherence of secondary rays, especially reflection and refraction rays, results in low utilization of the SIMD instructions' full width. In this thesis we look at using the AVX instruction set which expands upon SSE to allow twice as many rays, but exacerbates the problem of low utilization. We present a method of packaging secondary rays to improve ray coherence within the packet to improve SIMD utilization. While the improvement is only incremental

we believe the work is important as we see a trend of wider SIMD instructions such as AVX-512.

# Introduction

One of the key elements to achieving fast ray tracing is efficient ray primitive intersection tests. Naive testing of all rays with all primitives result in $O(nm)$ time where $n = (\#$ of rays) and $m = (\#$ of primitives), which can quickly become very expensive.

The first improvement to this was tracing rays one at a time against an acceleration structure, for example a BVH (bounding volume hierarchy), which is akin to traversing down a search tree. This greatly reduced the runtime complexity to logorithmic with respect to the number of primitives, however this is still not adequate as the golden standard for raytracing has been realtime performance.

Consider an image with a resolution of 1024x1024. In such a case, even tracing one ray per pixel, it would require over 1 million rays traced. However, creating a realistic rendering often requires many rays traced per pixel, up to thousands of rays or more, which means potentially billions of rays traced. Ray packets is one technique that was introduced to further reduce trace time.

In ray packet tracing, rather than tracing a single ray through an acceleration structure, multiple rays are traced through it in a single pass. By tracing multiple rays at once, two main benefits we gain are reduction in memory bandwidth for the acceleration structure and exploitation of the similarity in rays. Ray packet tracing is not without its difficulties though. It is not a trivial task to select rays to place in a packet. By randomly selecting rays we would likely pick a set of mostly incoherent rays, or rays with dissimilar origins and directions, which

result in worse performance compared to single ray tracing due to the divergent traversals of the acceleration structure.

Due to the problem of incoherent rays, packets have generally been kept small. In addition large packets exacerbate the problems associated with incoherent rays. Conversely, smaller ray packets reduce the benefits that ray packets provide. More recently, techniques have been introduced by Boulos et al. [4] and by Overbeck et al. [11] which perform well on larger ray packets, with the latter showing promise on packets with up to 1024 rays even in scenes with high complexity.

Relevant to ray packet tracing is the paradigm of SIMD instructions as the same intersection operation is performed on multiple rays within a packet in parallel. Of particular importance is the streaming SIMD extensions, or SSE, instruction set introduced by Intel for their CPUs which started with 128-bit registers and subsequently increased in width. Theoretically, representing rays using 32-bit floating point, there would be a 4x speedup when using SSE instructions not counting overheads required for using SSE registers. Combined with the problem of ray coherency, it is difficult to reach the theoretical 4x speedup. The problems become worse with advanced vector extension (AVX) and AVX-512 which are 256-bit and 512-bit successors to SSE.

To address the issue of ray coherency with SIMD instructions, Wald et al. [18] explored a theoretical SSE instruction set with scatter and gather operations in hopes that they would eventually become part of the SSE instruction set. Others shifted the paradigm from intersecting *multiple* rays with an acceleration structure to ones that intersect *single* rays with an acceleration structure with *higher number of branching paths*. This takes advantage of SIMD instructions by testing against all branches simultaneously. Ernst et al. [7] first introduced this for SSE by extending BVHs to what they called multiple bounding volume hierarchies (MBVH). This was later extended to AVX by Attila [15].

In light of increasing SIMD widths, our work focuses on improving the per-

formance of large ray packets by addressing the task of constructing coherent ray packets rather than the paradigm of increasing the number of branches. We take this approach for two reasons.

1. Large ray packets have a higher ray-to-acceleration-structure ratio than acceleration-structure-branch-to-ray-ratio of MBVHs. Thus large ray packets have more potential for speed improvements.

2. MBVHs have the same diminishing speedups as the SIMD width increases. Thus improvements on MBVH would still depend on ray coherency.

As such we hope to reduce the diminishing speedups of increasing SIMD width by focusing on improving ray coherency.

In Chapter 2, we discuss the current state of ray packet tracing and use of SIMD instructions which serve as the inspiration for our work. Then in Chapter 3 we detail our implementation. Specifically we look at using Intel SSE and AVX instructions to trace multiple rays while using large ray packets. To efficiently construct ray packets we introduce what we call queued tracing. In Chapter 4, we analyze our results and in Chapter 5 we discuss potential future work.

# Previous Work

## 2.1 Ray Packet Tracing

When ray tracing, multiple rays with similar properties can be bundled together such that they can all be traced in a single pass. This gives multiple advantages, which includes better memory usage and reduction of operations by testing multiple rays in a single operation. For example, the rays can be encompassed by a bounding frustum; if the frustum does not intersect an object, then no rays within the frustum need be tested.

Even with these advantages ray packet tracing has been generally restricted in size due to the issue of incoherent rays. Traditionally packets were limited to sizes of 2x2 or 4x4, but methods have been introduced for larger ray packets. Many of these methods take advantage of SIMD instructions to parallelize the ray intersection operations.

### 2.1.1 Naive Traversal

Two of the earlier packet techniques introduced by Wald et al [19] and Reshetov et al [14] were both based on KD-trees. At each stage of the traversal, the rays stored the current maximum and minimum segment of the currently valid ray. If the intersection of the KD-tree node was outside of the segment range, then the ray would not be tested further against the node. These techniques were found to be only useful for up to 2x2 and 4x4 sized packets respectively.

### 2.1.2 First Active Ray Traversal

Later Wald et al [17] improved upon ray packet tracing using a BVH tree method, which they called *first active ray tracking*. At each traversal step, the set of rays are only tested up until the first ray hits the bounding volume, at which point traversal continues down the tree. The ray that hits the bounding volume is tracked so that in the child node it is the first ray to be tested since all previous rays are known to miss the child node. This reduces the number of redundant ray checks that were required by the naive traversal methods discussed in the previous section. It was found that this traversal method handled packets of up to 16x16 well, but in practice it was found that 8x8 packets were optimal.

### 2.1.3 Adaptive Ray Packet Reordering

As noted by Boulos et al [4], methods such as *first active ray tracking* is speculative in nature, and is in essence a greedy like algorithm. By only testing until the first ray hit, it is likely that missed rays higher in the tree are unnecessarily tested multiple times near the bottom of the tree. Hence, *adaptive ray packet reordering* was introduced, which modified its traversal behavior based on the SIMD utilization and packet utilization.

At each traversal step a all-hit or all-miss operation is done, which continues down the tree or exits early as expected. If neither case holds then the number of active rays is determined. If the number of active rays is below a pre-determined threshold, then the rays are reordered and new bounding frustums are calculated and the traversal continues with the reduced packet. However, if the number of active rays are above the threshold, then traversal continues with those rays using a naive traversal.

This makes the traversal no worse than a naive traversal, but has the advantage of filtering out incoherent rays. Adaptive ray packet reordering was found to perform well on packets of size up to 16x16.

### 2.1.4 Partition Traversal

Overbeck et al [11] introduced a method of tracing large ray packets called *partition traversal*, which performed well on packets sized up to 32x32. They addressed the issue of tracking which rays being traced are still active. Previous methods either required iterating through a live ray mask or deferring ray intersection to avoid iteration through a mask. Partition traversal avoids redundant operations by iterating through an array of ray indices rather than iterating through an array of rays. By reordering the ray indices at each traversal step as shown in Algorithm 2.1 and 2.2, all live rays placed at the beginning of the array. As such it is possible to simply iterate only until the last live ray and skip operations on the inactive rays.

Indices are used because reordering indices is cheaper than moving ray data. At minimum, the ray would require 6 floats (3 for origin and 3 for direction) compared to 1 integer for the indices. In this way *partition traversal* is able to exactly filter out inactive rays with low cost.

Of course *partition traversal* is not without its problems. It brings back the issue in naive traversal where all rays are tested at each traversal step, whereas a technique such as *first active ray* traversal avoids testing all the rays. The advantage of *partition traversal* is seen in more complex scene setups where rays tend to be less coherent. Our work will be further examining and expanding upon *partition traversal*.

## 2.2 Frustum Culling

If at each step of a ray packet traversal all rays need to be traced, then the total number of intersection tests done would be equal to tracing single rays one at a time. Of course there is the benefit that the acceleration structure nodes need only be loaded into memory once for all the rays in the packet rather than once

**Algorithm 2.1** traverseBVH

```
1: procedure TRAVERSEBVH(p)
2:     curNode ← BVHroot
3:     nodeStack                          ▷ contains node and index of max alive ray
4:     while true do
5:         ia ← partRays(rays, frustum, curNode, index, indices)
6:         if ia > 0 then
7:             if curNode.isInternalNode() then
8:                 nodeStack.push(curNode.leftChild, ia)
9:                 nodeStack.push(curNode.rightChild, ia)
10:            else
11:                leafFrustum ← createLeafFrustum(rays.alive(indices, ia))
12:                for obj ← curNode.obj_0, curNode.obj_n do
13:                    if obj.intersects(leafFrustum) then
14:                        for ray ← rays.alive(indices, ia) do
15:                            obj.intersect(ray)
16:                        end for
17:                    end if
18:                end for
19:            end if
20:        end if
21:        if nodeStack.isEmpty() then
22:            break                                  ▷ break out of while loop
23:        end if
24:        (curNode, index) ← nodeStack.pop()
25:    end while
26: end procedure
```

**Algorithm 2.2** partRays

```
1: procedure PARTRAYS(rays, frustum, node, index, indices)
2:     if node.intersects(frustum) then
3:         return 0
4:     end if
5:     ie ← 0
6:     for ray ← rays.alive(indices, index) do
7:         indices.swap(ie, ray.index)
8:         ie ← ie + 1
9:     end for
10:    return ie
11: end procedure
```

for each ray. With packet traversal, another advantage is reducing the number of intersection operations. By finding a bounding primitive enclosing the rays, it is then possible to do a single operation to cull all rays from intersecting a node of the acceleration structure. The efficacy of such a technique depends on the coherency of the rays within the packet.

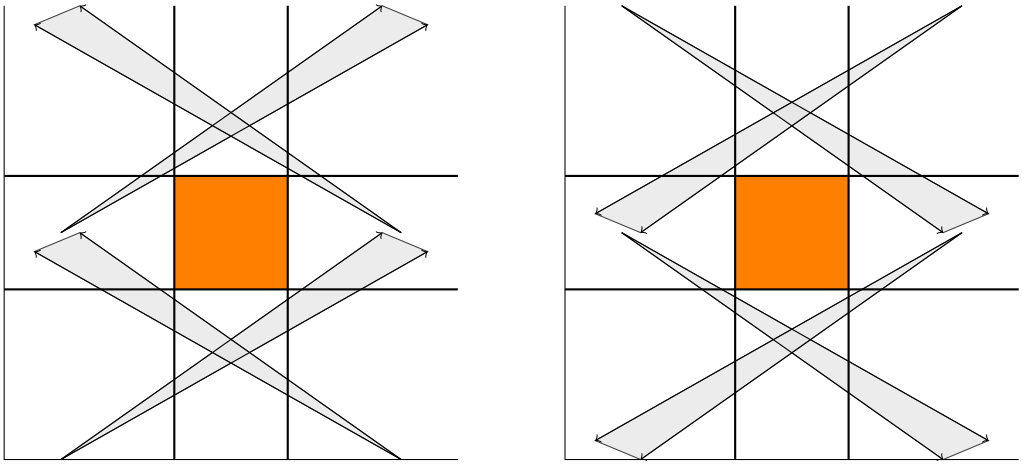
The culling techniques depend on the type of bounding volume used for the acceleration structure and the rays. Most commonly, a frustum is used to bound the rays. A frustum is defined as a volume of area between two planes. This can be represented either using a set of rays or set of planes. There is no limit to the number of rays or planes used to define a frustum and by increasing the number of rays or planes, it is possible to define a tighter frustum leading to more culls. However, this number is typically limited to 4 as there is diminishing returns on increasing the number of rays or planes. One of the more increasingly popular acceleration structures has been the BVH, hence we focus our discussion on this particular setup.

### 2.2.1 Corner Rays and AABB

Reshetov et al [14] discuss a frustum culling technique with the frustum defined as a set of rays with a common origin. They discuss this for a KD-tree, however the test at the leaf nodes reduce to an axis aligned bounding box (AABB) which is frequently used as the bounding volume in a BVH.

Here, the rays are projected onto one of the axis as shown in Figure 2.1. Then the entry and exit hit points for the axis are calculated. If the two following conditions are found to be true, then it can be concluded that the frustum misses the AABB:

1. (minimum of y-entry values) > (maximum of x-exit values)
2. (minimum of x-entry values) > (maximum of y-exit values)



(a) $Ray_y > 0$ (b) $Ray_y < 0$

Figure 2.1: Frustum AABB culling - If all rays of the frustum intersect all of one set of parallel bounding planes prior to intersecting the other set of bounding planes, then the frustum misses the bounding box. For example, if all the rays hit all the $x$ bounding planes prior to hitting a $y$ bounding plane.

### 2.2.2 Interval Arithmetic

To perform culling of a triangle from a frustum Boulos et al [5] discuss interval arithmetic. Given that the frustum is represented using corner rays, it is possible to find the barycentric coordinates of the ray intersection with the triangle. If one component of the barycentric coordinates for all the rays is negative then it can be concluded that the entire frustum does not intersect with the triangle. This comes with the caveat that the intersection of the rays with the triangle plane is in the positive ray direction.

