Ian Mallett
CS 6960.001

Final Project Report

Original Goals:

The original goal of the project was to break 1e9 rays/second on a reasonably sized chip by any means necessary. I proposed three major ideas:

1: Storing the entire current path through the BVH on the stack

2: Tracing rays with a DDA within each thread to handle broad traversal

3: Dynamic rescheduling of DDA ray queues

Original Plan:

0: Fix/improve the pathtracer

1: Implement idea 1

2: Implement idea 2

3: Implement idea 3

I anticipated being able to finish steps 0, 1, and possibly 2, with 3 being an improbable “stretch goal”.

Outcome:

Step 0 happened fairly straightforwardly. However, there was much more room for optimization than I had realized. Consequently, I spent far longer in this stage than I had intended. I do not regret this, since this extra optimization was valuable effort toward the performance goal. The rough numbers (no precise data available) indicates I ultimately got a significant performance boost merely by thoroughly optimizing the given code.

After finishing step 0, I proceeded to implement step 1. This, again, turned out to take much longer than expected. The first reason is that further investigation revealed even more optimizations were possible in the underlying code, and second, a series of implementation-related problems. These ranged from unimplemented instructions, bugs in the simulator, to my adding compiler flags without checking for regressions. Additionally, making a sane implementation of this weird stack-structure without a debugging environment was difficult.

The third and probably most important reason was I also realized that idea 1 had *a lot* more potential than I had initially given it credence. In particular, I was able to expand the algorithm to use all stack space (and efficiently, at that).

At the end of doing all of this, the semester was nearly over, so it was natural to focus the rest of the time on making the most of this improved idea. The most difficult part was ensuring correctness; stack deletion in particular is easily half the algorithm. I also implemented a profiler so I can give statistics about ray performance.

In the end, I have been partially successful in achieving > 1e9 rays/second in a reasonably sized chip. I did this by doing more work than I had anticipated on fewer ideas than I had anticipated. To say my algorithm is a panacea for all scenes is inaccurate; the improvement in performance is exponential in

stack size but inverse to traversed nodes. Performance ranges from about 1e8 to 3e9 rays/second depending on the scene. The utter randomness of path tracing is a worst case for any memory-coherency-bound application. Consequently, the algorithm should be combined with better scheduling of rays to processors.

Algorithm Description:

The basic idea is to store as much as possible on the stack. Originally, this was to be only a single traversal, but I quickly realized that one should store as much data as possible.

As nodes are encountered in the traversal, the traversal path is checked in the stack. If a node is already in the stack, then it does not need to be loaded from memory. If the node is not in the stack, then it is loaded into the next available slot in the stack.

Eventually, of course, for most scenes the stack will run out of space. To make room for more, stack elements must be deleted. This is accomplished by the heuristic *delete as much as possible as far away as possible*. This heuristic was chosen for implementation ease and also for a number of key properties. First, if traversals are relatively coherent, faraway nodes can be deleted safely, as it is unlikely they will be needed specifically again. Second, deletions were modeled to be expensive, so the fewer deletions executed, the more efficient the algorithm. Of course, any deleted nodes may be needed again, and deletion may be cheap in hardware.

