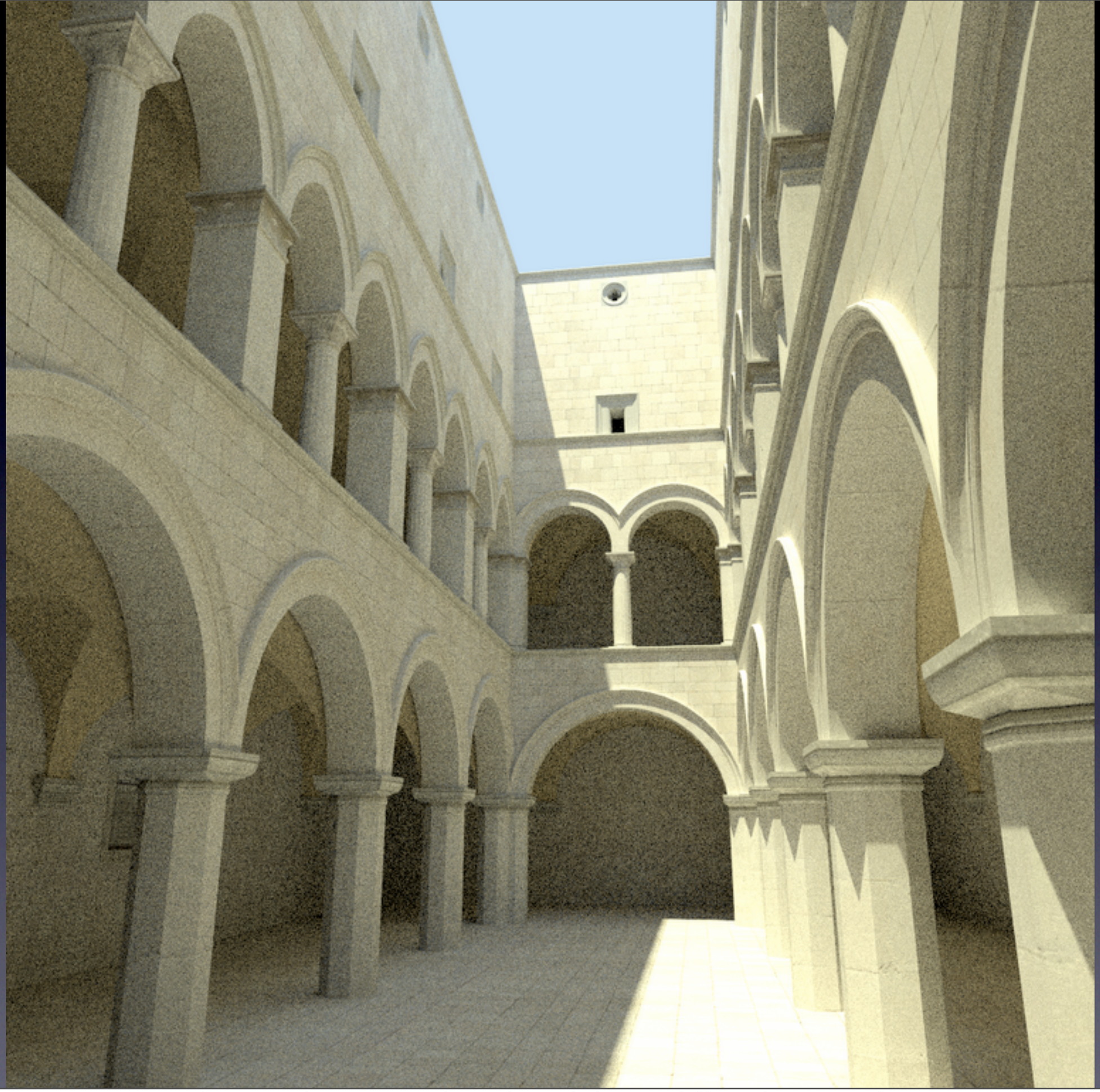
Adaptive Ray Packet Reordering

Solomon Boulos, Ingo Wald and Carsten Benthin

Hi, I'm Solomon Boulos from Stanford University and today I'll be presenting our paper on adaptive ray packet reordering. This work was done in collaboration with Ingo Wald and Carsten Benthin, both currently at Intel. As mentioned previously, this work started as a project a couple of years ago at Utah that we've all split up and turned into a few publications.

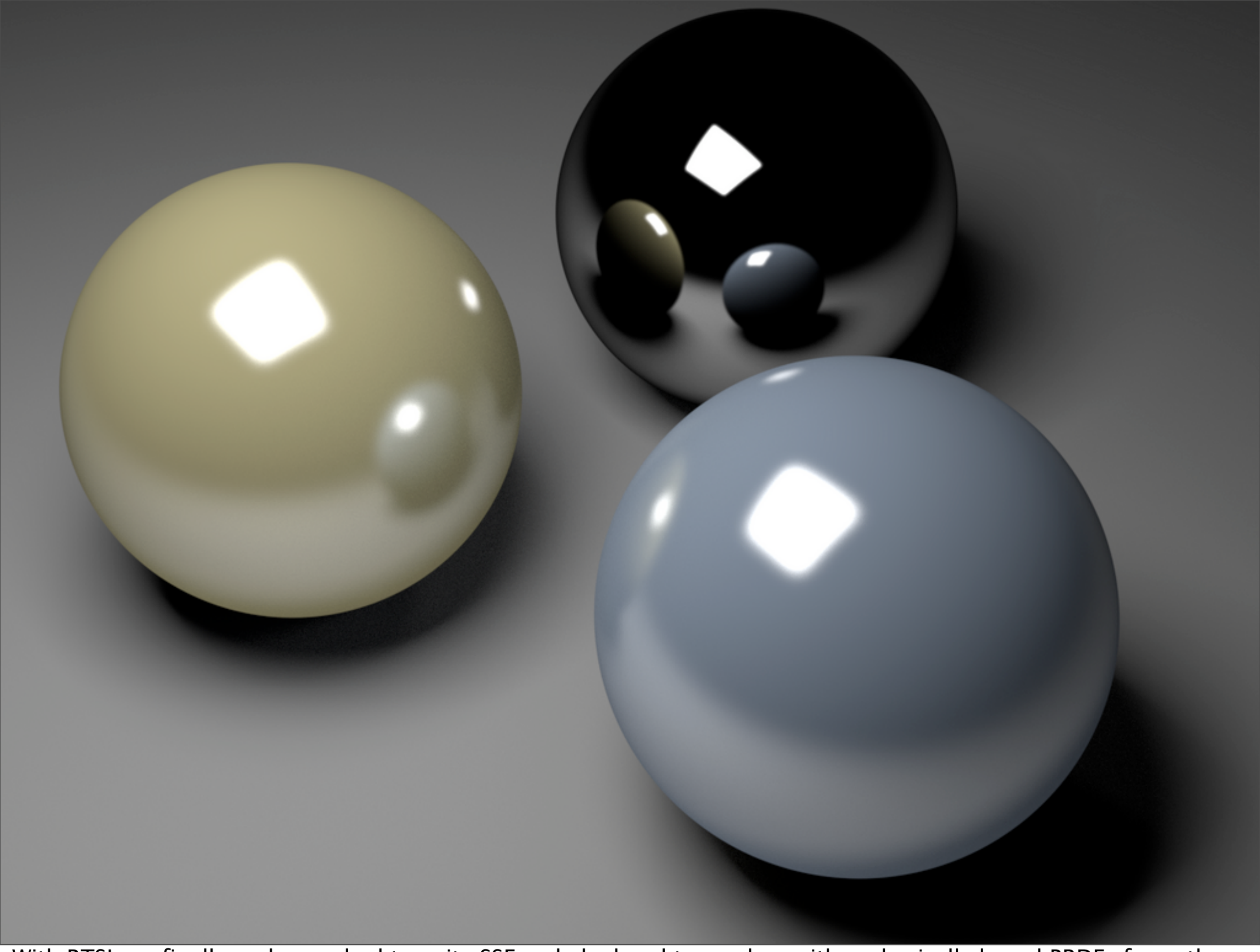


This is the Sponza atrium with 64 paths per pixel. Each path has 2 diffuse bounces for a total of 6 rays per path. It’s essentially a canonical image for any student’s path tracer, but sadly has been quite out of reach for the packet based ray tracing community. The point of this paper was to demonstrate a feasible reordering method to maintain high performance for these very incoherent ray distributions. But first for a trip down previous work, where I’ll only use my own work to make fun of.

Here’s a nearly useless test scene thanks to Pete, Ingo, Alex, and many others. The interactive ray tracing community very quickly caught on that all sorts of data structures could be used to ray trace this thing super fast with packets. However, it basically didn’t bring us any closer to our goals of producing pictures that ray tracing systems have been known for but it was pretty damn fast (particular Alex’s).

This was probably the best image we made for our original large packet paper. It looks like a z-buffer with a high shadow map resolution made it but much more slowly. In fact, you might notice that the plants look wrong due to their lack of alpha texture which is similarly true on the fairy's clothing. Overall, we traced only very coherent primary and second rays and argued that we were focusing on simply handing dynamic scenes.

Next up came our foray into distribution ray tracing. I was proud to say that we finally made some pictures I wasn't embarrassed to show. The biggest change over our previous work was that instead of just taking a big input packet and recursively tracing rays (with more and more of them needing strange masking to disable them), we filled in large queues of rays in order and only worked on the compact set of new rays. Writing these shaders in SSE was too painful, so most of it was scalar C++ code. This resulted in decreased shading performance, but was able to demonstrate useful gains over single ray traversal for somewhat incoherent rays. A downside, however, was that arbitrary shaders weren't possible; all rays were batched into queues and so had to carry a weight with them. Like others, we liked claiming that it was okay to use ray queueing.