### 2.2.3 Vertex Culling

Further triangle culling has been described by Reshetov [13] where they reconstruct the bounding frustum at the leaf node. The frustum formed when constructing the ray packet contains all rays including those which miss the current node. This makes the original frustum much looser than than needed for the currently live rays. By reconstructing the frustum, a much tighter frustum is created which better culls triangles. Potentially the two frustum could be similar,

however this is desirable because it means that there is high ray coherency in the packet.

To form the new frustum a dominant axis is chosen. Then, the rays are intersected with the bounding planes formed by the two other axes. The minimum and maximum intersections of those axises on the planes are found and used to form the corner rays of the frustum. This forms 4 corner rays which are used to determine 4 bounding planes for the frustum. Triangles are represented as 3 vertices or points in space. A simple distance test checks what side of the bounding plane the vertices are on. If all 3 vertices of the triangle are on the side of a plane away from the frustum then it can be concluded that the triangle does not lie inside the frustum.

## 2.3 SIMD

Since the 70's the single core performance of CPUs have continually increased allowing free increases in raytracing performance. However, since around 2003 CPU designers have turned to increasing the number of cores per CPU rather than increase CPU clock speed. As such single threaded raytracing techniques no longer have free increases in performance.

Another alternative to increasing single CPU performance has been increasing the width of CPU instructions with SIMD operations. Streaming SIMD Extensions (SSE) instructions were first introduced by Intel in 1999 with 128-bit registers capable to handling 4x32-bit floating point numbers in a single register. In 2008, AVX which had 256-bit registers was proposed and in 2011 the first CPUs that supported AVX were introduced. In July 2013, a 512-bit extension to AVX, AVX-512 was introduced with the first processors supporting the new ISA set to come out in 2015.

Considering the trend of increasing SIMD width it is important to understand how to best utilize SIMD registers and what limits it presents.

### 2.3.1 Interactive Ray Tracing

The earlier works on ray packet tracing by Wald et al. [19] explored using SIMD instructions to trace multiple rays at once against single objects. They used SSE to trace 4 rays at a time and found a 3.5-3.7x speedup against a single ray implementation. However, this speedup only holds when there is perfect coherence for the rays, i.e. all rays are active during the traversal of the acceleration structure. As packet size was increased, the overhead due to incoherence increased. In contrast, increasing image resolution decreased the overhead, because it leads to tighter primary rays. This is beneficial as increasing image resolution also leads to better image quality, hence there is an expectation of good scaling when increasing image quality.

The downside of this work is that only primary, shadow, and reflection rays were implemented. The test setups contained a relatively high percentage of primary rays and shadow rays. If tested on a scene with a good number of reflective and refractive surfaces, the majority of the rays would be reflection and refraction rays which would not have the high coherence desired. Even with these issues, it was shown that ray tracers could achieve interactive ray tracing speeds, with the hope that at some point ray tracers could replace rasterization hardware.

### 2.3.2 SIMD Ray Stream Tracing

Wald et al. [18] expressed concerns over incoherent rays which would result in low utilization of the SIMD operations and proposed the idea of streamed SIMD instructions. The problem with incoherent rays is that it leads to operations where only 1 out of the $n$ rays in the SIMD register are still active. This means that the utilization of the SIMD register is only $\frac{1}{n}$, with wider SIMD registers suffering from more degradation of performance. When combined with the overheads required for using SIMD registers and operations it becomes more costly to use SIMD registers than to do single ray tracing.

To solve the issue of low utilization would require inactive rays to be filtered out or sorted so that all active rays are contiguous in memory. However, shuffling of rays could be costly and complicated. Hence, it was predicted by Wald et al. that eventually gather and scatter operations akin to those in the parallel programming paradigm would be introduced into SSE and other such SIMD extensions. The idea is that rays are processed in chunks or streams; then as the chunks are processed the gather operation would easily allow active rays to be gathered into another stream to be processed; thus, increasing the utilization of SIMD operations on the stream.

Since this was a predicted extension, they simulated the instruction and found a wide variation in utilization ranging from 99% to 13% depending on the operation. Even though there were cases of low utilization, it proved to be an improvement upon naive packet tracing. Unfortunately, the predicted operations never became part of SSE.

### 2.3.3 Multiple Bounding Volume Hierarchy

Due to the complexity of constructing coherent ray packets, Ernst et al. [7] introduced the multi bounding volume hierarchy (MBVH) which increased the number of branches in a BVH. At each traversal step in a traditional BVH there are 2 branches, while there are 4 branches in Ernst's MBVH. By using SSE instructions, all 4 branches are tested at once. Compared to a ray packet scheme, the branch test operation has nearly perfect SIMD utilization.

It was found that a ray packet tracer using a BVH degrades much quicker than a MBVH tracing a single ray. In fact, Ernst's work found that ray packet tracers performed worse than single ray tracers as the incoherence increased. On the other hand, a MBVH degrades approximately at the same rate as a traditional single ray tracer, as it is the same except with a higher branching factor.

The MBVH was further explored by Wald et al. [16] and extended to AVX

by Attila [15]. Attila found that increasing the MBVH to a branching factor of 8 decreased the average number of traversal steps and intersection tests by 1.52x relative to one with a branching factor of 4. However, due to the overhead of using wider registers and the increased cache misses due to an increase in data read, the actual average speedup was less than 1.52x. Their tests scenes showed anywhere from 2-25% speedup.

## 2.4 Ray Coherency

In order for a ray packet to be efficient, it is necessary that the rays within the packet are similar. This would affect how tight of a bounding frustum we can define for the packet. Formally stated, we want to have a set of *coherent rays*. The ideal would be that rays within a packet have origins that are close and have similar directions.

### 2.4.1 5D Classification of Rays

A ray in 3D space is usually defined as an origin, $o$, and a direction, $d$, each of which contains 3 components.

**Definition 2.4.1.** A ray, $R = (o, d)$, where:

$o = (o_x, o_y, o_z)$ is the origin

$d = (d_x, d_y, d_z)$ is the direction

Using this definition, rays are contained within a 6D space, which makes determining coherency of rays analogous to a 6D sorting problem.

However, Arvo et al. [3] noted that while rays are traditionally represented with 6 values, they only had 5 degrees of freedom. With the origin still composed of a 3D vector, the direction can be represent as a 2D vector of spherical angles. In this representation, rays are in the $\boldsymbol{R}^5$ space. So now, instead of dividing into traditional bounding volumes, the rays can be split into 5D hypercubes. Using

this schemei, rays are separated into distinct sets of 5D hypercubes that intersect only with specific sets of objects in 3D space. As a result, this reduces the number of ray-object and ray-bound intersection tests.

The issue with 5D rays is that they utilize a less intuitive representation. Due to the complexity of reasoning with 5D rays, not much work has been done using them. Using 3D origin and direction remains the most popular form for rays to be represented.

### 2.4.2 Ray Grouping Heuristics

Returning to 6D rays, Månsson et al. [9] focus on methods of regrouping rays in order to increase coherency. Ray packets can be formed with any arbitrary set of rays. However, it would likely result in large bounding volumes which are inefficient at culling. By grouping up rays cleverly, the bounding volume holding the ray packet can potentially be significantly reduced in size. Grouping rays come at a cost because a portion of the tracing time becomes devoted to sorting rays. The following formula can be used to determine whether grouping is cost efficient: $time_{sort} + time_{packet\_trace} < time_{single\_trace}$.

In their work, Månsson tested 6 different sorting heuristics:

- **none** - No regrouping is done. This is the standard and was used as the basis for measurement.
- **dir** - Rays are grouped into 8 sets based on the sign of the 3 components of the direction vector.
- **mdir(x)** - Rays are grouped similarly to dir, but are additionally divided into $x$ cells on a direction cube meaning each groups covers a smaller solid angle as $x$ increases.
- **fastpos** - Rays are grouped into octants of the scene bounding box using the ray origin.
- **pos(x)** - Rays are sorted into packets where new rays are added to packets with the closest origin based on distance.

- **opos** - Rays are sorted into lists based on sign direction. Packets are formed by extracting a ray, then using the lists to find other rays closest in origin.

They found that regrouping rays did not help increase performance by much. In fact, many of the grouping schemes reduced performance relative to the **none** heuristic. This meant that the rays tested already had relatively good coherency by nature so regrouping did not have a significant impact. In addition, this meant that the sorting cost of these heuristics were too expensive relative to the performance gains. Further, Månsson et al. found that even in test cases where the **none** heuristic had a high decline in coherency, that the other heuristics were not able to mitigate the coherency decline well either.

### 2.4.3 Ray Voxel Scheduling

With methods described in Section 2.4.2, the base runtime of the tracing routine is determined by $time_{total\_trace} = time_{sort} + time_{packet\_trace}$. When tracing packets with higher coherency $time_{packet\_trace}$ is reduced, however it comes at the cost of increasing $time_{sort}$. Increased coherency reduces trace time for two reasons. First, the bounding volumes of the ray packets become tighter leading to more culling against scene primitives. Secondly, coherent rays tend to intersect a smaller subset of primitives compared to incoherent rays. As a result less geometry needs to be loaded and thus improves memory bandwidth.

By observing the reduced memory bandwidth from coherent rays, Pharr et al. [12] devised voxel queues to render scenes using memory coherent rays. Rather than forming a ray packet and then fully tracing them through the scene, the scene is divided into voxels which they call the *scheduling grid*. Rays are queued up in voxels based on their origin. Then, rays are traced against the geometry within the voxel to determine intersections. If an intersection is found, then shading operations are done on the ray, otherwise the ray is placed in the next voxel that it hits.

By tracing rays within a voxel at the same time, only geometry contained within the voxel needs to be in cache. This reduces the loading of geometry if the geometry along with all the other data fits within the cache. As the amount of geometry increases, thrashing occurs and the voxel scheduling becomes useless. In addition to tracing within the voxel, care is taken to choose voxels that have the most benefit. A voxel with more rays has higher priority as more rays will be tested against the geometry leading to better reuse of the geometry cache. In addition, a voxel which has more of its geometry within the cache should be chosen, because the geometry does not need to be loaded again. However, determining what geometry is within the cache is not an easy problem.

# Methodology

## 3.1 Motivation and Objective

Considering the hardware trend of increasing register sizes for SIMD instructions, we wanted to test the limits that would be encountered. For reasons mentioned in Section 1, we base our work off of the large ray packet techniques introduced by Overbeck et al. [11], rather than MBVH techniques.

In *partition traversal*, sets of 4 rays were treated as a ray primitive to be traced. It would be assumed that operations on the rays are implemented using SSE. It is natural to expand the number of ray primitive to 8 rays with AVX. It can be assumed that there is a theoretical 2x speedup in tracing speed when increasing the number of SIMD rays from 4 to 8. However, as noted in Section 2.4 ray coherency plays an important role in the effectiveness of increasing SIMD width.

To mitigate the problem of ray coherency we consider methods of constructing ray packets with higher coherency. The work of Månsson et al. discussed in Section 2.4.2 implied that ray grouping techniques had no positive impact on tracing speeds. However, with wider SIMD registers the effects of low coherency becomes more problematic, meaning grouping techniques could become more viable and important as wider SIMD registers become available.

## 3.2 Tracing

For our base case, we implement single ray tracing. Due to the recursive nature of single ray tracing the order in which rays are traced differs from ray packet tracing. Admittedly, this results in a differing memory access pattern of the acceleration structure. However, we do not consider this problematic since the packet traversal already forces a change in the acceleration structure traversal. In addition, we are more interested in the reduction ray intersection operations required by increasing ray coherency.

Ray packets are traced in 3 stages starting with primary rays, or rays originating from the camera pointed at the location of the pixel coordinates in 3D space. After all primary ray packets are traced the rays are divided into two sets. Specular rays, or rays which hit reflective or refractive surfraces, and shadow rays, or rays formed from hits on diffuse surfaces. We split the rays into distinct groups as we can exploit particular properties of their origin and direction. The specular rays are formed into packets and traced, with a limitation of a maximum of $n$ bounces. Rays are organized such that only rays which are on the same $i$th bounce are placed into the same packet. Finally after all specular rays are traced the shadow rays are traced. All rays within a single shadow packet are comprised of rays that originate from a single point light source.

We use a bounding volume hierarchy (BVH) as our acceleration structure using the surface area heuristic (SAH). The scenes tested had static geometries, so the BVH was generated beforehand. Thus, the BVH build time was not included in the time measures.

## 3.3 BVH Traversal

We follow the work of Overbeck et al. [11] on *partition traversal*. They use 2x2 ray packets as the smallest ray primitive. Operations on these ray primitives can

**Algorithm 3.1** Packet Tracing

```
1: procedure PacketTrace(p)                          ▷ p is the set of all pixels
2:     while notEmpty(p) do
3:         P ← getPrimaryPacket(p)
4:         H ← tracePacket(P)
5:         Q_0 ← Q_0+ getSpecularHits(H)
6:         PS ← PS+ getShadowHits(H)
7:     end while
8:     for i ← 0, n do                  ▷ n is the max number of specular bounces
9:         while notEmpty(Q_i) do
10:            P ← getSpecularPacket(Q_i)
11:            H ← tracePacket(P)
12:            Q_{i+1} ← Q_{i+1}+ getSpecularHits(H)
13:            PS ← PS+ getShadowHits(H)
14:        end while
15:    end for
16:    for i ← 0, l do                              ▷ l is the number of lights
17:        while n dootDone(L_i,Q_n)
18:            P ← getShadowPacket(L_i,Q_n)
19:            H ← tracePacket(P)
20:            I ← I+ calculateContribution(H)
21:        end while
22:    end for
23: end procedure
```

be efficiently implemented with SSE instructions to run 4-wide SIMD instructions on all 4 rays at a time. We extend *partition traversal* to use AVX instructions which are 8-wide SIMD instructions and choose to replace the 2x2 ray primitives with 2x4 ray primitives, by arbitrarily scaling one of the dimensions.