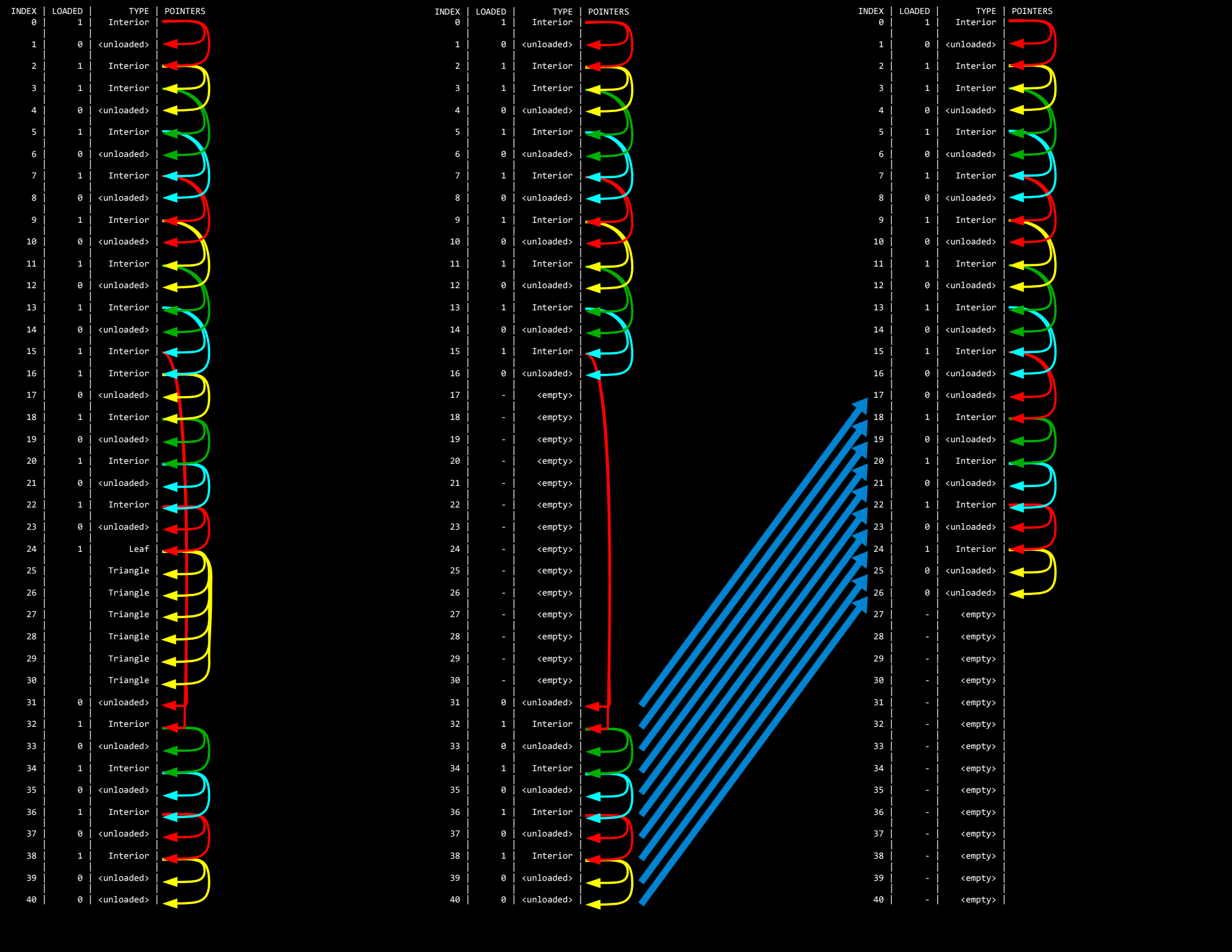
The notion of distance is defined as vertical distance within the traversal tree. To implement the heuristic, the highest loaded BVH branch not on the current traversal path is found (this is efficient to find) and then it and all of its children are unloaded recursively. This can unload up to nearly 50% of the tree at a time.

A visual depiction of the deletion algorithm running on a tiny stack is presented on the next page.

INDEX	LOADED	TYPE	POINTERS
0	1	Interior	
1	0	<unloaded>	
2	1	Interior	
3	1	Interior	
4	0	<unloaded>	
5	1	Interior	
6	0	<unloaded>	
7	1	Interior	
8	0	<unloaded>	
9	1	Interior	
10	0	<unloaded>	
11	1	Interior	
12	0	<unloaded>	
13	1	Interior	
14	0	<unloaded>	
15	1	Interior	
16	1	Interior	
17	0	<unloaded>	
18	1	Interior	
19	0	<unloaded>	
20	1	Interior	
21	0	<unloaded>	
22	1	Interior	
23	0	<unloaded>	
24	1	Leaf	
25		Triangle	
26		Triangle	
27		Triangle	
28		Triangle	
29		Triangle	
30		Triangle	
31	0	<unloaded>	
32	1	Interior	
33	0	<unloaded>	
34	1	Interior	
35	0	<unloaded>	
36	1	Interior	
37	0	<unloaded>	
38	1	Interior	
39	0	<unloaded>	
40	0	<unloaded>	

