Bonsai Path Tracer: Exploring Scene-Specific Rendering Optimizations

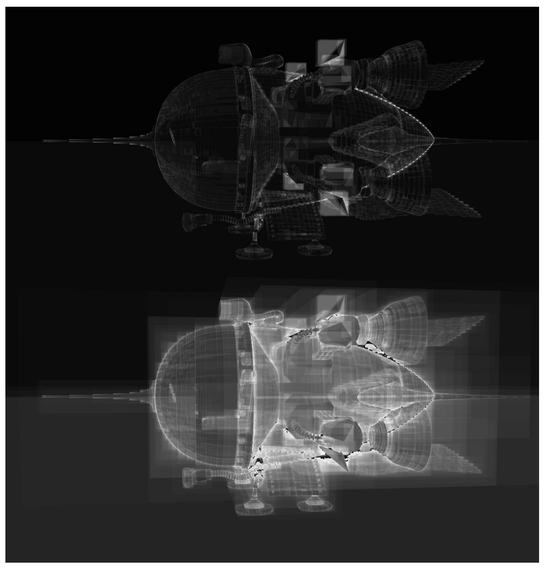
Elton Manchester, AJ Root, Kayvon Fatahalian
Department of Computer Science, Stanford University

Background

Path tracing is a rendering technique that simulates the bouncing of light rays through a scene. This often involves tracing billions of rays through scenes with millions of triangles, making efficiency vital. In particular, ray-scene intersection takes the bulk of render time.

To speed up this task, most renderers use a bounding volume hierarchy (BVH) to arrange triangles into a tree structure based on their bounding boxes. Once the BVH is created, further gains can be found by improving the parallelism of traversal. Towards this goal, various methods have been developed to increase thread convergence and to even out the workloads of each thread group.

We want to investigate the relationship between scene characteristics and method performance towards the goal of automatically generating the highest quality optimizations.



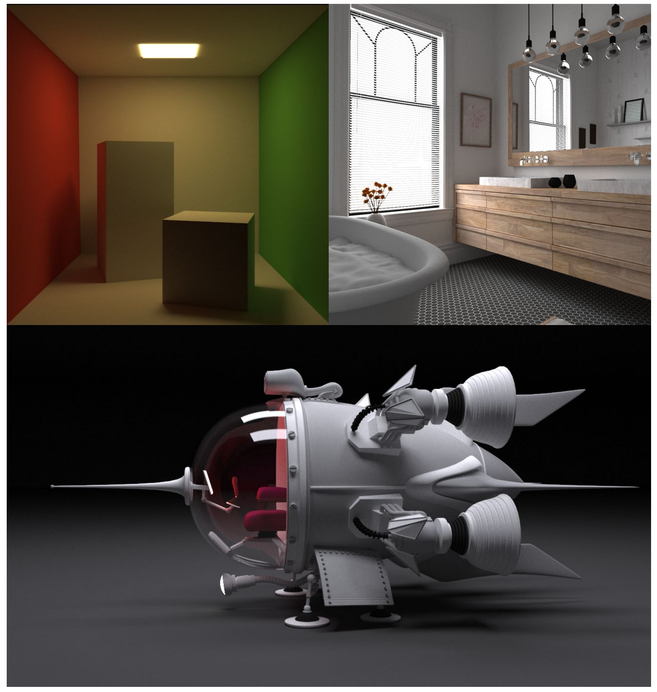
Top: Triangle intersection tests performed by each ray
Bottom: BVH nodes traversed by each ray

Algorithms

Rendering pipelines can be expressed as a composition of smaller algorithms and optimizations.This renderer focuses on GPU rendering, so the listed techniques are only relevant in GPU applications.

BVH Traversal Algorithms
while-while: While the current node is an interior node, go to the next node. While the current node has triangles to intersect, intersect the next triangle.
if-if: Replaces the while loops with if statements. This reduces memory coherence, but reduces the time threads must wait for their desired traversal step.
speculative: Upon reaching a leaf node, threads continue traversal until the entire warp has a leaf node to check. Since some threads would execute traversal anyway, joining them should increase throughput.
Thread Management
thread-per-pixel (default): Each thread is responsible for tracing every sample for a single pixel.
persistent threads: Upon ray termination, each thread fetches a new ray from a global queue. This addresses work-balancing issues where long-running warps hurt performance.

Rendered Scenes



Top left: Cornell Box, Top Right: Bathroom
Bottom: Spaceship

Scene credit: Benedikt Bitterli rendering resources

Render Times (s)

Algorithm	Cornell Box	Spaceship	Bathroom	Dragon	Camera Rays
polygon count	36	457557	592187	831812	592187
while-while	2.039	15.69	39.945	5.282	1.244
persistent while-while	**1.598**	14.994	45.371	4.825	2.546
if-if	2.052	12.263	**37.259**	3.946	**1.147**
persistent if-if	1.718	**11.986**	41.357	**3.793**	2.16
speculative	2.266	17.02	44.864	5.613	1.303
persistent speculative	1.864	16.04	48.453	5.294	2.417

Analysis

First and foremost, the variety of top-performing methods shows that the optimal rendering algorithm is scene-dependent. This behaviour demonstrates the importance of analyzing the best use-cases for each technique.

Of the traversal algorithms, if-if has the best overall performance. Compared to while-while, this is likely due to the reduction in long-running warps. This explanation is corroborated by while-while's relatively large improvement from persistent threads, which specifically targets long-running warps. The bad performance of speculative traversal is somewhat surprising, since the behavior should be similar to while-while. One possible explanation is the overhead from frequent warp synchronization.

Thread management was largely scene-dependent. Performance improvements from persistent threads are to be expected due to the reduction in variance for warp runtimes. However, the "Bathroom" and "Camera Ray" tests saw worse performance from persistent threads. In both cases, there is no early ray termination, significantly reducing warp runtime variance and therefore also the usefulness of persistent threads. This effect is amplified further by convergent camera rays.

Overall, if-if is a safe choice. Meanwhile, persistent threads should only be used in cases with high variance in path length and complexity (such as with an open sky or with early termination in Monte Carlo methods).

Future Work

- Test a larger algorithm space, including wavefront and megakernel with in-depth scheduling.
- Develop heuristics to determine a favorable combination of techniques for a given scene.
- Generate automated, optimized pipeline code based on a given scene and BVH format.