In addition we implemented 4x4 ray packets to evaluate the SIMD usage potential for the announced AVX-512 extensions which will be available for the Intel Knights Landing processor. However, without a processor that currently supports AVX-512 instructions it is not possible to test for actual performance. However, a glance at the SIMD utilization can give us insight into the potential performance benefits. Given the current trend of increasing SIMD width in CPU designs there is a need for this to be further explored.

The key to *partition traversal* is the *partRays*() function where rays are eliminated from further traversal. As described by Overbeck, whenever the frustum intersects the node all the currently live rays are tested against the node. Then ray indices are reordered so that all indices in the beginning of the array point to live rays. It was noted by Overbeck that *partition traversal* had the downside of redundant ray-bounding volume tests in highly coherent packets compared to first active ray tracking. We address this issue by first testing the four corner rays of the frustum against the node in line 5 of Algorithm 3.2 . If all four corners are hit, then it is guaranteed that all rays will hit the node so it is unnecessary to do the ray elimination tests. Though, it is only useful to perform this test before any rays in the packet have been eliminated unless a tighter frustum is recalculated.

## 3.4 Ray Packet Construction

In order to have optimal ray coherence within a packet, we desire that rays within a packet be as similar as possible. A ray is defined as an origin, $o = (o_x, o_y, o_z)$ and a direction, $d = (d_x, d_y, d_z)$. As such the ray would look to have 6 degrees of freedom, or DOF, but as noted in Section 2.4.1 there is only 5 DOF as

**Algorithm 3.2** partRays

```
1: procedure partRays(rays, frustum, node, index, indices)
2:     if node.intersects(frustum) then
3:         return 0
4:     end if
5:     if node.hitCorners(frustum) then
6:         return index
7:     end if
8:     ie ← 0
9:     for ray ← rays.alive(indices, index) do
10:        indices.swap(ie, ray.index)
11:        ie ← ie + 1
12:    end for
13:    return ie
14: end procedure
```

the direction is fully determinable when constrained to a unit vector. Even still, placing a set of coherent rays into packets means sorting 5D points. If we treat the ray as a 5D point it is possible to solve this sorting problem using a KD-tree to find closest neighbors to the rays. However, this is problematic in 2 ways.

First, purely calculating the Euclidean distance of two rays when treating them as points in 5D space does not necessarily give a good approximation of their coherency. For the sake of simplicity we present this problem using the traditional 6D representation of rays, however the principle still holds in the 5D case.

Given rays defined as $R_i = (o_i, d_i)$ we have 3 rays $o_1 = (1, 0, 0), d_1 = (1, 0, 0), o_2 = (-1, 0, 0), d_2 = (1, 0, 0)$,and $o_3 = (1, 0, 0), d_3 = (-1, 0, 0)$. $R_1$ and $R_2$ should have relatively good coherency since they overlap except at the line segment determined by $(-1, 0, 0)$ and $(1, 0, 0)$. In contrast, $R_1$ and $R_3$ are completely incoherent since they start at the same origin and point in opposite directions. However, the Euclidean distance between $R_1$ and $R_2$ is equal to the distance between $R_1$ and $R_3$.

This is only one example of how Euclidean distance is a poor indicator of ray coherency. In fact, it unknown how to best determine ray coherency.

A second problem with most methods of sorting rays into coherent packets is

the high cost of sorting. Even if there was a suitable distance metric for finding closest neighbors that indicated high coherency it would essentially duplicate tracing rays through an acceleration structure. As a result any performance gains due to higher coherency would be lost due to the sorting cost.

Regardless of whether we use 5D or 6D rays, we run into the same problems mentioned above. However, we can reduce this problem into a simpler task by making a few assumptions. For primary and shadow rays we can choose rays that have the same origin, which are the eye location and the light position respectively. This immediately reduces the problem to sorting over the 3D ray directions. Then it can then be parameterized further into a 2D sort by multiplying the direction vector such that one of the axis has a value of 1. For primary rays it is even less problematic as rays are generated using known pixel coordinates. This makes the task of sorting rays into packets much simpler and is shown in Section 3.4.1 and 3.4.2.

Unfortunately, due to how specular rays are generated they do not share any exploitable common origin. It is still possible to make some assumptions about specular rays to reduce the complexity of the sorting problem, and is described in Section 3.4.3. There we discuss a few different ways to divide specular rays to have better coherency, but it still can not match that of primary and shadow rays. We use 6D rays instead of 5D rays as we believe that the simplicity outweighs the gains we would get from reducing the dimensionality.

### 3.4.1 Primary Rays

Primary ray packets are easily created by dividing the pixels into $n$ x $n$ quadrants to form ray packets of size $n^2$. The rays at the four corners of the quadrant are used to form a frustum that tightly bounds the packet. In our tests we use $n$ that divided evenly into the width and height of the image for simplicity. This results in all primary ray packets being fully filled.

In addition, the rays are generated from pixels in a Hilbert curve order as described by Zhang et al [20]. The Hilbert curve ensures that consecutive rays are closer in proximity, meaning higher coherency, compared to scan lines where consecutive rays could be on opposite sides of the image. Even in a single ray tracing scheme this allows for exploitation of the acceleration data structure still potentially residing in cache. A packet tracer already exploits memory bandwidth and caching by simultaneously tracing rays. Rather, increased coherency in this manner allows for better SIMD utilization.

In fact, this is analogous to the surface area heuristic for constructing a BVH. When considering a packet frustum, a tighter frustum is one which has a smaller perimeter to number of rays ratio. Take for example $n^2$ rays generated by a scan line versus generated by a Hilbert curve, where $n$ is a power of 2. The frustum formed by the scan line will have a perimeter of $2n^2$ pixels while the Hilbert curve rays will have a perimeter of $4n$, giving a ratio of $\frac{n}{2}$. Meaning, as we increase the size of the packets the quality of scan line packets decreases compared to Hilbert curve packets.

### 3.4.2 Shadow Rays

Shadow packets are slightly more complicated than primary rays, but still relatively simple. All light sources in our test setup are point lights. A light source is selected as the origin of the packet. Then all rays for that particular light source are split into 8 sets based on the sign of each component of the direction vector, i.e. (+,+,+) , (-,+,+), (+,-,+), etc.

Then we select an axis as the maximum direction of the set of rays. We intersect the rays with the plane that is a distance of 1 away from the origin along the maximum axis to find the maximum and minimum values of the two other direction components. For example, if the maximum direction was the $x$ component then we find $min_y$, $max_y$, $min_z$, and $max_z$. Then, the four rays

bounding the frustum are $(1, min_y, min_z)$, $(1, max_y, min_z)$, $(1, min_y, max_z)$, and $(1, max_y, max_z)$.

If a particular direction set for the light source has more rays than the max size of a ray packet, then multiple packets are created for that direction. Since ray packets consist only of rays in a specific direction set from a single light source not all shadow ray packets are filled to capacity in the same way as primary rays. In the case where there are multiple packets for a direction there is an opportunity to further sort the rays into better packets, such as further subdividing the directions. However, this requires a sorting the rays a second time. If this were the case it would have been better to have split the light source into more than the 8 sets. This highlights one of the shortcomings of a uniformed grid approaches.

In addition, when the light source is outside of the scene geometry not all of the 8 packet directions are used. It is only when the light source is within the scene geometry that all 8 directions might be used. Hence, when lights are outside of the scene geometry then its rays are just filled into the ray packets with needless sorting.

### 3.4.3 Specular Rays

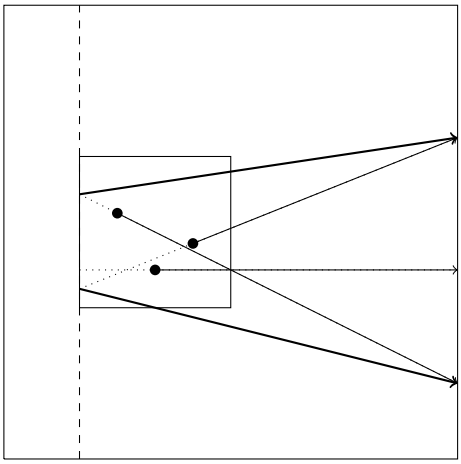


Figure 3.1: 2D Example of Specular Packet Construction with Max Heuristic - The inner box is the BVH queue node, and the outer box is the root BVH node.

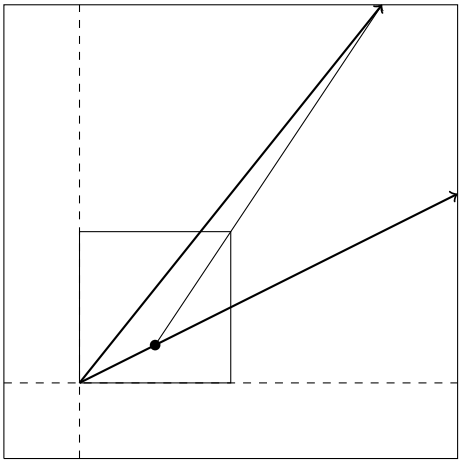


Figure 3.2: 2D Example of Specular Packet Construction with Corner Heuristic - The inner box is the BVH queue node, and the outer box is the root BVH node. Note two vectors are used to create the bounding frustum for the single ray.

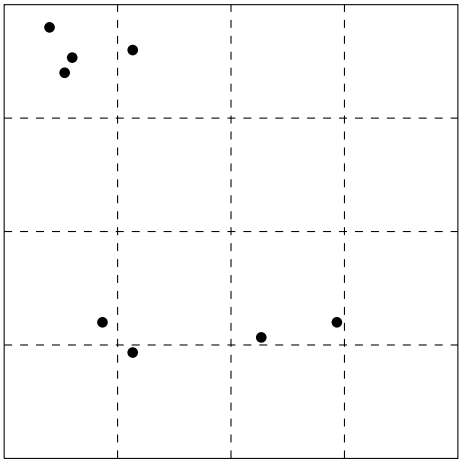


Figure 3.3: Grid for sorting order of rays in packet. The points are the uv coordinates of the ray and box bounding plane intersection. Rays are selected using Hilbert curve order based on their uv coordinates. Rays in the same grid are treated as the same and chosen in arbitrary order.

Specular rays present the biggest challenge when constructing packets. Specular rays are generated when rays intersect with a specular surface and generate either a reflection or refraction ray. As a result they can have an origin at any point on the surface of a specular object. While this means that there is no way to exploit a common origin point, we can still take advantage of a common origin space.

Two methods of specular ray packeting were tested. The first method splits all the rays into 3 sets based on the maximum component of the direction vector

which we will call *global division*. This is essentially randomly selecting rays to be in a packet. Splitting rays into 3 sets just makes it easier to calculate the packet frustum. The second method is what we call *queued rays*, where rays are split based on which BVH node the origin resides in. *Queued rays* take advantage of the shared origin space.

### 3.4.4 Global Division

To form a packet based on global division we take rays from one of the 3 sets. In the same way that the shadow ray frustum is formed, we intersect the rays with the plane perpendicular to the maximum component direction. However, in this case we need to use two planes since the specular rays do not share a common origin. The two planes selected are the bounding planes of the root BVH node perpendicular to the max ray direction. Then we can use the max and min hit values on the far and near plane to create 4 rays for the specular frustum.

### 3.4.5 Queued Tracing

#### Queuing Rays

When forming ray packets for primary and shadow rays the shared common origin is exploited to efficiently construct highly coherent packets. Specular rays do not have this shared origin so it requires them to be sorted into coherent bins of rays. It has been stated previously that using a nearby origin approach coherent rays can be missed, however this does not reduce the quality of the ray packet formed. In fact, as the number of rays being sorted increases this becomes less of a problem as the ray packets would be filled regardless.

Using a search tree to select rays close in proximity is too inefficient due to the need of a search tree as previously stated. Binning using a uniformed grid can quickly partition the rays into packets, but can lead to poor quality packets depending on the distribution of geometry in the scene. However, given that

specular rays can only originate from a specular surface the search space for ray origins can be reduced.

Instead of using a uniform grid, BVH nodes are selected as the origin spaces for binning the rays. In Navrátil et al. [10] rays are queued at nodes in a k-d tree down the traversal. Their technique queued the rays during the traversal process at the nodes to preserve locality, we instead queue at the nodes based on the ray origin and traverse the BVH fully. The rays at each queue node are split into ray packets similarly to how shadow rays packets are formed, though some extra work must be done to ensure the frustum bounds properly enclose the rays.

Since BVH nodes enclose their children, not all BVH nodes need be a queue point. The set of nodes that are selected as queue points must contain all the scene geometry in order to completely enclose the space of ray origins. In addition, rays queued at the same node will be placed in the same packet. It is thus necessary that the nodes are selected such that rays will likely be evenly spread amongst the queues. For our implementation we chose queue nodes to be those furthest down the BVH tree that contain at least $n$ objects.

The value of $n$ determines the total number of queue nodes used in the scene. As the number of queue nodes increases so does the coherency of the rays in a packet, but it also decreases the total number of rays per packet. We can estimate the average packet size with $q$, the number of queue nodes, and $r$, the number of rays being queued:

**Proposition 3.4.1.** *Average Packet Size* $= \frac{r}{q}$

It is desirable that the average packet size be higher while keeping the coherency high. As the number of packets increases then more frustum culling operations are done relative to the number of rays operations. There is a balance to be struck between the coherency of a packet and the total number of packets that must be traced. While the average packet size increases as the number of queue nodes decreases, it is limited by the maximum packet size. The necessity

of limiting the max packet size is due to the distribution of ray packet sizes. If there was no limiting of ray packet size then there could be a few extremely large packets which may also cause low coherency. Ray packet sizes is further discussed in Section 4.3.