INDEX	LOADED	TYPE	POINTERS
0	1	Interior	
1	0	<unloaded>	
2	1	Interior	
3	1	Interior	
4	0	<unloaded>	
5	1	Interior	
6	0	<unloaded>	
7	1	Interior	
8	0	<unloaded>	
9	1	Interior	
10	0	<unloaded>	
11	1	Interior	
12	0	<unloaded>	
13	1	Interior	
14	0	<unloaded>	
15	1	Interior	
16	0	<unloaded>	
17	-	<empty>	
18	-	<empty>	
19	-	<empty>	
20	-	<empty>	
21	-	<empty>	
22	-	<empty>	
23	-	<empty>	
24	-	<empty>	
25	-	<empty>	
26	-	<empty>	
27	-	<empty>	
28	-	<empty>	
29	-	<empty>	
30	-	<empty>	
31	0	<unloaded>	
32	1	Interior	
33	0	<unloaded>	
34	1	Interior	
35	0	<unloaded>	
36	1	Interior	
37	0	<unloaded>	
38	1	Interior	
39	0	<unloaded>	
40	0	<unloaded>	

INDEX	LOADED	TYPE	POINTERS
0	1	Interior	
1	0	<unloaded>	
2	1	Interior	
3	1	Interior	
4	0	<unloaded>	
5	1	Interior	
6	0	<unloaded>	
7	1	Interior	
8	0	<unloaded>	
9	1	Interior	
10	0	<unloaded>	
11	1	Interior	
12	0	<unloaded>	
13	1	Interior	
14	0	<unloaded>	
15	1	Interior	
16	0	<unloaded>	
17	0	<unloaded>	
18	1	Interior	
19	0	<unloaded>	
20	1	Interior	
21	0	<unloaded>	
22	1	Interior	
23	0	<unloaded>	
24	1	Interior	
25	0	<unloaded>	
26	0	<unloaded>	
27	-	<empty>	
28	-	<empty>	
29	-	<empty>	
30	-	<empty>	
31	-	<empty>	
32	-	<empty>	
33	-	<empty>	
34	-	<empty>	
35	-	<empty>	
36	-	<empty>	
37	-	<empty>	
38	-	<empty>	
39	-	<empty>	
40	-	<empty>	



Performance Data:

The following data was collected while rendering a 256*256 image, with 1spp and a ray depth of one (primary plus shadow). Since shadow rays are not handled specially, this is similar to a ray depth of two without shadowing. Simulation times (ranging from ~10 minutes to several hours) prevented more experiments. The hairball scene in particular ate 30 hours of compute time before I gave up. I suspect simulation times were slow primarily because they were running in a virtual machine.

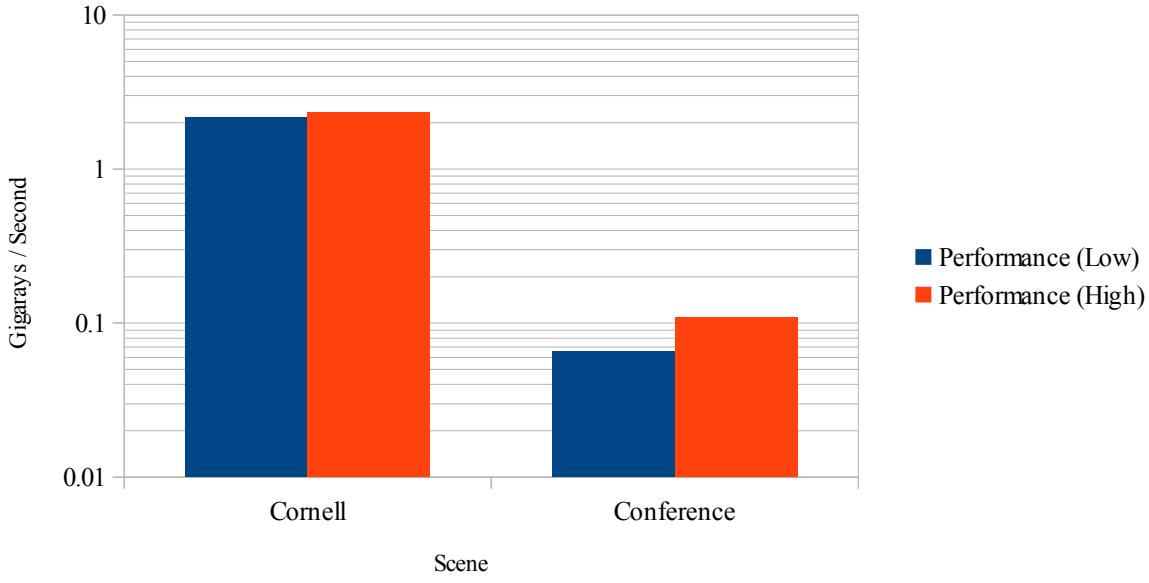
Since the object of the experiments was to test the effect of stack caching, using more samples per pixel is equivalent to rendering a larger image (but with a more complex implementation). Similarly, the access patterns for a deeper ray depth would be different (and worse), which would prevent elucidating more complicated performance metrics. This does, however, limit the applicability of this analysis.