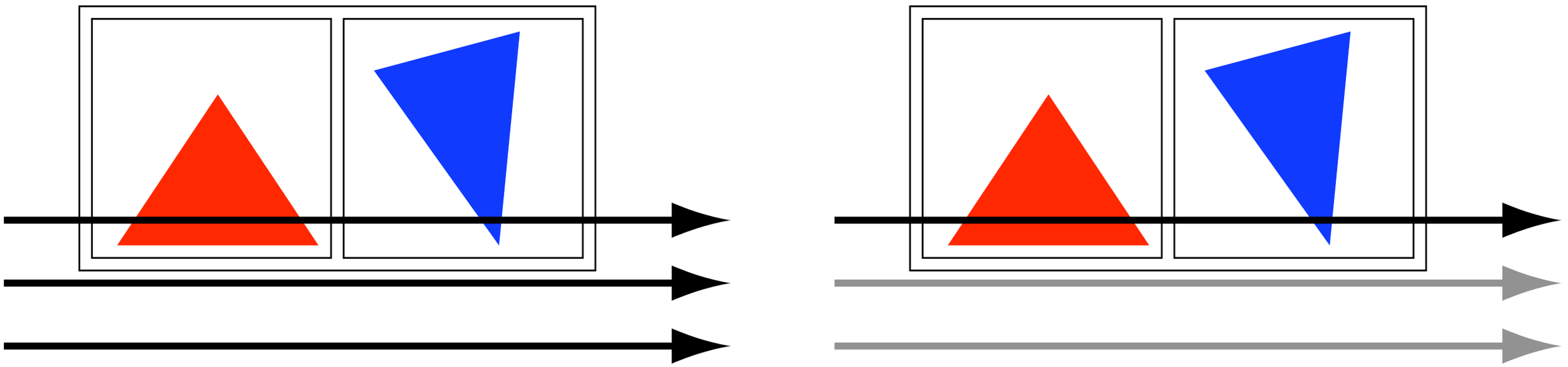
With RTSL we finally no longer had to write SSE code by hand to produce either physically based BRDFs for path tracing or arbitrary shaders for other pieces of code. This reduced our programming burden and for this paper was useful to be able to quickly prototype the shaders we used. Moreover, it's useful to reiterate the need to handle arbitrary shading and avoid ray queues as it seems like a slippery slope to have to write shaders in a higher level language that interact with these ray queues. As always of course, if you are a physically based purist you can go down the ray queue route; however, if you want to batch up shadow rays across different shaders you'll have to compute their shading results before shooting them which could be pretty expensive.

Overview

- Problems with current packet traversal
- Breadth First Ray Tracing
- Improving Full Packet Tests
- Adaptive Reordering Traversal
- Experiments and Results

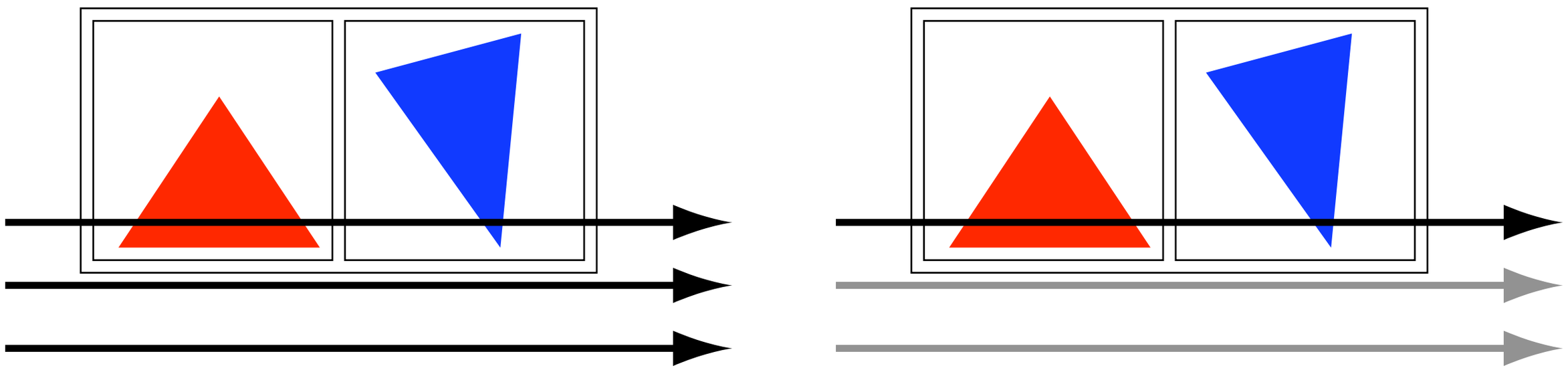
As a simple overview of this talk, I'll briefly discuss problems with current packet traversal methods wrt rays becoming inactive and how breadth first ray tracing improves the situation. Then I'll talk about some of the improvements we've made to the full packet tests from our original BVH paper, how we then leverage those for our adaptive reordering traversal, and finally talk about the experiments we ran and the results we obtained.

Inactive Rays



This figure from the paper demonstrates the current state of the world in packet traversal. Given a hierarchical structure such as this BVH and a packet of 3 input rays the first ray (top) drags the other 2 rays along with it. In SIMD packet tracing with a SIMD width of at least 3, the SIMD utilization would be very low. In BVH packet tracing, the 2 rays that should be marked inactive would not be noticed until the second level of node tests due to the speculative first hit test.

Breadth First Traversal



Instead, we could take the input set of rays on the left and filter out the inactive rays after the first box test. While this would probably require a linear sweep over all the rays to determine which are active or inactive, we wouldn't end up with a single ray dragging many others along with it. The only additional cost is in shuffling the ray data around to form a new packet. A downside of this traditional breadth first traversal, however, is that we lose the benefit of the quick first hit or all miss tests from recent packet tracing research.

Types of packet results

- All hit
- All miss
- Partial Overlap

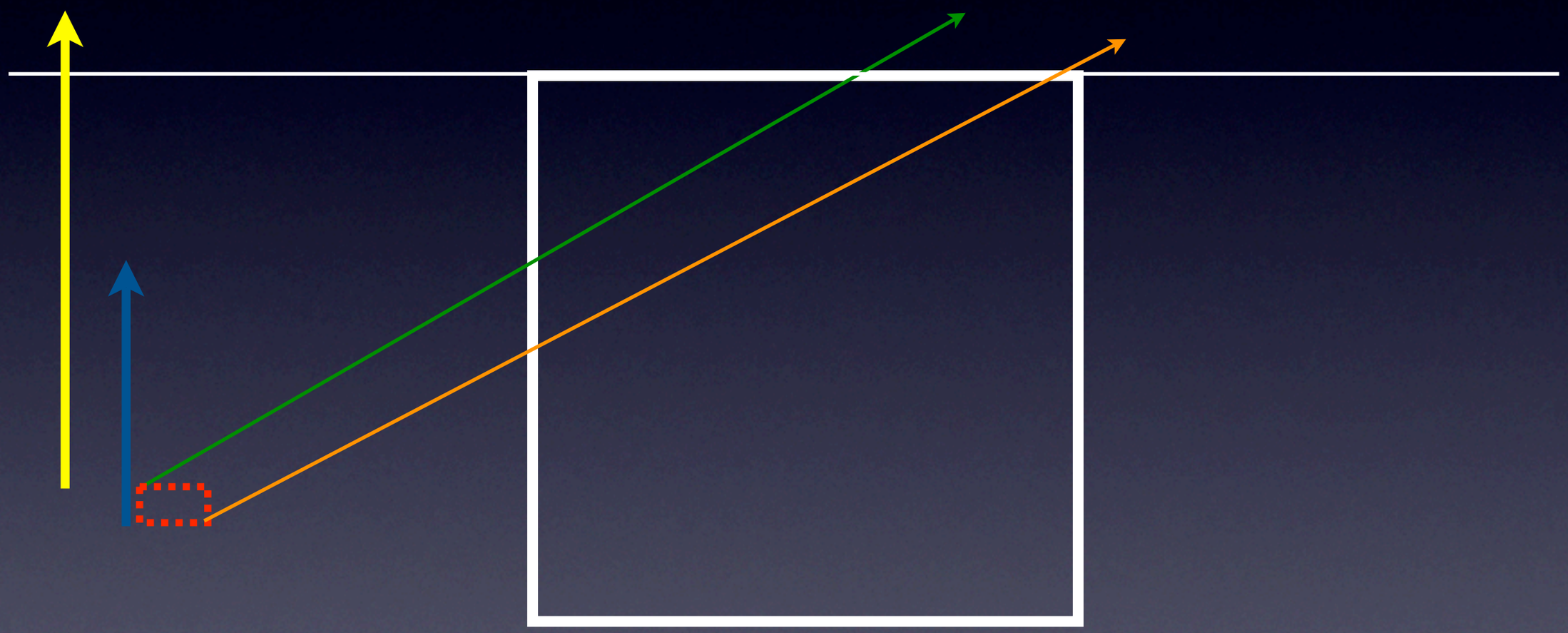
To frame the packet tracing problem, it's useful to consider that there are only three simple cases to consider for a ray packet test against primitives, boxes, kd-tree planes, etc. All the rays can hit, all the rays can miss, or there might be some overlap. In our original packet BVH paper, the all hit case was speculative based on the first ray in the packet and so could be fooled into dragging lots of inactive rays along with it. The all miss on the other hand was conservative and so while it might miss some possible cases for quick culling, it never drags inactive rays around. Finally, in the case of partial overlap the packet was only "resized" by moving the first active ray pointer forward. In this paper, we attempted to improve on all three of these issues.

Reordering Traversal

- If it's possible and useful, see if all rays hit
- If it's useful, see if all rays miss
- Test all rays
- If it's a good idea, compact the rays

Given the previous possible outcomes, there's a simple way to take advantage of them during traversal. For some definitions of possible, useful and good, we can determine which case of the previous 3 we are in and act accordingly. In our particular implementation, the first two tests use interval arithmetic and are disabled based on those possible and useful conditions. The test all rays loop is SIMD-ified as much as possible and then the compaction only happens when it's a good idea. On current architectures, compact the rays is not well supported so it's scalar in our prototype implementation (due to a lack of vector gather or scatter depending on how you see it). So let's start with how to do a quick all hit test that isn't speculative using interval arithmetic.