**Frustum Calculation**

Calculating the frustum for rays queued at a node is not as simple for as primary or shadow rays due to the lack of a common origin. To find the frustum, we need to find 4 rays which create 4 bounding planes that enclose the rays. Two different methods of determining bounding rays were considered, with each requiring the rays to be selected slightly differently.

1. MAX heuristic based on the direction vector component with the greatest magnitude.
2. CORNER heuristic based on the sign of the 3 components of the direction vector.

Using the MAX heuristic, rays within a queue node are divided into 6 sets of rays with each set of rays being ones whose direction vector share the same maximum component and sign. Then, the far and near bounding planes for the frustum are selected. The far plane being the axis aligned plane of the root BVH node perpendicular to the maximum ray component and is in the direction the rays are pointed towards. The near plane is again the axis aligned plane perpendicular to the maximum ray component, however it being the plane from the queue node that is away from the ray direction. Then the rays are intersected with the two planes to find the maximum and minimum intersection points in the two planes in the two other components, which will be $minu_{far}$, $maxu_{far}$, $minv_{far}$, $maxv_{far}$, $minu_{near}$, $maxu_{near}$, $minv_{near}$, and $maxv_{near}$. The four bounding rays would then be formed using these points. For example, if the $x$ component was the maximum direction then one of the rays would be one formed by the two points

$(near, minu_{near}, minv_{near})$ and $(far, minu_{far}, minv_{far})$, with the origin at the first point. This is illustrated in 2D form in Figure 3.1, but can be extended to 3D.

The CORNER heuristic, on the other hand is split into 8 different sets as there are 8 combinations of direction vector sign components. Here, the corner of the bounding box for the queue node opposite of the direction the rays are pointed is selected as the origin of the frustum. To find the direction vectors for the 4 bounding rays requires a similar technique as the MAX heuristic where the packet rays are intersected with the root BVH node bounds. We select the bounding plane as the one perpendicular to the axis which has the greatest summed value over all the rays in the packet. In addition to the packet ray intersection, the intersection of the ray which is formed by the frustum origin and the packet ray origin is also calculated. This is done so that the frustum properly encloses the packet rays and is shown using a 2D representation in Figure 3.2. To truly find minimal frustum bounding rays would actually require finding the packet ray intersections on the root BVH bounds for all 3 axis and use only the first intersection. However, this requires 3 times as much work so we opted to use only a single axis to reduce the frustum bound calculations at the cost of having a less optimal frustum.

### Ray Reordering

Often queues will contain more rays than the max packet size even after splitting into the requisite number of sets for the MAX or CORNER heuristics. Rather than randomly selecting the rays for a packet, they are first partially sorted, then selected in order.

The rays are inserted into a uniform grid based on their intersection point with the root BVH node bounding box as described in Section 3.4.5. Then rays are selected from the uniform grid in a Hilbert curve order, for the same reason that primary rays were selected in that manner, and is shown in Figure 3.3 as an example. This does not guarantee that an optimal minimal frustum will be

created, due to how the frustums need to be generated.

For example, in the CORNER heuristic the ray origin plays a part in forming the frustum bounds. Looking at Figure 3.2 in the 2D case, if the ray were to point mainly in the vertical direction and the ray formed by the origin pointing mainly in the horizontal direction then a large frustum would be created. In this case, it may be suitable for rays that also create large frustums to be grouped together since they inherently can not produce a tight frustum. Subsequently rays that produce smaller frustums and are closer can be placed in the same packet. However, finding such rays combinations is difficult and time consuming, so our method gives a greedy estimate of optimal ray compositions for minimal frustums.

# Results

The machine we tested on was a Macbook Pro with Mac OSX 10.9.2 installed. The processor was a 2.6 GHz Intel Core i7 with 4 cores. Our work does not take advantage of the multiple cores, so performance was limited to a single core with a 32KB L1 cache, 256KB L2 and a 6MB shared L3 cache.

All images were rendered using a 512x512 resolution. The number of specular reflections and refractions were limited to a max of 10 times per pixel.

## 4.1 Test Scenes

Figure 4.1: Rendering of the dragon scene

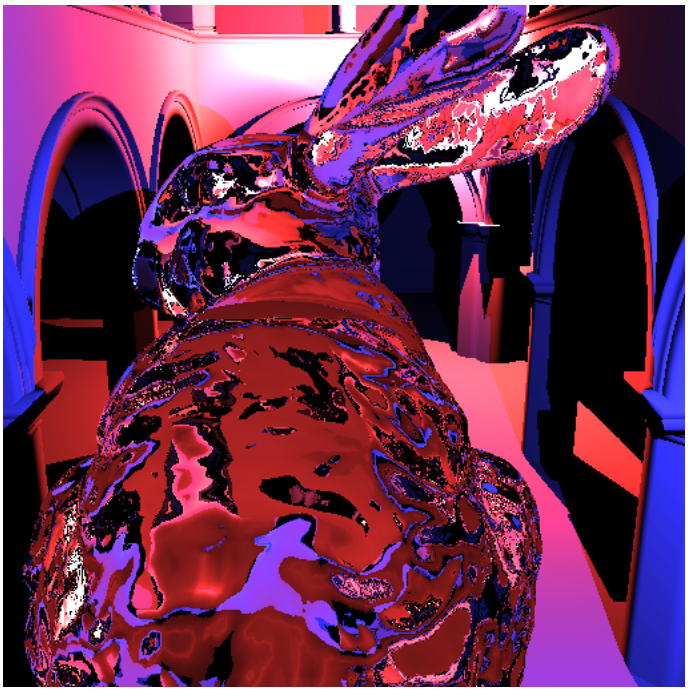


Figure 4.2: Rendering of the sponza+bunny scene

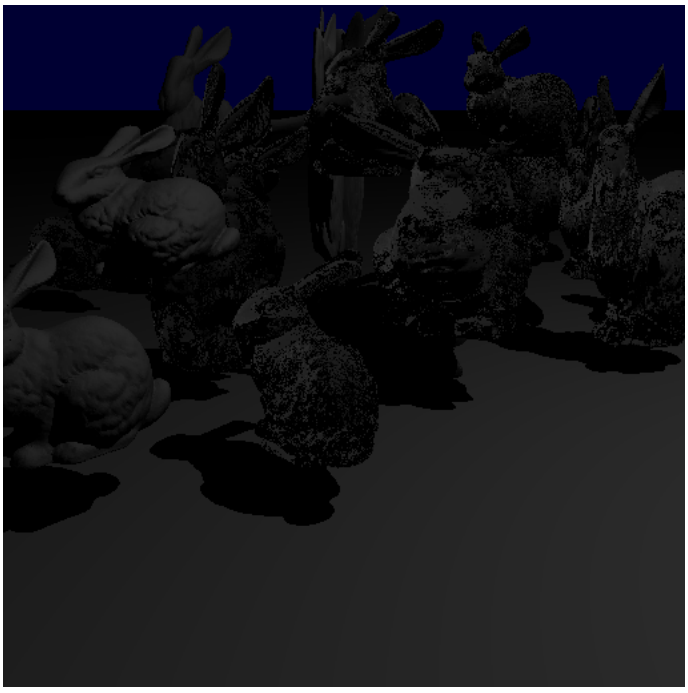


Figure 4.3: Rendering of the 20 bunny scene

A variety of test scenes were used to demonstrate the effects of varying ray composition. This ranged from fully diffuse to fully specular scenes. We also tested scenes with a combination of specular and diffuse objects to show the results of a more typical scene.

Test scene one contains only a diffuse Stanford dragon model [2] as shown in Figure 4.1. In addition to being a fully diffuse scene it, is also the scene containing the lowest number of triangles. Here only primary and shadow rays are traced, hence *queue tracing* has no effect as no specular rays are traced. Due to the manner in which primary and shadow packets are constructed, it was expected that the SIMD utilization be fairly high. The fully diffuse scene was meant to be a test of the ideal peak performance.

On the opposite end of the spectrum was a scene with the hairball model [8], which is completely composed of specular triangles. In theory this should be the worst case scenario in terms of ray coherency. However, it is also the case that the hairball model has an fairly even distribution of triangles over the scene. As a result ray queues will be well distributed over the scene with similar sizes.

Then there is the test scene with the Stanford bunny model [1] inserted into the middle of the sponza model [6] as shown in Figure 4.2. The sponza model is diffuse while the bunny model is specular. While both diffuse and specular rays are traced the scene can still be considered "easy", as all the specular triangles are concentrated at the bunny model. This leads to many specular rays gathered around a smaller volume, which allows for better ray packet construction.

The last scene is a set of 20 bunny models distributed over a triangle as shown in Figure 4.3. Half of the models are diffuse while the other half are specular. While not as specular dense as the hairball scene, this scene presents another set of difficulties. With the specular objects are distributed over the scene, rather than being concentrated or in regular pattern, ray queues are not as effective in capturing groups of specular rays. This would likely result in a higher number of smaller ray packets being formed.

Table 4.1: Scene Composition

| Scene | # Triangles | Ray Types |
|---|---|---|
| Dragon | 100K | Primary + Shadow |
| Sponza + Bunny | 136K | Primary + Shadow + Specular |
| 20 Bunnies | 1.39M | Primary + Shadow + Specular |
| Hairball | 2.88M | Primary + Specular |

Table 4.2: SIMD Utilization Degradation

Total SIMD utilization from 0 to 1, with 1 being full SIMD utilization

| Scene | No Queue | CORNER queue | MAX queue |
|---|---|---|---|
| Dragon | 0.8396 | 0.8396 | 0.8396 |
| Sponza + Bunny | 0.7897 | 0.8021 | 0.8013 |
| 20 Bunnies | 0.6301 | 0.6904 | 0.6751 |
| Hairball | 0.5752 | 0.6092 | 0.6269 |

## 4.2 SIMD Utilization

### 4.2.1 Metrics

The first measure of the effectiveness of our technique is how well SIMD registers are utilized. While we do not expect the worst case scenario of $\frac{1}{n}$ that Wald et al. [18] predicted, getting close to that is still not desirable. In fact, we want to see a $2:1$ ratio of SIMD utilization between a 4 wide and 8 wide instruction implementation. A $2:1$ ratio would mean that the same number of intersections per operation ratio. Even in this case the 8 wide implementation would have a relatively poorer performance because of the extra overheads required for wider register operations.

As stated previously, we can expect to have relatively high SIMD utilization in scenes with few specular surfaces. In contrast highly specular scenes will see a decrease in ray coherency as the number of times rays bounce increase. With a decrease in coherency there is corresponding drop in SIMD utilization. In our case we would define SIMD usage for an individual intersection operation as:

**Definition 4.2.1.** Intersection SIMD Utilization $= \frac{\#\text{of live rays}}{\text{SIMD width}}$

Table 4.3: Ray Composition

All scenes are rendered with a max of 10 specular bounces.

| Scene | Primary | Specular | Shadow | Total Rays |
|---|---|---|---|---|
| Dragon | 72.64% | 0% | 27.36% | 361K |
| Sponza + Bunny | 23.31% | 30.71% | 45.98% | 1.12M |
| 20 Bunnies | 37.74% | 31.92% | 30.34% | 694K |
| Hairball | 9.84% | 90.16% | 0% | 2.66M |

Now we can extend this definition of SIMD utilization to the intersection of a ray packet with the scene geometry. To do this we sum up the the SIMD utilization of all the ray intersections required to traverse the BVH. More formally:

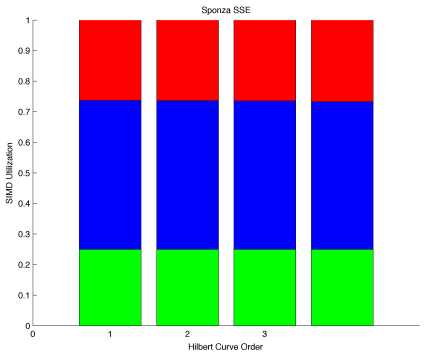
**Definition 4.2.2.** Raypacket SIMD Utilization $= \frac{\sum_r^n numLiveRays(r)}{sizeof(R) \times \text{SIMD width}}$

where, $R = (r_1, r_2, .., r_n)$ where $r_i$ is the ith SIMD ray intersection done

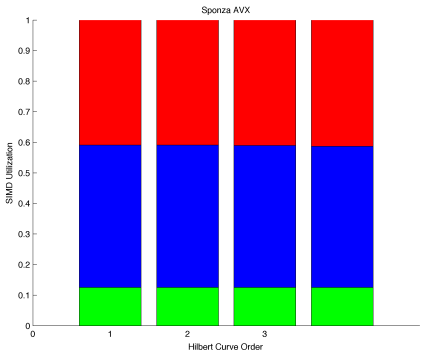
In the case of AVX packet tracing the SIMD width is 8. For each SIMD ray intersection the number of live rays must be at least 1 otherwise the intersection would have been filtered out at an earlier stage of the traversal. This means that in the worst case using AVX packet tracing, there would a $\frac{1}{8}$ usage, which is the same SIMD utilization as described by Wald et al. In these cases the AVX tracing should have much worse performance than normal *partition traversal*. This is the result of combined overheads in loading and setting up the rays and the increased loading of the unnecessary data.