The configuration chosen had 512 TMs, with each TM having 8 TPs. Each TP had one 8-banked ICache. No L1 or L2 caches were present. Each TP had:

1 INTMUL, FPINV, CONV
4 FPADD, FPMIN, FPCMP, FPMUL, BITWISE
8 INTADD

I found this to be a reasonable balance, if not excessive. The largest (real) instruction resource conflict was caused by the “mul_s” instruction with 120,326 cycles (divided over 512*8 threads and a minimum 1,524,159 total compute cycles, this is ~0.002% of each thread's total time) in the Conference scene. The chip size was < 300mm^2.

The following simple chart summarizes the results. Note the vertical log axis. In my implementation, threads stall when no more work is available.



The minimum and maximum performance of any particular thread is represented. In practice, this is a strongly right-skewed distribution. In fact, 46% of all stalled time in the conference scene was spent idling at the BARRIER instruction (that is, waiting for other threads to finish). Across 4096 threads, this corresponds to the 50% performance difference between the fastest and slowest thread seen in the

result. In reality, threads would be processing rays deeper and rescheduling. So, average-case performance should be read as slightly below the higher number to midway between.

Difficulties:

The main difficulty was the simulator itself. Danny very excellently was able to add needed instructions, but ultimately working with the simulator—with its lack of debugging support, slow execution speed, and awkward build process—was the main inhibitor to progress. Even with the customization I did, simulator-related issues accounted for the bulk of the time I spent.

I spent a long time trying to figure out what I assumed to be a memory corruption bug in my deletion code. After much deliberation, up to and including writing a verifier for the stack's integrity, I finally discovered that a compilation flag I had added had corrupted the BVH build. It was satisfying that my stack deletion code—ludicrously complex as it is—was in fact correct, but concerning that so much time had been spent to confirm this. The issue was especially subtle since the error only occurred on the Conference scene; the Cornell scene, in particular, was unaffected.

Conclusions: Utility:

I can only reasonably draw conclusions about idea 1.

I think it's pretty obvious that there's plenty to be gained simply from a careful optimization of existing code, but the major performance boost here comes from this stack caching thing. When loading nodes from memory is expensive and localstore/stack space is free, caching as many nodes as possible on the stack is an obvious and winning thing to do.

The implementation isn't easy, of course, and the efficiency could be substantially improved simply by implementing this cache in hardware. I think one would see the greatest gains from implementing idea 1 in tandem with a dedicated ray rescheduling policy (such as idea 2 and 3, or treelet streaming).

Conclusions: Future Work:

Neither idea 2 nor 3 were implemented. I feel like idea 1 was a huge step forward, but at the same time, it is clearly not a standalone solution. Memory access patterns are fundamentally the same; this is just a smarter cache. Ideas 2 and 3 should be combined and implemented together, on top of idea 1.

The compiler flags I added and had to remove as they caused the “memory corruption” bug in the simulator should be added back in. Since I wrote the majority of my code under the assumption that the flags would be enabled, the code does not have micro-optimizations I know the compiler would do. For example, “y=x*x*x*x” should be optimized as “temp=x*x; y=temp*temp;”.

It's definitely my fault that I didn't actually check that adding these optimized flags would break the simulator, but I still don't understand why they do. There oughtn't to be anything in a BVH builder that relies on precise IEEE associativity rules or similarly pedantic nonsense. The only exception might be the use of infinity, but the finite math flag wasn't enabled since my box intersection method uses it.

The shadow ray traversal should be an optimized special case.

I essentially ignored DRAM. For large scenes, the stack cache is still underfed, and so performance is limited by DRAM bandwidth. Ray rescheduling would help this, but pandering directly to DRAM's inane structure might also be of use.

The deletion heuristic should be reconsidered, and implemented in simulated hardware.