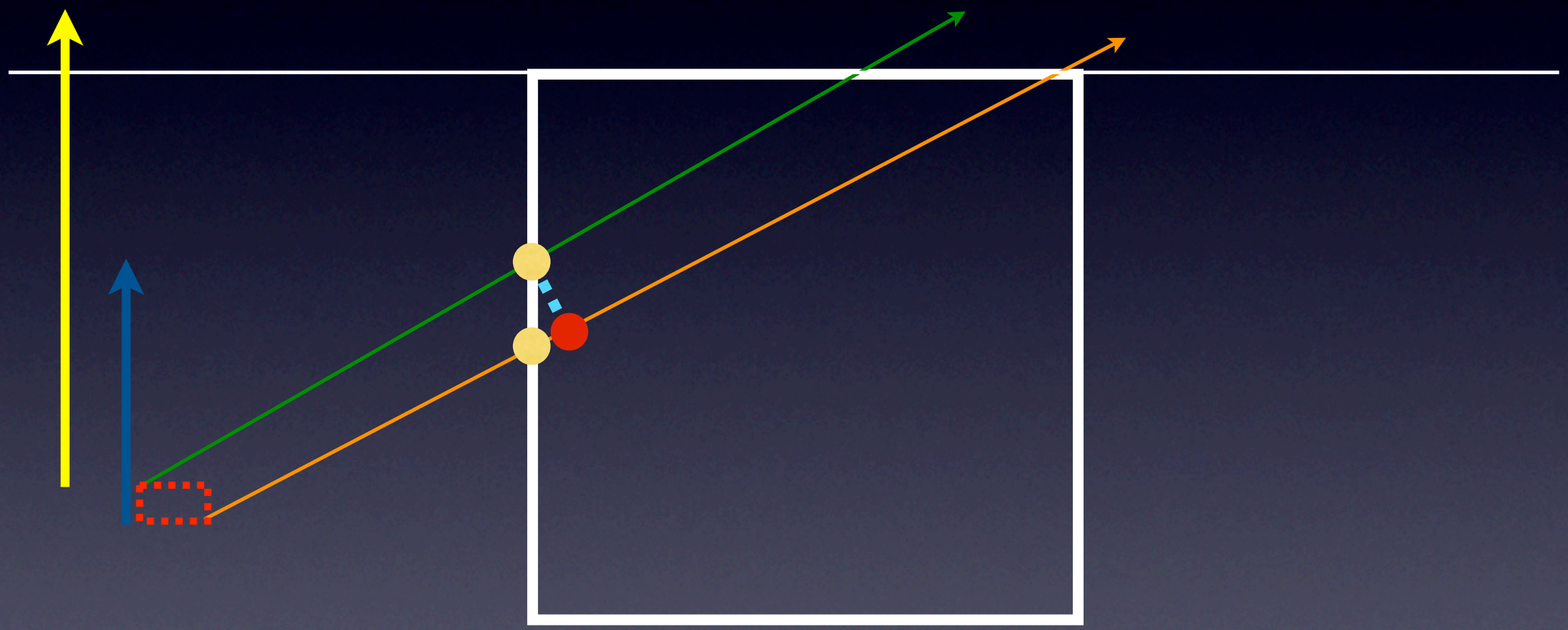
Interval All Hit



While interval arithmetic is not really geometric in nature, I hope that this poorly drawn diagram will be of assistance during the explanation of how to think about this test. First, we convert a set of rays (here shown in green and orange) into a single “Interval Ray”. This interval ray has an interval vector as its origin (shown here in the dashed red box) and an interval direction in place of a standard direction. For example, the interval origin is simply a bounding box of the ray origins as this represents min_x to max_x in the x axis and similar values for other dimensions. For direction the same idea is used min_dx to max_dx or in this diagram, min_dy and max_dy shown in blue and yellow respectively.

For a standard plane test (of which a BVH test is the union of 6 such tests) the ray origin in that dimension is subtracted from the plane position and then divided by the ray direction. With interval arithmetic, we instead take the plane position and subtract the interval of the ray origin before dividing by the interval of the ray direction. Instead of a single ray parameter, we obtain an interval of ray parameters. We can compute several of these for a single BVH node (one for each plane) and compose them using union and intersection operations. This eventually gives us a single interval ray parameter for the box entry (usually tmin) and box exit (usually called tmax). Combining the interval of ray parameters with the interval origin and interval direction produces an interval of hit points.

Interval All Hit



For example, in this case the standard box tmin (after all the intervals have been collapsed through intersection operations) is shown in yellow. If we take the green ray's tmin value (the larger of the two assuming both ray directions are normalized) and use it for the orange ray we can see that we're still inside the bounding box. In this case, we can safely say that all rays will hit the bounding box at some point. Note that this all hit test like the previous all miss test is still conservative; if there was a vast difference in ray lengths then it's possible that the orange ray when evaluated at the ray parameter would end up outside the box despite having intersected the box. Normalizing the ray directions helps alleviate this issue by ensuring that all rays treat ray parameters in the same units. Also, instead of testing each ray individually we apply the "inside the box test" using the interval ray instead of individual rays.

Possible/Useful?

- Possible: all rays are active / intervals up-to-date
- Useful: Might provide some speedup over linear testing (packet size > k * SIMD width)

So when is the all hit test possible and when is it useful? In the context of packet tracing, if you have inactive rays floating around and you don't update your intervals to reflect that (read: all previous papers) then the all hit test is useless: you already know that your rays don't all hit the current node since at least one missed a previous node. For the all miss test that's a little less clear; the intervals don't necessarily need to be updated to have the test succeed but the tighter they are the more likely it is.

If it is possibly useful, it's important to know when interval tests actually have some chance of being useful. While the probability of success is a function of the rays and the box, it's useful to consider the maximum possible benefit before even wasting time with it. In this case, if you consider the test as wasting SIMD width rays worth of effort you clearly need at least SIMD # of rays to get any benefit from an interval test. Because you're unlikely to get that 100% probability of an all hit test succeeding, you probably want to scale this factor by some value of k depending on your architecture. Again it depends how expensive the interval tests are relative to the linear sweep.

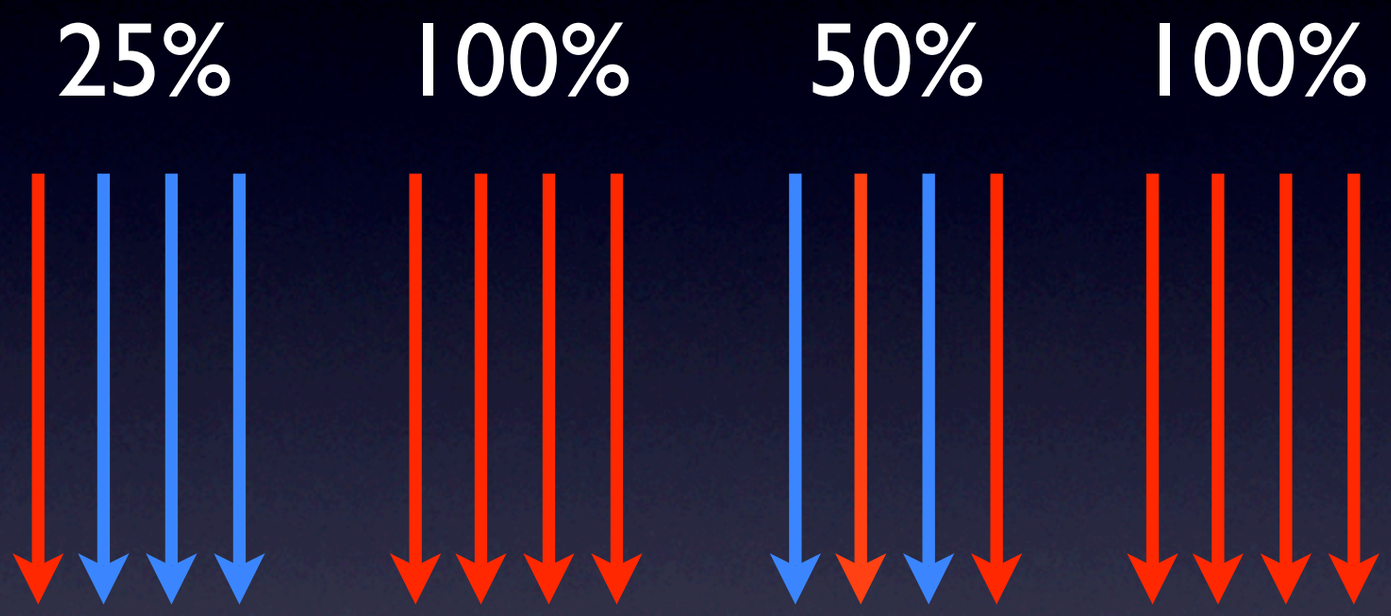
Adaptive Reordering

- Only reorder when it's a good idea
- Initial thought: cost function
- In practice: SIMD utilization < threshold

"True" breadth first ray tracing always takes an input set of rays and filters it depending on whether or not the rays take the same path through the BVH. In reality, you'd love to reorder or compact the rays whenever you know that you would reduce execution time. To do so, you'd have to model the average number of SIMD box tests, primitive tests, etc that will occur if you leave inactive rays in the packet, compare it to the cost of reordering in addition to the cost of tracing the reordered packet, and then hope for the best since your estimated cost is probably quite wrong. This analysis is essentially impossible without many assumptions that are too scene dependent to be useful.

On the other hand, the speedup gained in using SIMD operations is directly proportional to the SIMD utilization; in the case of ray packets this is the number of active rays per SIMD group averaged over the packet. This makes SIMD utilization a natural reordering metric that happens to be easily computed. As a side note, the utilization threshold allows for a natural continuum between always reordering (a la breadth first ray tracing), never reordering (SIMD packet tracing) and our reordering heuristic. We've found .5 to work pretty well in practice but could envision some wiggle room depending on architecture specifics (primarily the cost of shuffling data around).