### 4.2.2 SIMD Loss

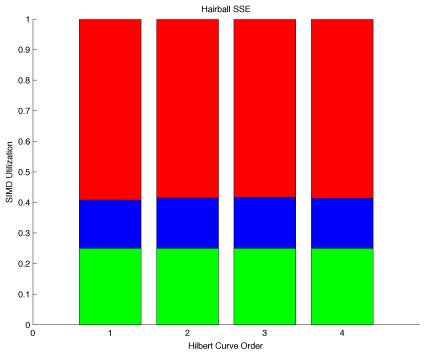
Since the dragon scene contains no specular surfaces it is unaffected by ray queuing. As seen in Figure 4.5, the SIMD utilization of the dragon image is 0.79 and 0.67 for SSE and AVX respectively regardless of any queuing methods used. However, we can not say that the SIMD utilization in the dragon scene is completely representative of any fully diffuse scene. Consider a trivial scene with only a single triangle, which covers the entire image space. Such a scene would have a SIMD utilization of 1.



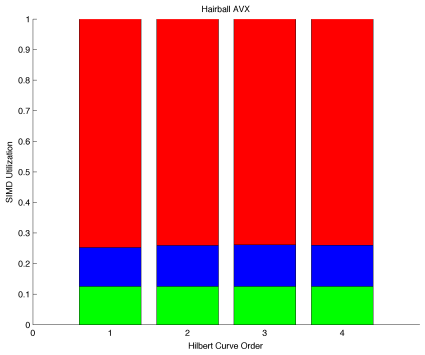

(a) Hilbert Curve Order vs SIMD Utilization of SSE on sponza using CORNER heuristic




(b) Hilbert Curve Order vs SIMD Utilization of AVX on sponza using CORNER heuristic



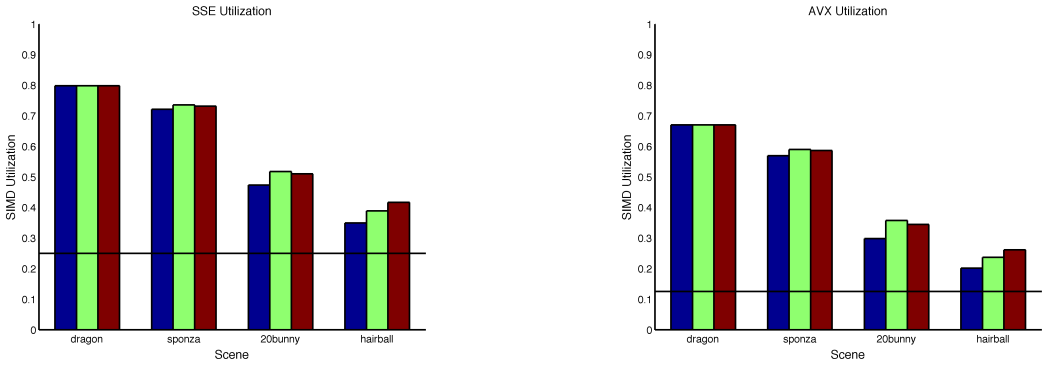

(c) Hilbert Curve Order vs SIMD Utilization of SSE on hairball using MAX heuristic




(d) Hilbert Curve Order vs SIMD Utilization of AVX on hairball using MAX heuristic

Figure 4.4: Effect of Hilbert curve order on SIMD utilization. The red shows the maximum SIMD utilization of 1. The blue shows the SIMD utilization using queues. The green shows the minimal possible SIMD utilization of $\frac{1}{\text{SIMD width}}$

It is clear that scenes, such as 20 bunny and hairball, which create more specular rays have lower SIMD utilization, but also important to note is the difference between the SSE and AVX implementations. Table 4.2 shows the ratio between the AVX SIMD utilization and SSE SIMD utilization. As can be seen in the dragon scene, which has only primary and shadow rays, there is relatively less loss of SIMD utilization as the SIMD width increases. As scenes contain more specular rays the loss of SIMD utilization becomes more substantial. In the case of the hairball scene where specular rays are the majority of the rays, it reaches




(a) SIMD Utilization using SSE

(b) SIMD Utilization using AVX

Figure 4.5: SIMD utilization on different scenes using packet size of 1024, Hilbert curve order 4, and queue size of 1024. Blue bar - no queues. Green bar - CORNER queue. Red bar - MAX queue.

close to 0.5 SIMD utilization, meaning it is likely that the performance of the AVX implementation would be worse that SSE. On a positive there is a decrease in the loss of SIMD utilization when using queues of any type.

### 4.2.3 Queue Size

A perfect SIMD utilization of 1 is the limit that we desire to achieve, but is also impossible to achieve. Figure 4.6 shows the SIMD utilization of the various scenes as we vary the size of the queues. The y-axis represents the SIMD utilization and the x-axis is the number of primitives within the queue node. As the number of objects within the queue increases the SIMD utilization decreases, since there are more surfaces for the rays to hit within the queue. The decrease follows close to a logarithmic curve, with the limit being when the entire scene is encapsulated in a single queue node.

It would appear that reducing the number of primitives within the queue would be the most optimal. However, this is not the case. Reducing the number of objects within the queue also increases the total number of queues. This increases the amount of time and space required to store and iterate through the queues

when forming ray packets. In addition, this also reduces the average number of rays per queue as mentioned in Section 3.4.5. The consequence of decreased average number of rays per queue is discussed in the Section 4.3.

In Figure 4.6 we compare the relative drop-off in SIMD utilization for the different scenes. The drop-off rate helps give intuition as to how much the queues affect the coherency of packets.

The sponza scene has relatively little drop-off compared to the bunnies and hairball scenes. The assumption would be made that this is the result of the sponza scene having fewer specular rays. However, this is not the case looking at Table 4.3 where we see that the sponza and the 20 bunnies scenes have about the same percentage of specular rays. Rather than the number of rays, it has more to do with where the specular surfaces are. The sponza scene has a single specular bunny meaning that the all the specular rays are concentrated in a small volume. Contrast this with the 20 bunnies scene where the specular bunnies are distributed over the scene. Meaning that specular rays would be queued all over the scene rather than a few select queue nodes.

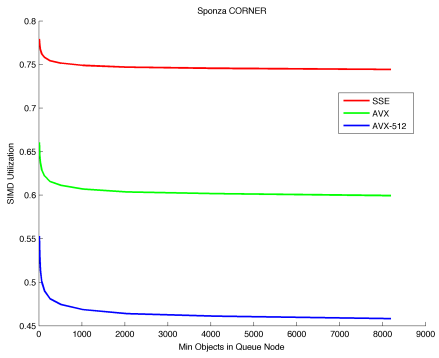
This reveals a weakness with the currently applied scheme for determining queue nodes. The BVH creation process ignores the reflectance properties of the objects and as a result queue nodes contain both specular and diffuse objects. So while two scenes can both use the same number of objects in a queue node, the number of queue nodes that actually contain specular objects and the actual number of specular objects contained in a queue node will differ.

The problem of queues containing both specular and diffuse objects explains why the hairball scene while containing many more specular rays than the 20 bunnies scenes does not degrade in SIMD utilization that much more. As such it is rather likely that the distribution of specular objects, rather than number of specular rays, that has a more significant effect the SIMD utilization. A larger number of specular rays obviously decreases SIMD utilization due to having a

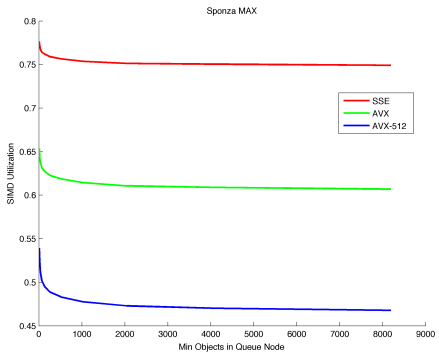
larger number of incoherent rays. More damaging is when these rays are generated further apart leading them to be even more incoherent.

We can not solve the problem of having more incoherent rays, because it lies within the scene geometry. But we can instead be wiser in our formation of ray queues such as ignoring diffuse objects. Unfortunately, we may have to admit that certain scenes may not lend themselves to easy ray queuing.

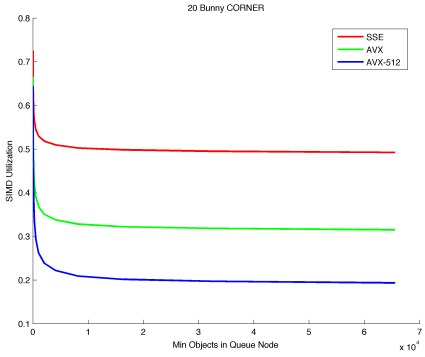
## 4.3 Packet Sizes



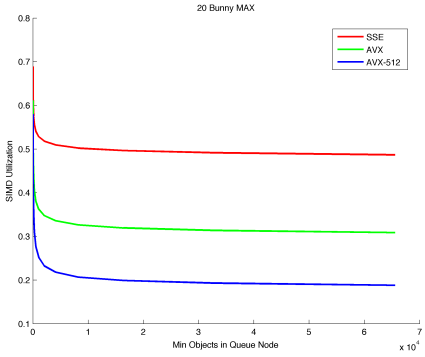

(a) Sponza using CORNER heuristic



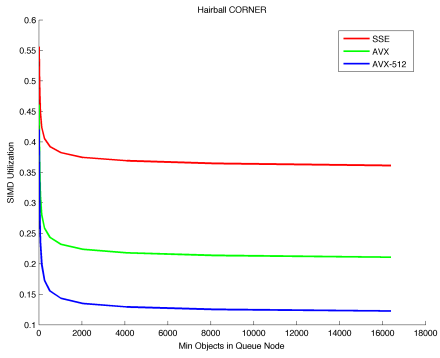

(b) Sponza using MAX heuristic



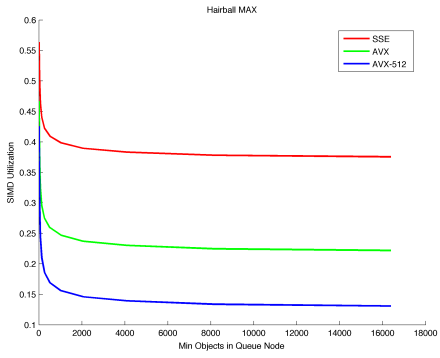

(c) 20 bunny using CORNER heuristic




(d) 20 bunny using MAX heuristic




(e) Hairball using CORNER heuristic




(f) Hairball using MAX heuristic

Figure 4.6: Effect of minimum number of objects in queue on SIMD utilization. There is a logorithmic decrease in SIMD utilization as the minimum number of objects in a queue is increased.

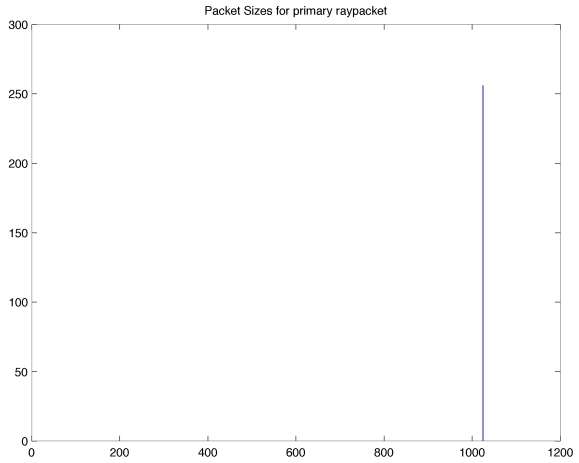



Figure 4.7: Histogram of number of rays in packets for primary rays in sponza+bunny scene

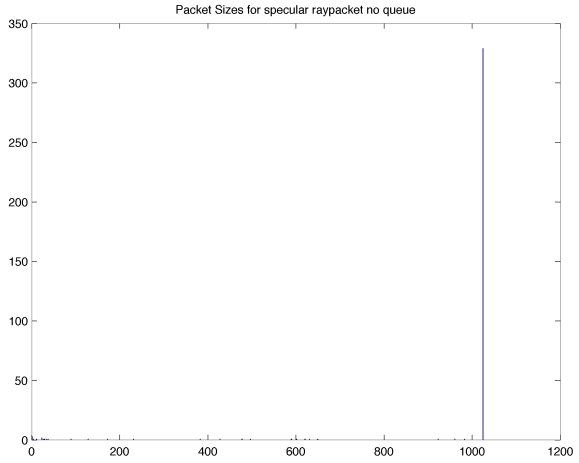



Figure 4.8: Histogram of number of rays in packets for specular rays without queuing in sponza+bunny scene

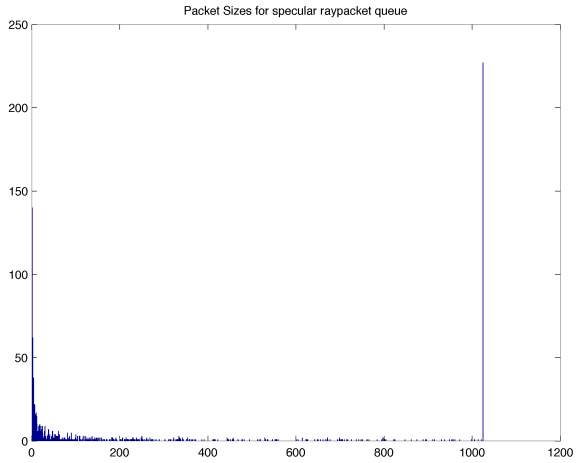



Figure 4.9: Histogram of number of rays in packets for specular rays with queued tracing using CORNER heuristic in sponza+bunny scene