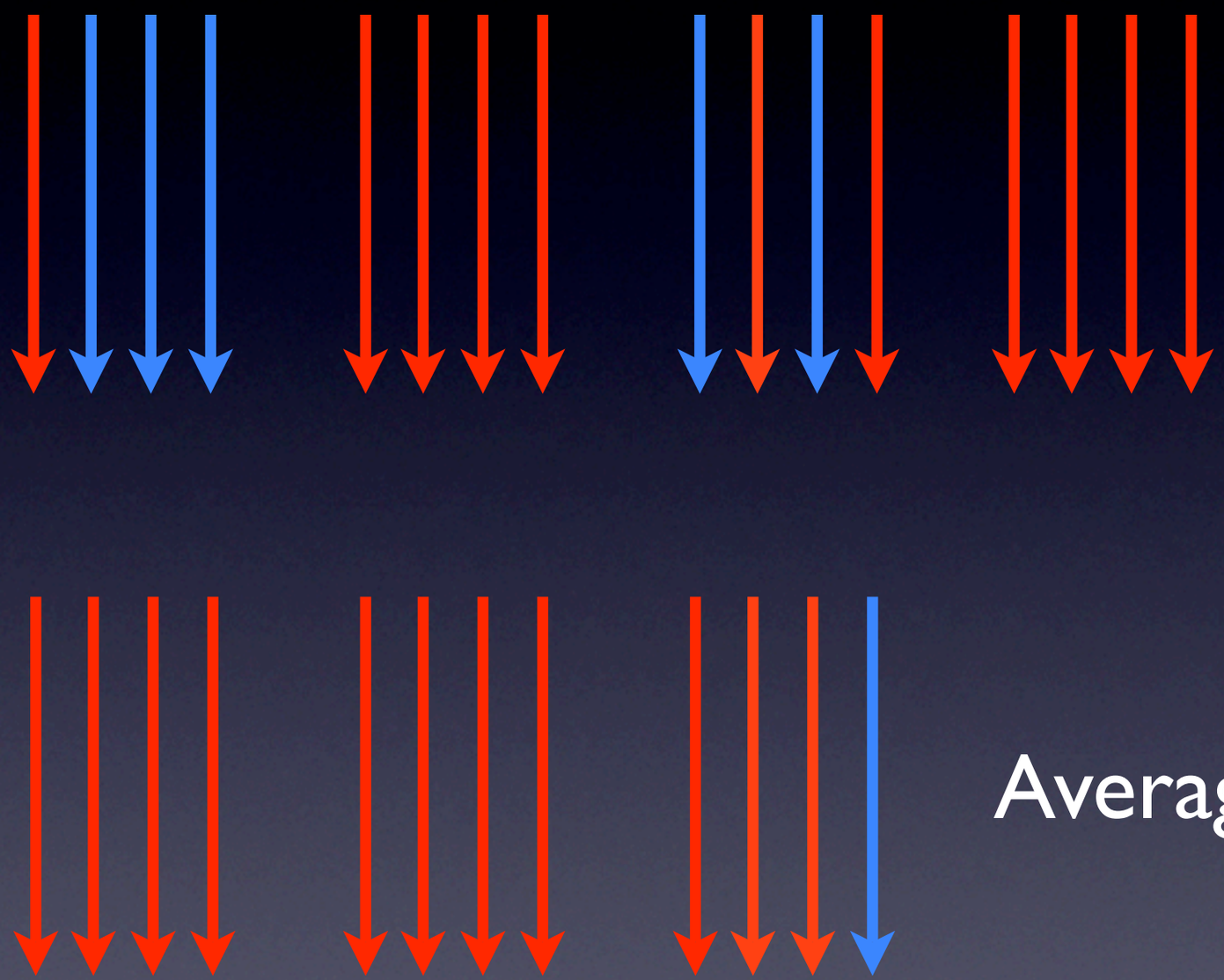
SIMD Utilization



Average = 68.75%

As a simple example, consider this packet of 16 rays with active rays marked in red (for hot) and inactive as blue (for cold). In this case, many rays are active, but they're slightly spread out. Overall, the SIMD utilization for this packet is around 68% which can be arrived at in many ways. The simplest from an implementation standpoint is actually to count the number of active rays and compare it to the size of the packet (so SIMD padded). With the latest hardware, there's a POPCNT instruction that could be useful for this case but counting the active rays is essentially free if you're already doing a linear sweep for the box tests.

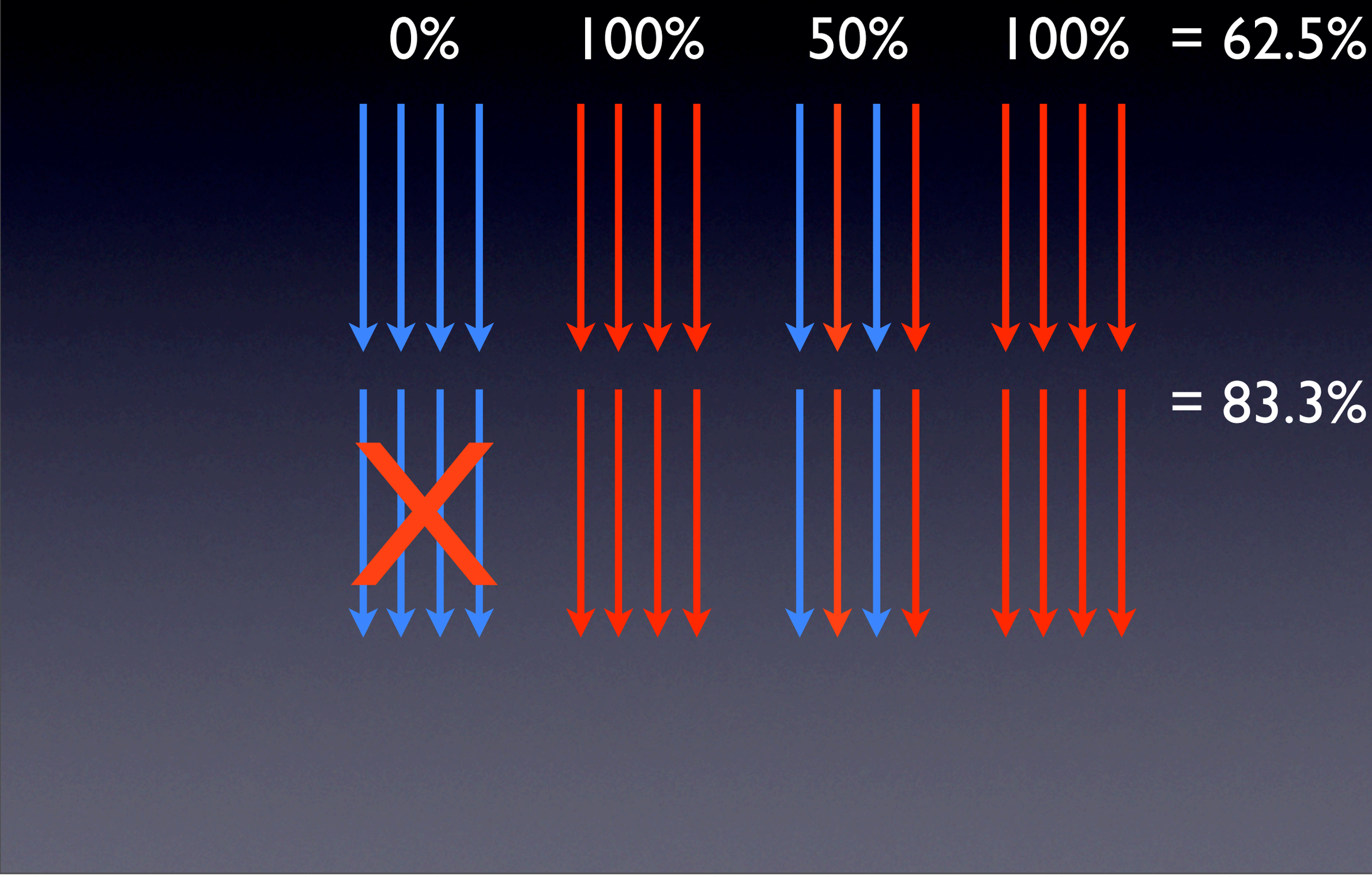
Reordering



Average = 91.67%

Reordering the packet then means shuffling all the ray data around so that the active rays appear in the first portion of the packet while a few remaining inactive rays are left trailing. In our Manta prototype I just make a new ray packet on the stack (lazyness wins) but we've also done an implementation for LRB that does the sorting in place and requires no extra memory. Just imagine the inactive rays moving to the right and the active ones moving to the left.

SIMD Trimming



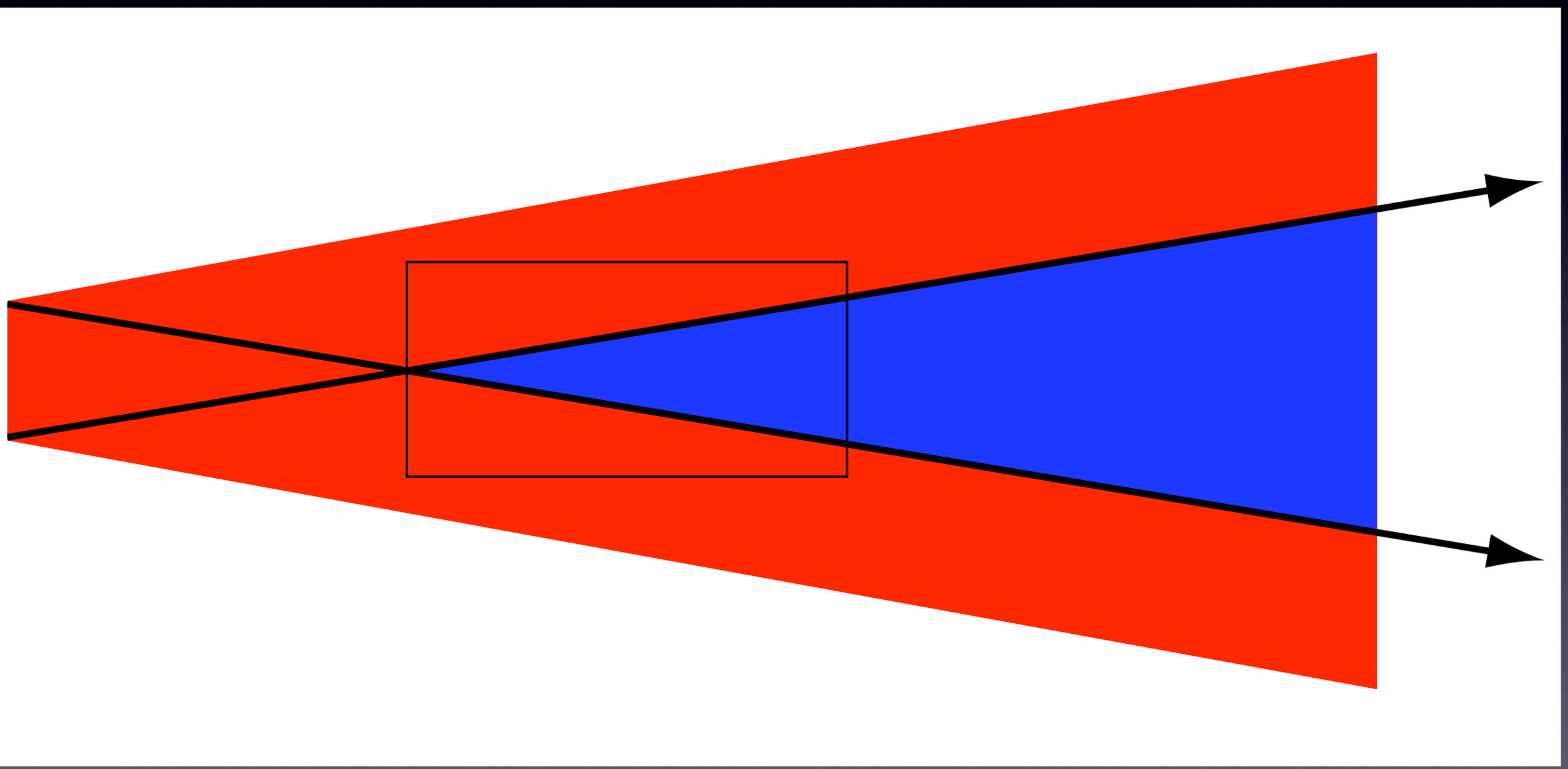
It’s important to note, however, that SIMD trimming (adjusting the first active and last active indices for a packet) can have a similarly positive effect on SIMD utilization. Because of it’s essentially free cost, we always first trim the packet from both ends before deciding what the new packet’s utilization will be. Because of this, we end up avoiding lots of unnecessary reordering.

Recomputing Intervals

- Doing a linear sweep, how about intervals?
- Only if useful (again, # active > k * SIMD)

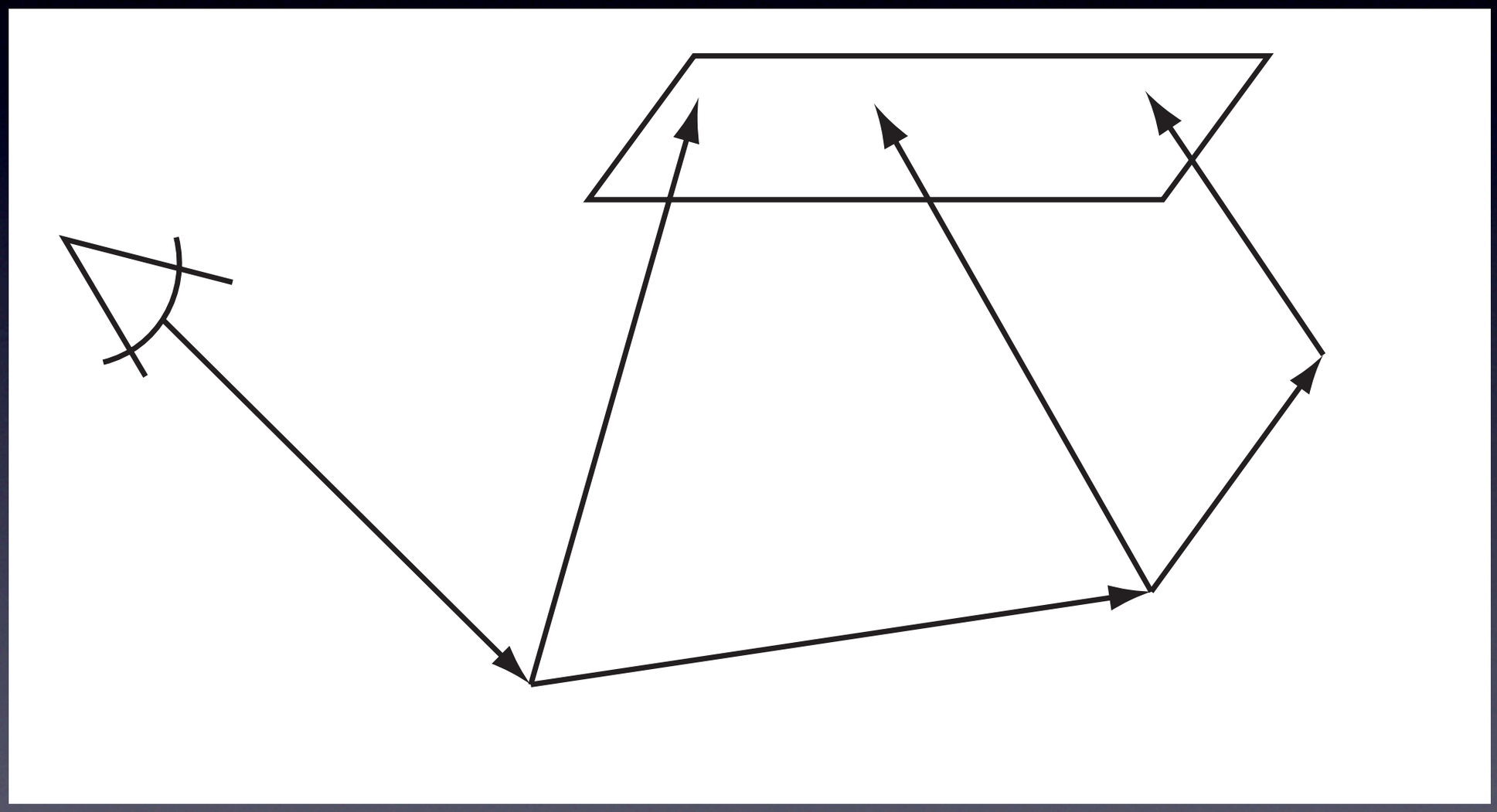
When we reorder the rays in the packet, we have an opportunity to recompute the ray intervals in hopes of improving the probability of success for the interval arithmetic tests. Of course this linear sweep isn't free either and more over, the maximum possible benefit of the IA tests decreases as our packet shrinks. As before, we just compare the number of active rays against some multiple of the SIMD width (as this is the assumed unit of comparison of the IA tests vs the SIMD box tests).

Tighter Intervals



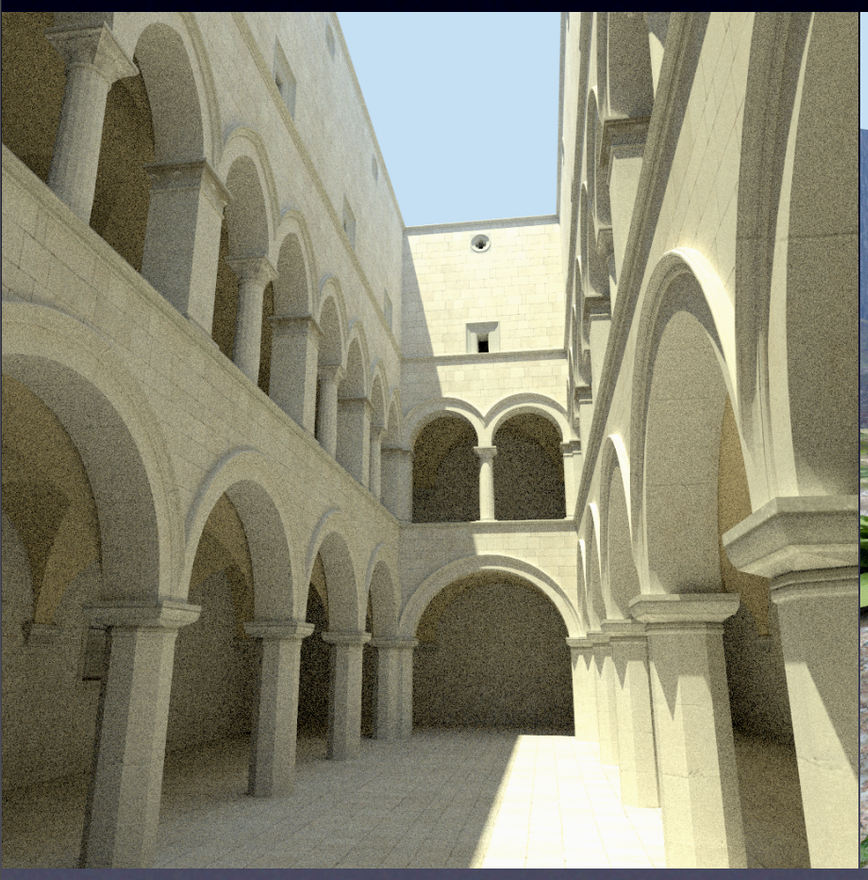
When we do recompute the intervals, we can do slightly better than just computing the interval origin and interval directions. Because we know that future tests are constrained to be within the child node we just intersected, we can move the effective ray origins to their entry points for the BVH node. When computing the tmax value for each ray we then use the difference between the exit point and entry point ray parameter values. This is somewhat similar to Alex Reshetov's "on the fly frustum" from last year, but applied to interval arithmetic. As we can see in some cases this should tighten the intervals considerably even if rays do not become deactivated.