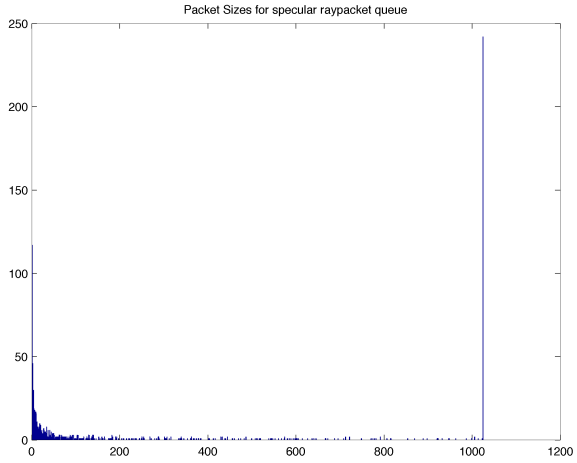



Figure 4.10: Histogram of number of rays in packets for specular rays with queued tracing using MAX heuristic in sponza+bunny

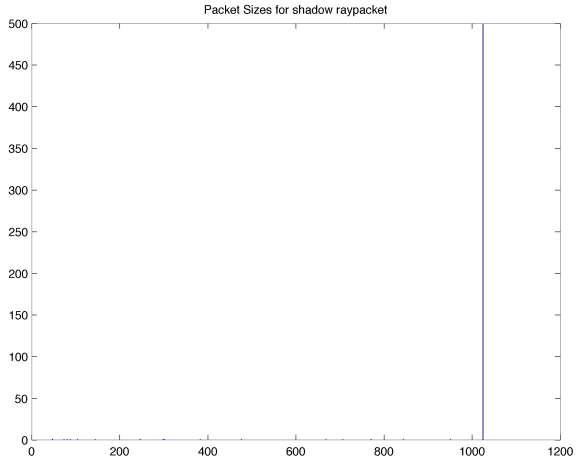
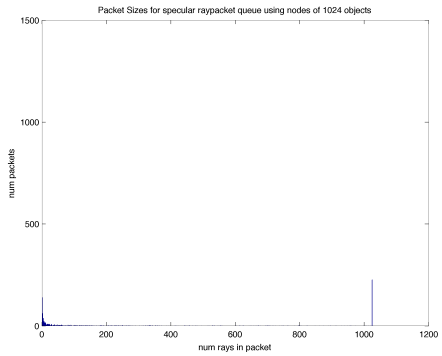
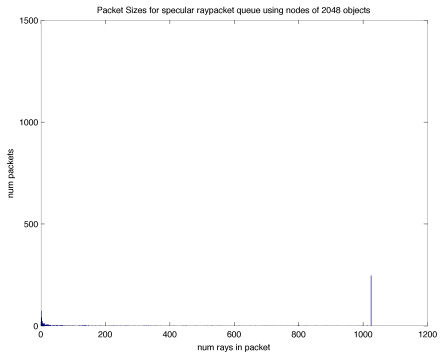


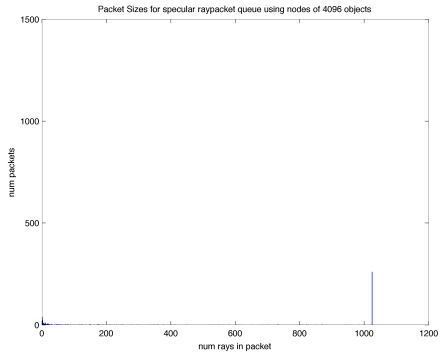

Figure 4.11: Histogram of number of rays in packets for shadow rays in sponza+bunny scene



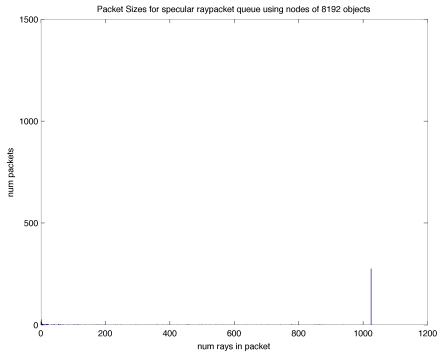

(a) Queues of 1024 for sponza+bunny scene




(b) Queues of 2048 for sponza+bunny scene




(c) Queues of 4096 for sponza+bunny scene

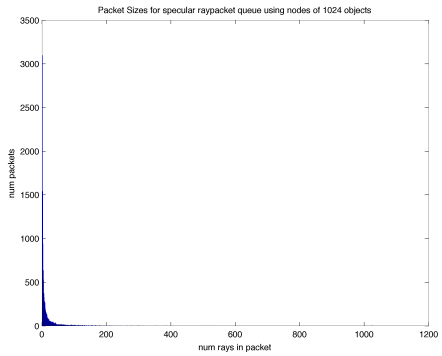



(d) Queues of 8192 for sponza+bunny scene

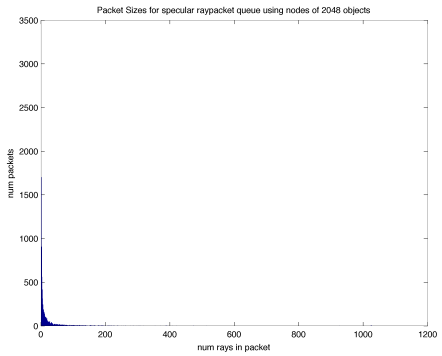
Figure 4.12: Histogram of packets having n rays for sponza+bunny scene using CORNER heuristic

Partition traversal is suited to large packets sizes, around 1024 rays. In order to have efficient traversal we need the frustum as close to filled as possible. As the number of rays decreases in the packet, the cost of frustum operations becomes relatively more expensive.

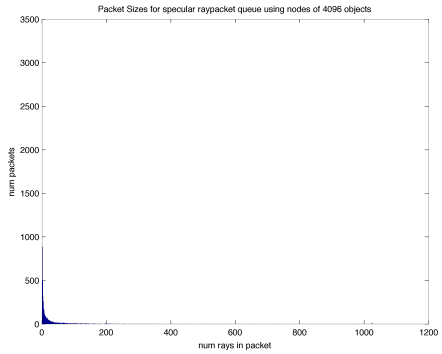
Looking at Figure 4.7 and Figure 4.11, we see packets for both primary and shadow rays are mostly filled due to how they are constructed. In fact, for primary rays as long as the number of rays per packet evenly divides into the number of pixels traced then all packets will be completely filled. The number of partially filled shadow packets is dependent on the number of lights. If the number of lights is $l$, then at max there are $8l$ partially filled shadow packets, since shadow rays



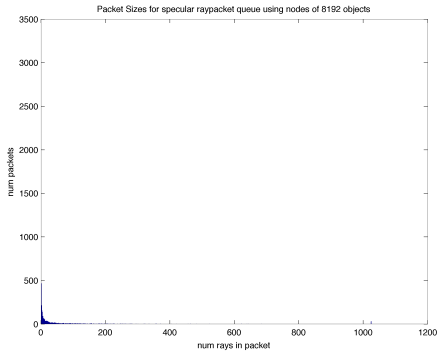

(a) Queues of 1024 for 20 bunny scene




(b) Queues of 2048 for 20 bunny scene




(c) Queues of 4096 for 20 bunny scene

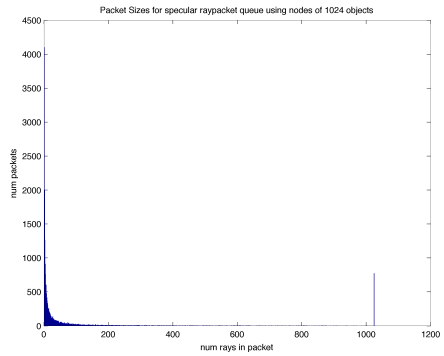



(d) Queues of 8192 for 20 bunny scene

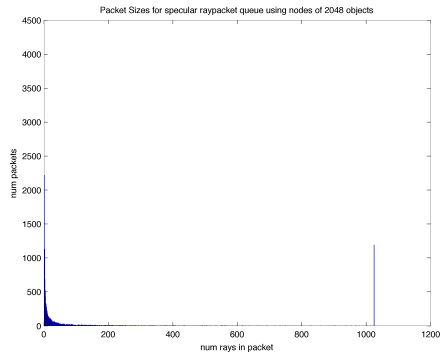
Figure 4.13: Histogram of packets having n rays for 20 bunny scene using CORNER heuristic

are split into 8 directions.

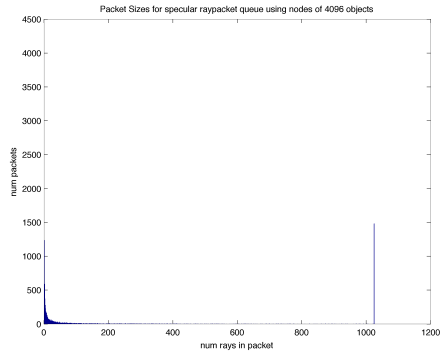
It can be seen in Figure 4.8 that when we fill the ray packets without queuing the rays then almost all packets are fully filled, since all ray packets can and will be filled except the last packet due to insufficient rays. Comparatively in Figure 4.9 and Figure 4.10, it can be seen that when queued packet sizes can vary significantly with many small sized packets. In our implementation, each packet contains only rays at the $i$th bounce. This means there will be at most $n$ partially filled packets without queuing, where $n$ is the max number of specular bounces. This is compared to queuing where each queue can have a partially filled packet, leading to up to $n \times q$ partially filled packets, where $q$ is the number of queues. This is especially troublesome since it leads to a high concentration of relatively small



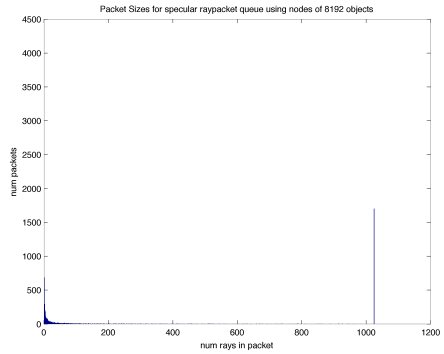

(a) Queues of 1024 for hairball scene




(b) Queues of 2048 for hairball scene




(c) Queues of 4096 for hairball scene



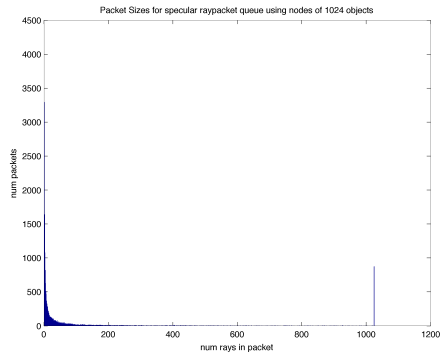

(d) Queues of 8192 for hairball scene

Figure 4.14: Histogram of packets having n rays for hairball scene using CORNER heuristic

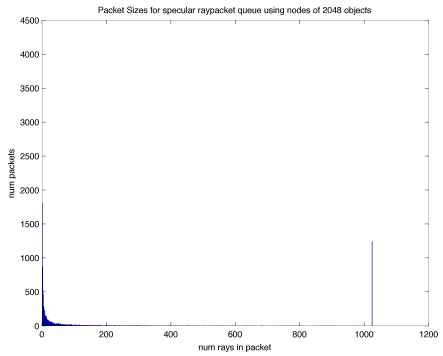
packets. As a result, there is a balance that needs to be struck between better SIMD utilization and larger packet size when selecting the number of queues.

### 4.3.1 Effects of Queue on Packet Size

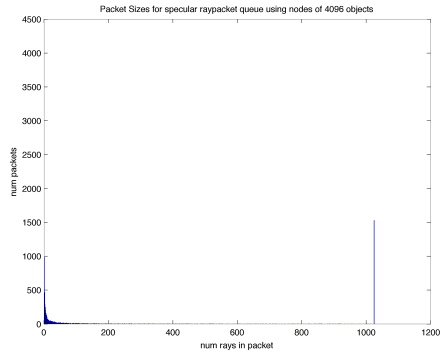
When there is only a single queue for the entire scene, the average packet size of queuing and non-queuing matches exactly. As we continue to increase the number of queues, or restrict the number of objects in a queue, there is a decrease in average packet size. However, this does not imply that with only one object per queue the packet size is reduced to 1 ray per queue. Rather, it is dependent on the structure of the scene as can be seen in Figure 4.16, 4.17, and 4.18. While all three graphs show a similar curvature, they start at different points.