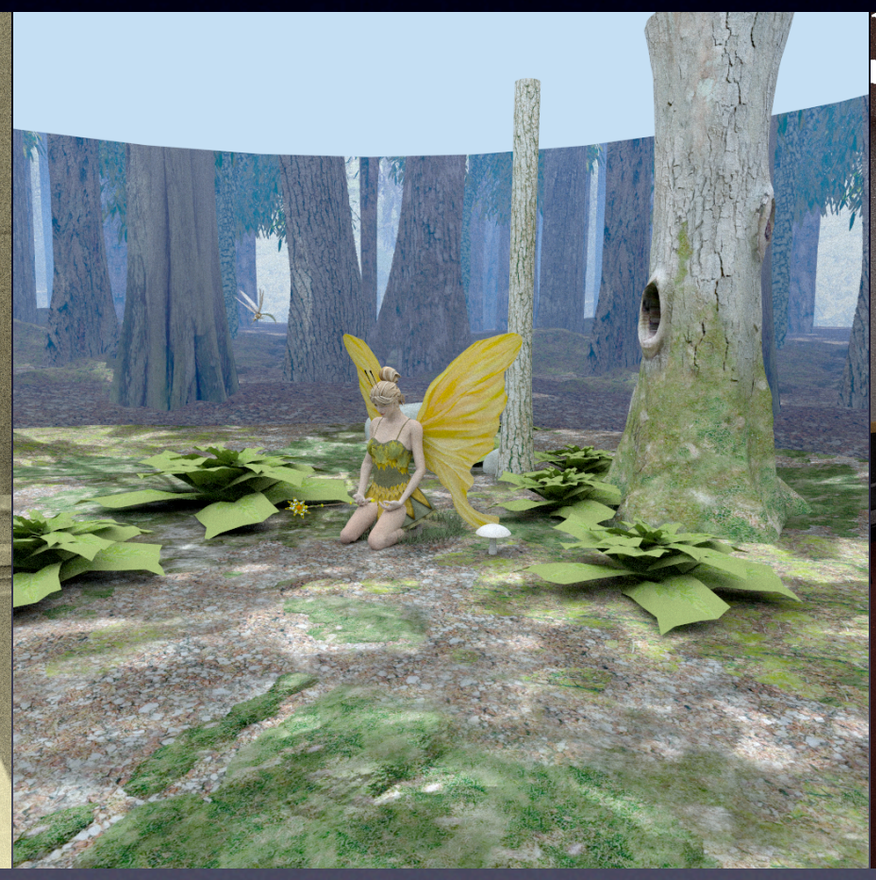
How many bounces?



In the setup I'm most interested in (glossy and diffuse reflection), I think 1 or 2 bounces is probably the setup of interest (as diagrammed here for 2 bounces). 1 bounce has been used in production rendering as it is sufficient for final gathering while 2 bounce would let you compute irradiance at the final gather points (as is used often as well). Similarly, in situations where ray tree attenuation is used for glossy setups many rays don't live past 2 incoherent bounces so a 2 bounce diffuse path trace seems to be fairly difficult. To span the spectrum of ray distributions though, we examined perfect specular reflection, glossy specular, and diffuse path tracing; all of these included soft shadows.

Which scenes?





For our scene setup, I chose the Sponza Atrium, Utah Fairy Forest and Conference room. These scenes span a gamut of model sizes, configurations and most importantly are easily available and renderable. Other scenes didn't make the cut since they either have too little geometry to be useful (Cornell Box and erw6), are not even somewhat representative of actual scenes (some tessellated mesh floating in space), or lack sufficient use to be comparative to other papers. While I feel all of these scenes we're using have their problems, they're what we've got to work with.

Implementation

- Branch of Manta
- 4-wide SSE (Intel Clovertown system)
- Reordering == new compacted ray packet
- All timings single core

To make sure we're all on the same page for the results here, our prototype implementation is a branch of the ever popular Manta interactive ray tracer. That being said, it targets SSE and so the SIMD width is a measly 4. Since we used a Clovertown with massive amounts of L2 cache, I went the lazy route and just used a new ray packet allocated on the stack to do the reordering (by just filling in the active rays only). Lots of stuff could be improved here, but I was more concerned about whether or not it would already show an improvement. All timings are going to be given for a single core, since core count seems to be a number that will change wildly. Everyone's renderer basically scales linearly anyway, so if you want to imagine my system as being faster scale by 8 or 16 if you'd like. Now, on to the measurements.

Specular Performance

Scene	Single	SIMD	Packet	Reordered
Sponza	7.7	3.4 (2.2×)	2.1 (3.7×)	2.5 (3.1×)
Fairy	5.5	2.9 (1.9×)	2.2 (2.6×)	2.2 (2.5×)
Conference	7.8	4.2 (1.9×)	2.8 (2.8×)	2.8 (2.8×)
Sponza	15.4	8.0 (1.9×)	6.6 (2.3×)	7.3 (2.1×)
Fairy	7.5	4.4 (1.7×)	3.8 (2.0×)	3.5 (2.1×)
Conference	17.3	10.0 (1.7×)	8.2 (2.1×)	7.9 (2.2×)
Sponza	27.6	18.5 (1.5×)	22.1 (1.2×)	21.0 (1.3×)
Fairy	9.5	6.0 (1.6×)	5.8 (1.6×)	4.8 (2.0×)
Conference	32.2	22.8 (1.4×)	24.4 (1.3×)	22.0 (1.5×)

Table 1: Performance in seconds per frame for the three different traversal methods with 2 (top), 5 (middle) and 10 (bottom) perfect specular reflections. For SIMD, packet, and reordered traversal, the relative speedup over single ray is also computed. Reordering is competitive with packet tracing for small numbers of bounces and pulls further away for higher numbers.

As a first sanity check, I want to ensure that we're not losing much performance off of speculative packet tracing for fairly coherent rays. Because only a few bounces of perfectly specular reflections seems like a pointless comparison, I've gone ahead and run this test with 2, 5, and 10 bounces of specular reflection (with each bounce sending a single shadow ray towards a large area light). Ray tree attenuation is not used to avoid thinning out the rays. As the caption indicates, we retain most of the benefits of packet traversal but perform better for higher bounce depths.

Glossy Performance

Scene	Single	SIMD	Packet	Reordered
Sponza	2.4	1.0 (2.4×)	0.5 (4.8×)	0.7 (3.4×)
Fairy	2.1	1.1 (1.9×)	0.8 (2.6×)	0.8 (2.6×)
Conference	2.3	1.2 (1.9×)	0.9 (2.6×)	0.8 (2.9×)
Sponza	5.2	2.9 (1.8×)	2.2 (2.4×)	2.2 (2.4×)
Fairy	4.4	2.5 (1.8×)	2.2 (2.0×)	2.0 (2.2×)
Conference	5.0	2.9 (1.7×)	2.4 (2.1×)	2.0 (2.5×)
Sponza	8.2	5.3 (1.5×)	4.9 (1.7×)	4.3 (1.9×)
Fairy	5.8	3.8 (1.5×)	3.7 (1.6×)	3.0 (1.9×)
Conference	8.0	5.2 (1.5×)	4.9 (1.6×)	3.9 (2.1×)

Table 2: Performance in seconds per frame for the three different traversal methods with 0 (top), 1 (middle) and 2 (bottom) glossy bounces. As before, relative speedup over single ray is given in parentheses.

For glossy specular reflections (with the same material setup used in the original glossy conference scene I did) a similar pattern is visible: reordering is competitive with packet tracing for coherent rays and still better than single SIMD tracing for incoherent rays. I should mention that I unfortunately first rounded off the render time and then computed speedup, this makes bounce 0 a little difficult between the packet and reordered since they're so fast.

Diffuse Performance

Scene	Single	SIMD	Packet	Reordered
Sponza	2.4	1.0 (2.4×)	0.5 (4.8×)	0.6 (4.0×)
Fairy	2.1	1.0 (2.1×)	0.8 (2.6×)	0.8 (2.6×)
Conference	2.3	1.2 (1.9×)	0.9 (2.6×)	0.7 (3.3×)
Sponza	6.0	3.9 (1.5×)	4.4 (1.4×)	3.4 (1.8×)
Fairy	4.1	2.7 (1.5×)	3.0 (1.4×)	2.2 (1.9×)
Conference	5.0	3.3 (1.5×)	3.7 (1.4×)	2.6 (1.9×)
Sponza	8.8	7.3 (1.2×)	9.8 (0.9×)	7.2 (1.2×)
Fairy	5.3	4.1 (1.3×)	5.0 (1.1×)	3.1 (1.7×)
Conference	8.0	6.0 (1.3×)	7.7 (1.0×)	5.3 (1.5×)

Table 3: Performance in seconds per frame for diffuse path tracing 0 (top), 1 (middle) and 2 (bottom) diffuse bounces. While many methods are competitive for the shadows only case (0 bounces) BVH packet tracing become as slow as or worse than single ray by the second bounce.

As before, the relative performance between the different methods is fairly similar. It is important to note that this is a real implementation with only 4 wide SIMD and that our results also come close to the theoretical maximums we had hoped for with breadth first ray tracing given this input size of packets (256 rays in this case). We'll see this theoretical comparison in the next slides when comparing against exact reordering. Also, while the gain over SIMD packet tracing is fairly small in for very incoherent rays, we'll see that for larger SIMD width we would expect a larger relative gain in performance.

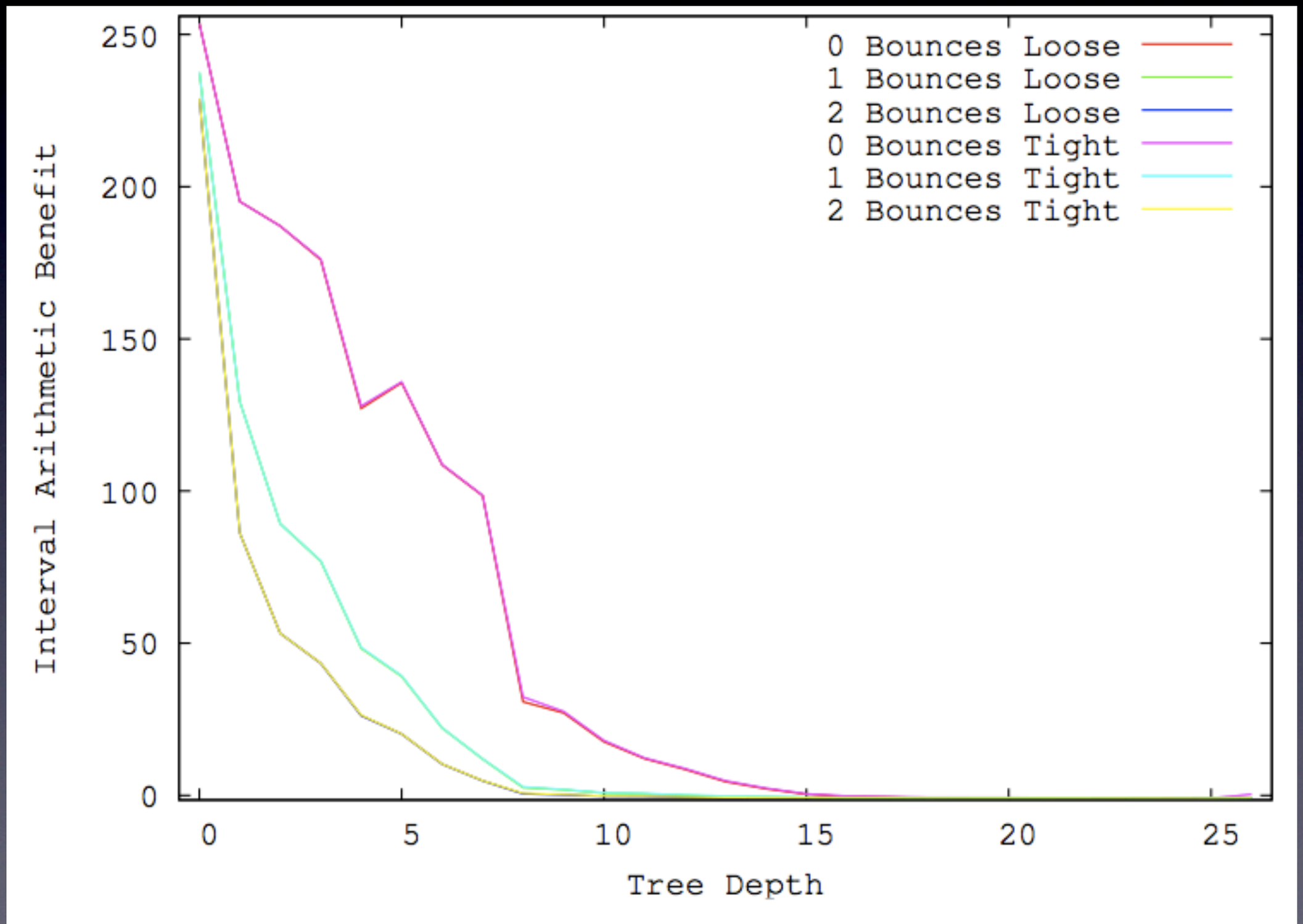
Arch Independent Stats

	Single	SIMD	Packet	Adaptive	Exact
Box	46.6	23.1	49.4	16.3	14.0
IA	0.0	0.0	6.7	0.2	0.2
Reorder	0.0	0.0	0.0	3.2	8.9
Tri	6.0	4.7	7.5	4.9	4.2
Trav	46.6	23.1	7.9	4.5	4.5
Prim	6.0	4.7	3.1	2.2	2.2

Table 4: Performance independent statistics for 2 bounce diffuse path tracing on the Conference scene. All numbers are averages per ray for the full rendering. A SIMD box or triangle test counts as 1 test.

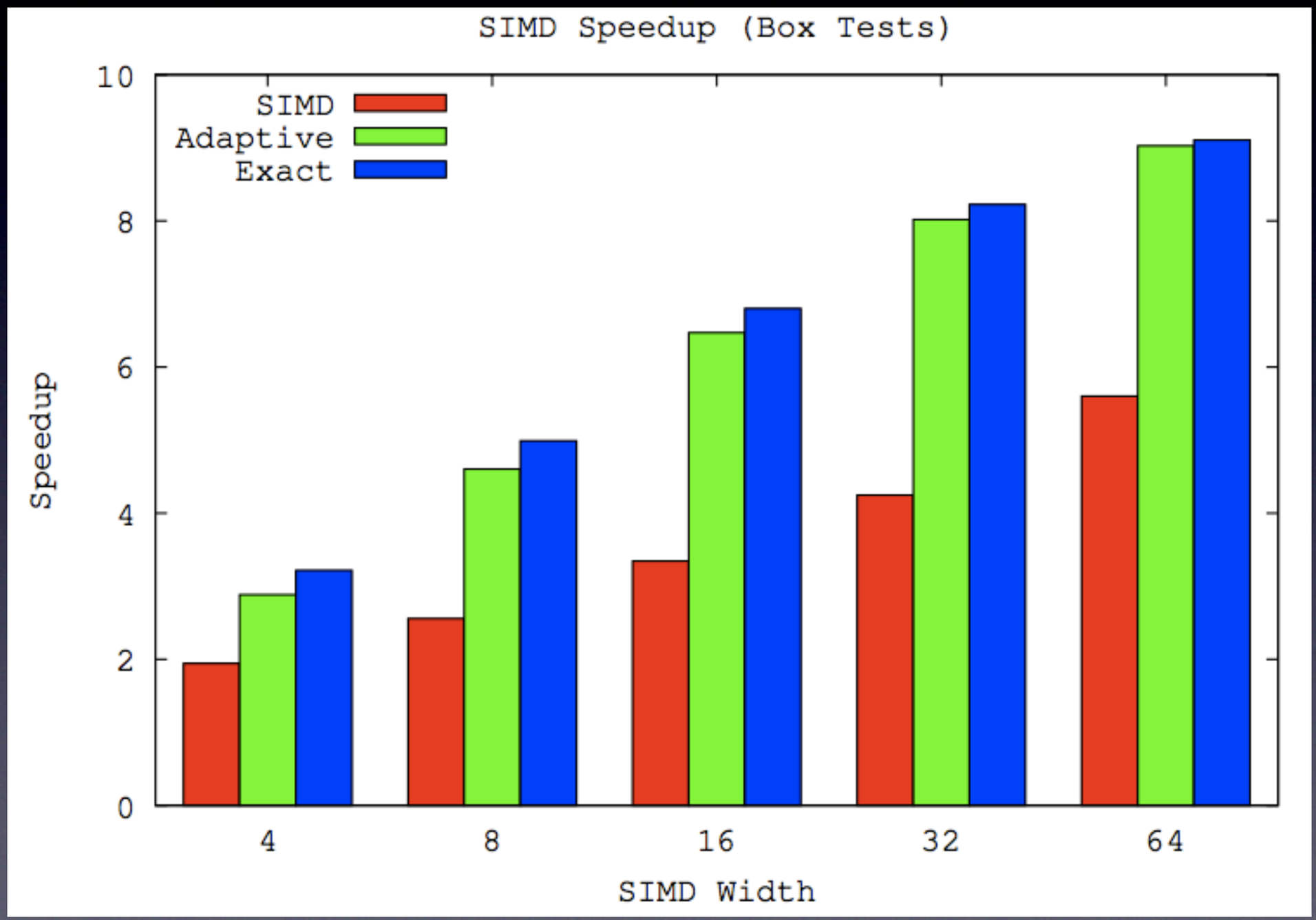
Since your architecture may change in the future and your implementation details may differ, I find it useful to also compare architecturally independent statistics across the different traversal methods for the 2 bounce diffuse path tracing case. Our adaptive reordering compares favorably to exact/breadth first ray tracing in terms of a number of statistics such as box tests per ray, triangle tests per ray, traversal steps per ray and primitive tests per ray. It does so with about 3x less reordering moves per ray. Each reordering move corresponds to shuffling a single ray's data from its current position to a new position; as this code does not use in-place reordering, this number is actually higher than it could be. We can also see what has gone wrong with the packet traversal: it's doing more box tests than the single ray traversal and we're doing a lot of IA tests as well. Combine that with the higher number of triangle tests, we're really just dragging around a big packet to intersection tests that aren't desired.

IA Benefit



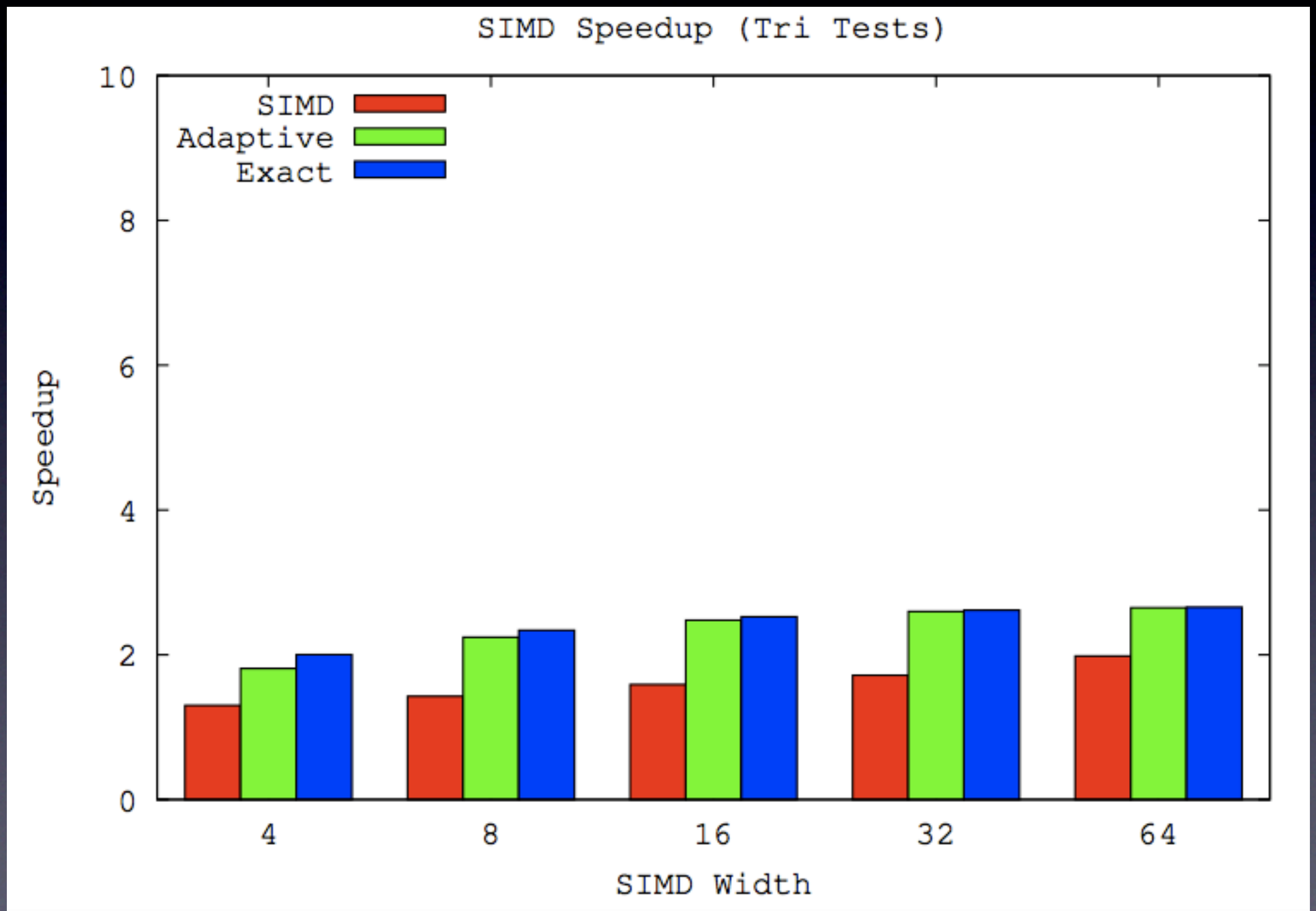
This graph shows the number of rays “skipped” by using our combination of IA tests and compares tight vs loose bounds (which I affectionately call the IA benefit). I’m certain you can’t see this, however, the tight bounds are slightly better than the loose bounds but certainly not worthy enough for their own publication ;). I unfortunately didn’t think to compare with not recomputing the intervals... Loose vs tight intervals might matter more on other scenes or ray distributions, but I felt I’d include it in the paper anyway. This slide also demonstrates the decreasing benefit of the interval arithmetic tests beyond even a few levels in the BVH for incoherent rays. The performance benefit is quite high for coherent rays, however, so a system that doesn’t use large packet tests would lose quite a bit for coherent primary, shadow, and perfect reflection rays.