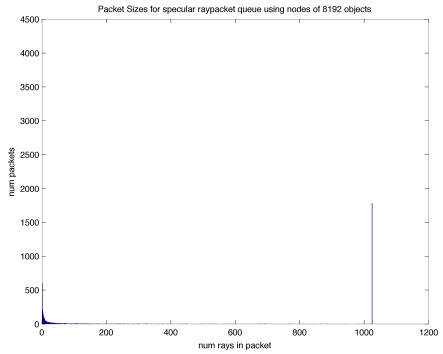

(a) Queues of 1024 for hairball scene




(b) Queues of 2048 for hairball scene




(c) Queues of 4096 for hairball scene




(d) Queues of 8192 for hairball scene

Figure 4.15: Histogram of packets having n rays for hairball scene using MAX heuristic

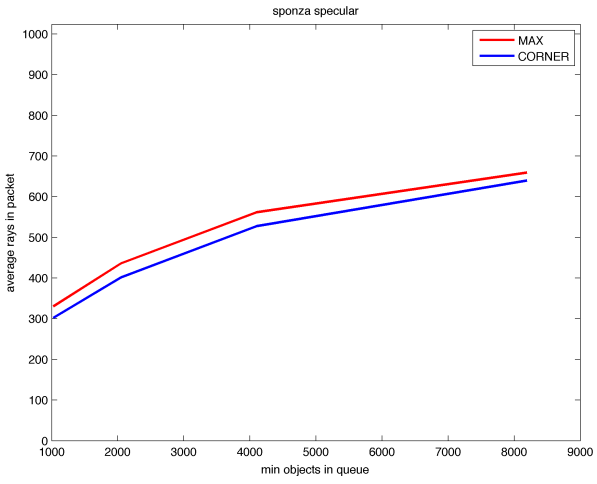



Figure 4.16: Min objects in queue vs average packet size in sponza

It is not surprising that the sponza scene has the largest average packet size as only the bunny model in the scene is specular. Two reasons likely contribute to

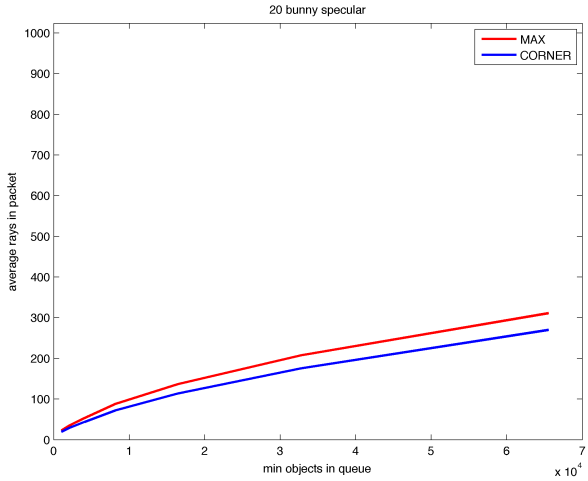



Figure 4.17: Min objects in queue vs average packet size in 20 bunny

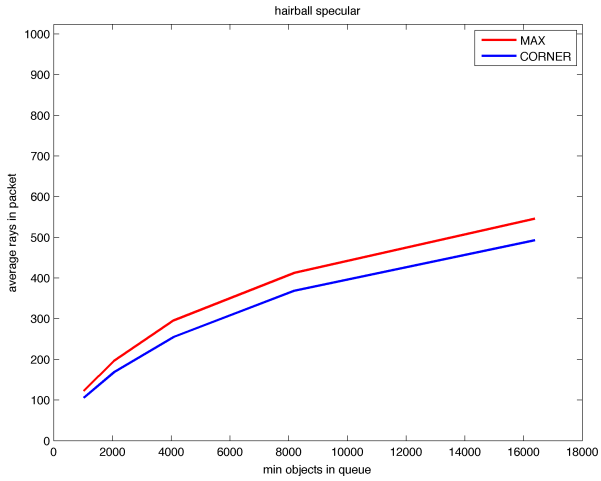



Figure 4.18: Min objects in queue vs average packet size in hairball

the large packets. First, the relatively simpler specular geometry generates fewer specular reflections. Hence, specular rays are concentrated in the earlier bounces where coherency is high. Secondly, queues nodes would tend to have less diversity of specular and diffuse surfaces leading to specular rays concentrated among fewer queues. Looking at Figure 4.12, it is clear that there is still a decent number of packets that are fully filled.

The problem of specular surfaces spread over the scene manifests itself in the 20 bunny scene. Since half the bunnies are specular and the other half diffuse, queues are more likely to have a split of specular and diffuse surfaces, meaning

queues are less efficient at capturing specular rays together. In fact, even though the hairball scene has many more specular surfaces and specular rays, the queues are able to better group rays into larger packets. We see this in Figures 4.13 and Figure 4.14 where the 20 bunnies scene almost has no packets filled up to the maximum of 1024 rays, while the hairball scene does have many. Though we must take into account the results shown in Figure 4.17 where it takes having many more objects in the ray queues for the 20 bunnies scene to have comparable average ray packet sizes.

## 4.4 Performance

Table 4.4: Singe Ray Rendering Times

| Scene | Time (seconds) |
|---|---|
| Dragon | 0.35 |
| Sponza + Bunny | 2.29 |
| 20 Bunnies | 1.11 |
| Hairball | 17.73 |

Table 4.5: SIMD Rendering Times

Best Rendering Times (seconds). Size column lists the size of queue used to achieve listed rendering time.

| Scene | Single | Size | SSE | Size | AVX | Size |
|---|---|---|---|---|---|---|
| Dragon | 0.22 | n/a | 0.14 | n/a | 0.12 | n/a |
| Sponza + Bunny | 1.51 | n/a | 0.85 | n/a | 0.78 | n/a |
| Sponza + Bunny MAX | 1.47 | 1024 | 0.83 | 1024 | 0.75 | 1024 |
| Sponza + Bunny CORNER | 1.5 | 4096 | 0.86 | 4096 | 0.80 | 4096 |
| 20 Bunnies | 1.22 | n/a | 1.01 | n/a | 1.01 | n/a |
| 20 Bunnies MAX | 1.13 | 16384 | 0.93 | 32768 | 0.93 | 32768 |
| 20 Bunnies CORNER | 1.30 | 32768 | 1.10 | 32768 | 1.11 | 32768 |
| Hairball | 20.20 | n/a | 16.77 | n/a | 17.56 | n/a |
| Hairball MAX | 17.63 | 4096 | 13.96 | 4096 | 14.19 | 4096 |
| Hairball CORNER | 21.97 | 2048 | 18.26 | 2048 | 18.47 | 2048 |

Regardless of SIMD utilization or use of ray packetsi, the speed at which images can be rendered is of most importance. If the overheads associated with queuing

Table 4.6: SIMD Speedup

| Speedup | | | | |
|---|---|---|---|---|
| Scene | S v SSE | P v SSE | S v AVX | P v AVX |
| Dragon | 2.5 | 1.57 | 2.92 | 1.83 |
| Sponza + Bunny | 2.69 | 1.77 | 2.94 | 1.94 |
| Sponza + Bunny MAX | 2.76 | 1.82 | 2.94 | 1.88 |
| Sponza + Bunny CORNER | 2.66 | 1.74 | 2.86 | 1.88 |
| 20 Bunnies | 1.1 | 1.21 | 1.1 | 1.21 |
| 20 Bunnies MAX | 1.2 | 1.22 | 1.2 | 1.22 |
| 20 Bunnies CORNER | 1.01 | 1.18 | 1 | 1.17 |
| Hairball | 1.06 | 1.2 | 1.01 | 1.15 |
| Hairball MAX | 1.27 | 1.26 | 1.25 | 1.24 |
| Hairball CORNER | 0.97 | 1.2 | 0.96 | 1.19 |

the ray packets outweigh the performance gains, then there is no reason for ray queuing. It is expected that scenes with only primary and shadow rays would have greatest benefit with regards to SIMD utilization and raypacketing due to the inherent coherency.

As a baseline, the dragon scene using SSE and AVX have a speedup of 2.5x and 2.92x respectively. This is the standard that we compare the other scenes to, as it does not contain specular surfaces and thus no ray queuing. It would seem intuitive that, as the number of specular surfaces increased there would be corresponding drop in speedup, since this meant that the percentage of specular rays vs other rays would increase. According to Figure 4.3, this would mean that the highest speedup would be in the dragon scene, then sponza + bunny, then 20 bunnies, and finally hairball. This was found to not be the case.

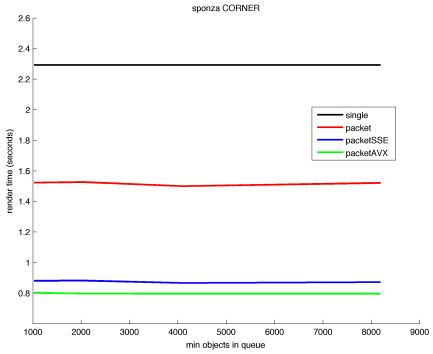
As mentioned previously, the selection of minimum objects encompassed by a queue is a tradeoff between good coherence of rays and sufficient rays for efficient packet traversal. Figure 4.19 shows the rendering times of different scenes with regards to the minimum number of objects in a queue. It is not clear how to select queue sizes that result in optimal rendering speed.

With the sponza + bunny scene, the optimal speedup was 2.75x and 3.05x respectively for SSE and AVX. It was a slight surprise that this scene had a better

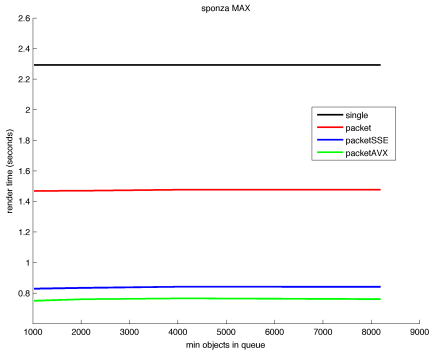
speedup than the dragon scene since the sponza + bunny scene contains specular rays, while the dragon scene does not. In addition, the sponza + bunny scene was illuminated by 2 lights rather than the 1 light in the dragon scene. Meaning the sponza + bunny scene had a larger percentage of shadow rays. However, it is not unexpected that certain scene setups will result in better SIMD speedup, considering speedup is a result of both SIMD optimization and ray packeting. For this particular setup, as evidenced by Figure 4.5, it is not so much SIMD utilization, but rather good ray packeting that results in the better speedups.

More interesting to compare are the 20 bunnies scene and the hairball scene. The 20 bunnies scene had 1.1x speedup for both SSE and AVX, while the hairball scene had 1.27x and 1.25x speedup for SSE and AVX respectively. Here we see barely any speedup compared to the single ray version. In fact, it would seem that attempting to use SIMD for speeding up tracing is not worth the effort here. Even more unfortunate is the drop off in speedup going from SSE to AVX in the hairball scene.

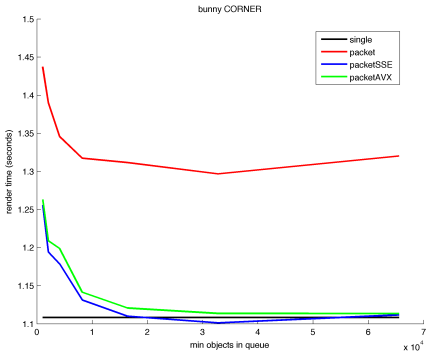
The reason that the hairball scene has a higher speedup can be seen in Figure 4.17 and Figure 4.18. The hairball scene has a much higher average packet size than the 20 bunnies scene, meaning while SIMD usage was higher in the 20 bunnies scene it was operations not SIMD optimized that had a more significant impact. The potential reason for this is that in this scene half the bunnies are specular while the other half are diffuse. What happens as a result is the specular surfaces are more dispersed in the scene, and the queues are not as well utilized since our method of creating queues does not take into account the object material. Secondly, the dispersal of specular surfaces means specular rays tend to be created further apart relative to objects in the scene. This relative distance is a factor in determining ray coherency in the scene.



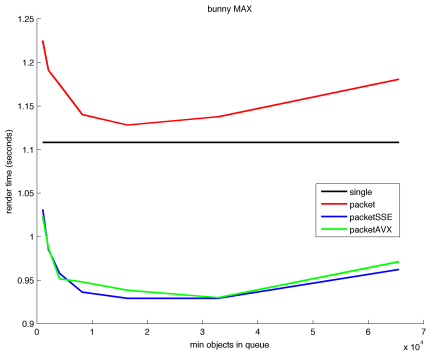

(a) Sponza using CORNER heuristic



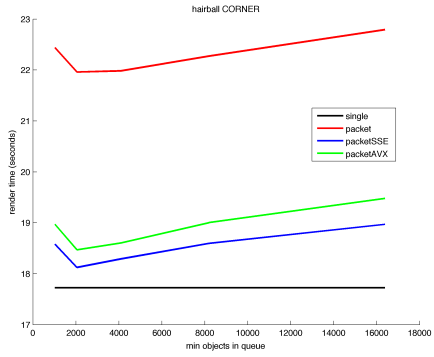

(b) Sponza using MAX heuristic



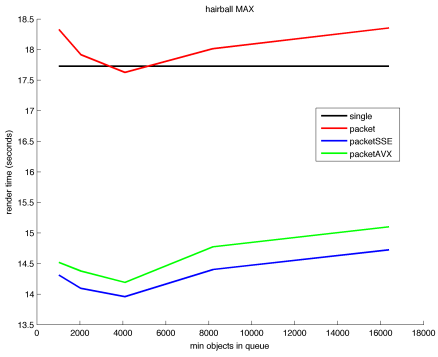

(c) 20 bunny using CORNER heuristic




(d) 20 bunny using MAX heuristic




(e) Hairball using CORNER heuristic




(f) Hairball using MAX heuristic

Figure 4.19: Effect of minimum number of objects in queue on rendering times

# Discussion and Future Work

Our work has shown that it is possible to increase the full SIMD utilization by carefully organizing the ray packets. However, it comes at the cost of increasing the number of packets that must be traced particularly small sized packets. In order to mitigate this problem a few possibilities come to mind.

One approach is to have more careful selection of queue nodes, by creating queue nodes based on specular objects rather than reusing the already built BVH structure for tracing rays. The weakness with the current implementation of queue nodes is that they contain both specular and diffuse surfaces. Inherently tracing rays using the BVH structure has no problems. However, many cases exist where an uneven distribution of specular rays is generated amongst the nodes.

It is noted though, even in cases where there are only specular nodes, such as the hairball, there are still many small packets. Hence, better queue nodes is not the only problem. It may be necessary to rely on hybrid techniques as suggested by other authors. First, use ray packet tracing for larger packets. Then switch to single ray tracing or other small packet techniques when the number of rays in a packet is below a threshold. In addition, rays should be queued such that they contain rays at any $n$th bounce, rather than at a specific bounce. Then only once a ray queue is sufficiently filled would we trace the rays in that queue.