SIMD Speedup: Boxes



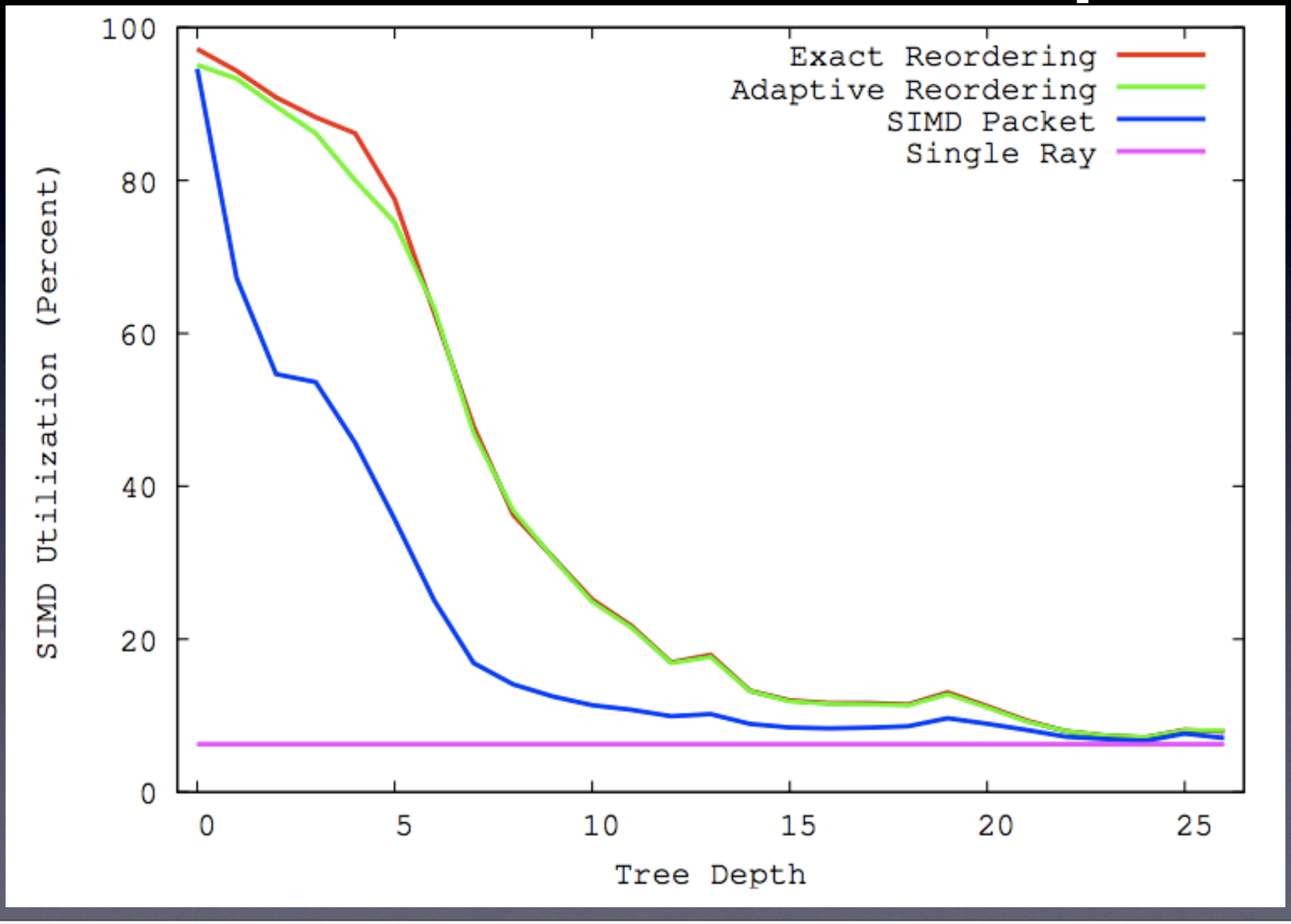
Current GPUs use larger than 4-wide SIMD units, and Intel's upcoming CPUs will have 8-wide SIMD in 2010. If SIMD happens to keep going up, here is how our method will compare with SIMD packet tracing in red, adaptive reordering in green and exact reordering in blue for the speedup over scalar box testing. As we can see, all methods enjoy some benefit to increased SIMD width even for this 2 bounce diffuse path tracing setting. A nice takeaway from this side that our method is more beneficial as SIMD width grows, but that if you prefer to avoid the headache of packets larger than SIMD width (which could have code autogenerated) you're still probably not much more than 2x from optimal. Then again, if you have the space for larger packets this benefit could be stretched further still; on the far right with 64-wide SIMD we only have 4 SIMD groups with our 256 rays.

SIMD Speedup: Leaves



The story for primitive intersection is much more grim however. For an input set of 256 rays, no matter what we do on average about 2 rays hit the same triangle. For large SIMD widths, this will then quickly become the bottleneck for ours and any other packet method. Our assumption that there was enough coherence within a window of 256 rays is simply not true at the primitive intersection level. Because of this, there has been a lot of N-ary BVH work recently including our own paper that appeared yesterday and Holger Dammertz's paper at EGSR. Similarly, Alex Reshetov's shallow trees combined with vertex culling seeks to solve the same problem; as we mention in the paper, however, the vertex culling approach relies on highly coherent rays to perform quick rejections of the primitives within the leaves. Luckily, since we can detect when our packet is heavily inactive, we won't be worse than SIMD packet tracing in this case and could even switch to an efficient single ray SIMD-width primitive testing traversal. It's important to remember, however, that while this speedup is not only very low at the leaves, but at all the bounding boxes just above them.

SIMD Benefit vs Depth



Finally I think it's useful to see where the cross over point is for SIMD utilization. This graph demonstrates our SIMD speedup including both box and primitive tests assuming a 16-wide SIMD width. Clearly we maintain a much higher utilization than SIMD packet tracing and do so for a longer period of time. Everything gets bad towards the leaves, but depending on your implementation there might be an interesting way to get the number to go back up just after it declines sharply. Our investigation with the N-ary BVH suggests something like that should be possible.

What's broken?

- Not enough coherence at leaves
- Resource requirements
- Not very large gain over SIMD tracing

As I always ask people what's broken about their stuff, I figure I should just tell you up front what you already probably have realized: the coherence at the leaves is so low that this method provides essentially no benefit. What you might not have noticed is that since I'm using 256 rays per packet, this might not be very amenable to resource constrained architectures such as recent GPUs or custom ray tracing hardware; cache vs compute is always a difficult tradeoff and many recent FLOP rich monsters have chosen compute quite heavily. Finally, even my plot out to 64-wide SIMD suggests that SIMD tracing is only about 2x less efficient for box testing; with heavily reduced IA benefit at 2 diffuse bounces, our method quickly becomes dominated by SIMD utilization. In many situations, I would guess that the cleanliness of code maintenance would win out instead.

What's right?

- Arbitrary shading
- Invisible to other parts of the renderer
- Retains coherent performance
- Less sensitive to packet size
- Simple

Then again, I think there are several things that are right about our approach. The first two are more important than most publications admit lately, but I think allowing arbitrary shading is still useful to give people the flexibility to do crazy things with your renderer. Removing the need to have other portions of the renderer attempt to generate very coherent rays in the first place is also incredibly beneficial. Even rays that are wildly different will magically separate into coherent groups when the time is right (which is as soon as they disagree enough). Similarly, if very coherent rays are put in at the top, they stay together and there is a benefit to doing so. Finally, the metric used is simple and therefore robust and cheap. In contrast to some of the complicated reordering schemes that we and others have attempted, I'm not scared that there will be some crazy case that breaks the reordering heuristic: when enough rays become inactive, you should regroup if possible and useful. Like I said before, the worst part is that you have more code than before.

Future Directions

- Enormous ray packets
- Transition from packets to single ray
- More beneficial full packet tests

As for some future directions. The SIMD utilization also improves for various statistics if we increase the packet size further; the only real issue is things like rays not being kept in registers, polluting caches and so on so enormous ray packets (think 1M rays per packet) might possibly have some benefit on certain architectures. On more resource constrained systems though, it's probably just useful to apply this as much as possible and know that you'll run out of coherence eventually. Once we're down to a single SIMD group, we currently switch to the SIMD traversal (unless we know we only have a single ray), but we could also switch to something like an N-ary BVH approach. Finally, I always hope that full packet tests, either geometric or interval based, might be improved further.

Questions?
Complaints?

- Code available as a Manta branch:
- https://code.sci.utah.edu/svn/Manta/branches/reorder/