As noted by others, considering the additional complexity required for collecting large ray packets, it is not necessarily the best choice for SIMD utilization at the moment. However, we have shown that even simply using the current BVH

structure we can improve the tracing speeds of SIMD ray packets. It is likely further examination can lead to notable speedups. More importantly, our work gives better understanding in how we can bundle coherent rays, which has implications not only in SIMD ray technique but also in single ray tracing.

# Appendices

# Code Listings

```
unsigned int
IntersectRayAABB8(const Vector3 &min, const Vector3 &max, const Rays &ray, float *tMin, float *tMax, int index)
{
  __m256 minx,miny,minz;
  __m256 maxx,maxy,maxz;
  __m256 ox,oy,oz;
  __m256 mint,maxt;
  __m256 xmin,xmax,ymin,ymax,zmin,zmax;
  __m256 results;
  unsigned int resultsMask;

  minx = _mm256_broadcast_ss(&min.x);
  miny = _mm256_broadcast_ss(&min.y);
  minz = _mm256_broadcast_ss(&min.z);
  maxx = _mm256_broadcast_ss(&max.x);
  maxy = _mm256_broadcast_ss(&max.y);
  maxz = _mm256_broadcast_ss(&max.z);

  ox = _mm256_load_ps(&ray.o->x[index]);
  oy = _mm256_load_ps(&ray.o->y[index]);
  oz = _mm256_load_ps(&ray.o->z[index]);

  __m256 invdx = _mm256_load_ps(&ray.dInv->x[index]);
  __m256 invdy = _mm256_load_ps(&ray.dInv->y[index]);
  __m256 invdz = _mm256_load_ps(&ray.dInv->z[index]);

  mint = _mm256_load_ps(tMin);
  maxt = _mm256_load_ps(&tMax[index]);

  __m256 tx1 = _mm256_mul_ps(_mm256_sub_ps(minx,ox),invdx);
  __m256 tx2 = _mm256_mul_ps(_mm256_sub_ps(maxx,ox),invdx);
  __m256 ty1 = _mm256_mul_ps(_mm256_sub_ps(miny,oy),invdy);
  __m256 ty2 = _mm256_mul_ps(_mm256_sub_ps(maxy,oy),invdy);
  __m256 tz1 = _mm256_mul_ps(_mm256_sub_ps(minz,oz),invdz);
  __m256 tz2 = _mm256_mul_ps(_mm256_sub_ps(maxz,oz),invdz);

  // do intersection tests
  xmin = _mm256_min_ps(tx1,tx2);
  xmax = _mm256_max_ps(tx1,tx2);
  ymin = _mm256_min_ps(ty1,ty2);
  ymax = _mm256_max_ps(ty1,ty2);
  zmin = _mm256_min_ps(tz1,tz2);
  zmax = _mm256_max_ps(tz1,tz2);

  __m256 tmax = _mm256_min_ps(xmax,_mm256_min_ps(ymax,zmax));
  results = _mm256_cmp_ps(tmax,mint,_CMP_GE_OS);

  __m256 tmin = _mm256_max_ps(xmin,_mm256_max_ps(ymin,zmin));
  results = _mm256_and_ps(results,_mm256_cmp_ps(tmin,maxt,_CMP_LE_OS));

  results = _mm256_and_ps(results,_mm256_cmp_ps(tmin,tmax,_CMP_LE_OS));

  resultsMask = _mm256_movemask_ps(results)&255;

  return resultsMask;
}

unsigned int
Triangle::intersect8(HitInfos &results, const Rays &rays, float *tMins, float *tMaxes, int index)
{
  __m256 mv0x,mv0y,mv0z;
  __m256 mv1x,mv1y,mv1z;
  __m256 mv2x,mv2y,mv2z;
  __m256 e1x,e1y,e1z;
  __m256 e2x,e2y,e2z;
  __m256 Px, Py, Pz;
  __m256 Tx, Ty, Tz;
  __m256 Qx, Qy, Qz;
  __m256 rox, roy, roz;
```

```
__m256 rdx, rdy, rdz;
__m256 det, inv_det;
__m256 u,v;
__m256 t;
__m256 ones,zeros;
__m256 hits;
unsigned int resultsMask = 0;

mv0x = _mm256_broadcast_ss(&m_vertices[0].x);
mv0y = _mm256_broadcast_ss(&m_vertices[0].y);
mv0z = _mm256_broadcast_ss(&m_vertices[0].z);
mv1x = _mm256_broadcast_ss(&m_vertices[1].x);
mv1y = _mm256_broadcast_ss(&m_vertices[1].y);
mv1z = _mm256_broadcast_ss(&m_vertices[1].z);
mv2x = _mm256_broadcast_ss(&m_vertices[2].x);
mv2y = _mm256_broadcast_ss(&m_vertices[2].y);
mv2z = _mm256_broadcast_ss(&m_vertices[2].z);

e1x = _mm256_sub_ps(mv1x,mv0x);
e1y = _mm256_sub_ps(mv1y,mv0y);
e1z = _mm256_sub_ps(mv1z,mv0z);

e2x = _mm256_sub_ps(mv2x,mv0x);
e2y = _mm256_sub_ps(mv2y,mv0y);
e2z = _mm256_sub_ps(mv2z,mv0z);

rox = _mm256_load_ps(&rays.o->x[index]);
roy = _mm256_load_ps(&rays.o->y[index]);
roz = _mm256_load_ps(&rays.o->z[index]);
rdx = _mm256_load_ps(&rays.d->x[index]);
rdy = _mm256_load_ps(&rays.d->y[index]);
rdz = _mm256_load_ps(&rays.d->z[index]);

Px = cross8x(rdy,rdz,e2y,e2z);
Py = cross8y(rdx,rdz,e2x,e2z);
Pz = cross8z(rdx,rdy,e2x,e2y);

det = dot8(e1x,e1y,e1z,Px,Py,Pz);

float epsilon = DET_EPSILON;
float negEpsilon = -DET_EPSILON;
hits = _mm256_cmp_ps(det,_mm256_broadcast_ss(&negEpsilon),_CMP_LE_OS);
hits = _mm256_or_ps(hits,_mm256_cmp_ps(det,_mm256_broadcast_ss(&epsilon),_CMP_GE_OS));

resultsMask = _mm256_movemask_ps(hits)&255;
if (!resultsMask)
  return resultsMask;

float one = 1.0f;
ones = _mm256_broadcast_ss(&one);
inv_det = _mm256_div_ps(ones,det);

Tx = _mm256_sub_ps(rox,mv0x);
Ty = _mm256_sub_ps(roy,mv0y);
Tz = _mm256_sub_ps(roz,mv0z);

u = _mm256_mul_ps(dot8(Tx,Ty,Tz,Px,Py,Pz),inv_det);

float zero = 0.0f;
zeros = _mm256_broadcast_ss(&zero);
hits = _mm256_and_ps(hits,_mm256_cmp_ps(u,zeros,_CMP_GE_OS));
hits = _mm256_and_ps(hits,_mm256_cmp_ps(u,ones,_CMP_LE_OS));

resultsMask = _mm256_movemask_ps(hits)&255;
if (!resultsMask)
  return resultsMask;

Qx = cross8x(Ty,Tz,e1y,e1z);
Qy = cross8y(Tx,Tz,e1x,e1z);
Qz = cross8z(Tx,Ty,e1x,e1y);

v = _mm256_mul_ps(dot8(rdx,rdy,rdz,Qx,Qy,Qz),inv_det);

hits = _mm256_and_ps(hits,_mm256_cmp_ps(v,zeros,_CMP_GE_OS));
hits = _mm256_and_ps(hits,_mm256_cmp_ps(_mm256_add_ps(u,v),ones,_CMP_LE_OS));

resultsMask = _mm256_movemask_ps(hits)&255;
if (!resultsMask)
  return resultsMask;

t = _mm256_mul_ps(dot8(e2x,e2y,e2z,Qx,Qy,Qz),inv_det);

// should be &tMins[index] &tMaxes[index] but need to fix how BVH works
hits = _mm256_and_ps(hits,_mm256_cmp_ps(t,_mm256_load_ps(&tMins[0]),_CMP_GT_OS));
hits = _mm256_and_ps(hits,_mm256_cmp_ps(t,_mm256_load_ps(&tMaxes[index]),_CMP_LT_OS));

resultsMask = _mm256_movemask_ps(hits)&255;
if (!resultsMask)
  return resultsMask;

_mm256_store_ps(results.t,t);

_mm256_store_ps(results.P->x,_mm256_add_ps(rox,_mm256_mul_ps(rdx,t)));
_mm256_store_ps(results.P->y,_mm256_add_ps(roy,_mm256_mul_ps(rdy,t)));
_mm256_store_ps(results.P->z,_mm256_add_ps(roz,_mm256_mul_ps(rdz,t)));
```

```
  mv0x = _mm256_broadcast_ss(&m_vertexNormals[0].x);
  mv0y = _mm256_broadcast_ss(&m_vertexNormals[0].y);
  mv0z = _mm256_broadcast_ss(&m_vertexNormals[0].z);
  mv1x = _mm256_broadcast_ss(&m_vertexNormals[1].x);
  mv1y = _mm256_broadcast_ss(&m_vertexNormals[1].y);
  mv1z = _mm256_broadcast_ss(&m_vertexNormals[1].z);
  mv2x = _mm256_broadcast_ss(&m_vertexNormals[2].x);
  mv2y = _mm256_broadcast_ss(&m_vertexNormals[2].y);
  mv2z = _mm256_broadcast_ss(&m_vertexNormals[2].z);

  __m256 x,y,z;
  x = _mm256_add_ps(_mm256_add_ps(_mm256_mul_ps(_mm256_sub_ps(_mm256_sub_ps(ones,u),v),mv0x),
                    _mm256_mul_ps(u,mv1x)),
                    _mm256_mul_ps(v,mv2x));
  y = _mm256_add_ps(_mm256_add_ps(_mm256_mul_ps(_mm256_sub_ps(_mm256_sub_ps(ones,u),v),mv0y),
                    _mm256_mul_ps(u,mv1y)),
                    _mm256_mul_ps(v,mv2y));
  z = _mm256_add_ps(_mm256_add_ps(_mm256_mul_ps(_mm256_sub_ps(_mm256_sub_ps(ones,u),v),mv0z),
                    _mm256_mul_ps(u,mv1z)),
                    _mm256_mul_ps(v,mv2z));
  __m256 len, inv_len;
  len = _mm256_sqrt_ps(_mm256_add_ps(_mm256_add_ps(_mm256_mul_ps(x,x),_mm256_mul_ps(y,y)),_mm256_mul_ps(z,z)));
  inv_len = _mm256_div_ps(ones,len);
  _mm256_store_ps(results.N->x,_mm256_mul_ps(x,inv_len));
  _mm256_store_ps(results.N->y,_mm256_mul_ps(y,inv_len));
  _mm256_store_ps(results.N->z,_mm256_mul_ps(z,inv_len));

  for (int i = 0; i < 8; i++)
  {
    results.material[i] = this->m_material;
  }

  resultsMask = _mm256_movemask_ps(hits)&255;
  return resultsMask;
}
```

# Bibliography


[1] Stanford bunny model. Stanford University Computer Graphics Laboratory, Retrieved from http://graphics.stanford.edu/data/3Dscanrep/.

[2] Stanford dragon model. Stanford University Computer Graphics Laboratory, Retrieved from http://graphics.stanford.edu/data/3Dscanrep/.

[3] James Arvo and David Kirk. Fast ray tracing by ray classification. In *ACM Siggraph Computer Graphics*, volume 21, pages 55–64, 1987.

[4] Solomon Boulos, Ingo Wald, and Carsten Benthin. Adaptive ray packet reordering. In *IEEE Symposium on Interactive RAy Tracing, 2008. RT 2008*, pages 131–138. IEEE, 2008.

[5] Solomon Boulos, Ingo Wald, and Peter Shirley. Geometric and arithmetic culling methods for entire ray packets. Technical report, School of Computing, University of Utah, 2006.

[6] Marko Dabrovic. Sponza model.

[7] Manfred Ernst and Günther Greiner. Multi bounding volume hierarchies. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 35–40. IEEE, 2008.

[8] Samuli Laine and Tero Karras. Hairball model. NVIDIA Research.

[9] Erik Månsson, Jacob Munkberg, and Tomas Akenine-Möller. Deep coherent ray tracing. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 79–85, 2007.

[10] Paul Arthur Navrátil, Donald S. Fussell, Calvin Lin, and William R. Mark. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 95–104, 2007.

[11] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. Large ray packets for real-time whitted ray tracing. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 41–48, 2008.

[12] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages

101–108, 1997.

[13] Alexander Reshetov. Faster ray packets-triangle intersection through vertex culling. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 105–112, 2007.

[14] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 1176–1185, 2005.

[15] Attila T. Áfra. Faster incoherent ray traversal using 8-wide avx instructions. Technical report, Budapest University of Technology and Economics, Hungary and Babes -Bolyai University, Cluj-Napoca, Romania, 2013.

[16] Ingo Wald, Carsten Benthin, and Solomon Boulos. Getting rid of packets - efficient simd single-ray traversal using multiple branching bvhs. In *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008*, pages 49–57, 2008.

[17] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics (TOG)*, 26:6, 2007.

[18] Ingo Wald, Christiaan P. Gribble, Solomon Boulos, and Andrew Kensler. Simd ray stream tracing-simd ray traversal with generalized ray packets and on-the-fly re-ordering. *Informe Técnico, SCI Institute*, 2007.

[19] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, volume 20, pages 153–165, 2001.

[20] Hanson Zhang and Shenquan Liu. Order of pixel traversal and parallel volume ray-tracing on the distributed shared volume buffer. In *Visualization in Scientific Computing '95: Proceedings of the Eurographics*, pages 96–104, 1995.