Improvements to Distributed Ray Tracing Via Binned Batches

Shane Boles

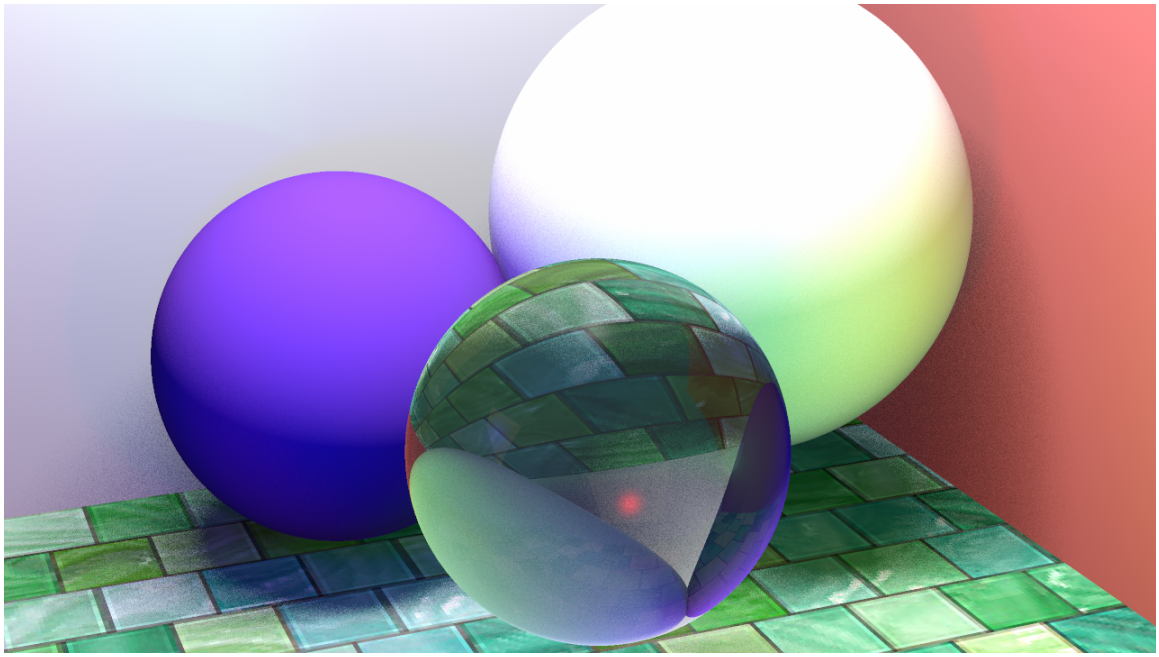

Figure 1: A test scene rendered within 19 minutes, presenting complex refraction and a high-gloss reflector.

ABSTRACT

In this paper we present the time-based limitations of single-threaded ray tracing systems, the performance issues faced by ray tracing in general, and one method of parallel programming to eliminate prohibitively high run times and generate more graphically interesting scenes.

1 INTRODUCTION

1.1 Ray Tracing

The struggle to reduce run time in graphics could be said to be one of the key reasons a researcher would choose to focus on the field. And even if one were to have no interest whatsoever in investigating methods to reduce run time, the benefits are immediately apparent: less time spent waiting. This saved time compounds upon itself in an iterative environment, giving a superlinear growth of productivity. Alternatively, this saved time is time that may be spent to further refine the result at no additional cost. Sometimes the correct outcome to choose is obvious, or perhaps even necessary.

Indeed, ray tracing is a great example of one such problem that has no answer, or at least has yet to be determined. As implemented in [Whitted 1980], the ray tracing algorithm consists of the conceptually simple idea that if light was able to bounce through a scene and enter the eye, one may cast a ray against the path of the light to sample where it came from. By treating each pixel a light receptor of the eye we can sample over the entire visible scene. Furthermore, we can use this algorithm recursively on perfect mirrors and perfect refracting objects as well, by simply doing the necessary mirroring/angling and re-firing the trace; thus we can sum the light at any visible point, provided we can determine which point the trace intersects first, or at all.

Due to the "perfect" nature of the trace, however, phenomena arising from imperfections could not be simulated. With Cook et al.'s contribution on the topic from [Cook et al. 1984] and we gained distributed ray tracing. With this algorithm, as the name suggests, traces of rays are distributed over space and/or time. The result of this is that we can apply our ray tracing algorithm to the same origin spot in space, vary the trajectory of the ray each time, and take the average of all of these. Thus we can simulate imperfect reflectors. Similarly, we can vary the ray across a simulated lens to emulate depth of field blur, vary a light-checking trace's direction to various points the light to sample the shadow's penumbra, and even reduce aliasing by varying the starting position of the ray at the sub-pixel level.

However, there is still the fundamental part of tracing a ray; the ray must be compared with geometry to see if, and where, and with what it intersects. And this is not free. So if you cast multiple rays per pixel, which cast multiple rays over the radius of a lens, which cast multiple rays from an imperfect reflector, which sample across a light's surface for a penumbra, you end up with an astronomical number multiplied by a non-zero number, which is still astronomical even if the individual cost is minuscule.

1.2 Parallelism

Of course, such a bleak outlook discounts the various means and circumstances at the disposal of a contemporary implementation. For example, a spatial data structure such as an octree may vastly cut down on the effective number of intersection tests to make for each ray. And as an implicitly available alternative, superior or more specialized can be employed to reduce the time taken for a given number of computations. But in between hardware revisions, and in spite of rigid data structures that cannot be cooperatively treed, it is still possible divide and conquer with parallelism and more of the same hardware.

Ray tracing is what can be described as an *embarrassingly parallel* task, a task that is readily done in parallel with minimal concern for the pitfalls of parallelism. While ray intersections are being computed, geometry data structures are not altered. And each ray intersection leaves no data behind to experience race conditions with other rays originating from a different pixel landing on the same spot since the algorithm is recursive. But that does not make parallelism a "set it and forget it" solution, as it is still requires a solution to coordinate and manage the process safely.

2 PAST WORK

With such good theoretical compatibility between parallelism and ray tracing, it should not be surprising that much research into their combination is available. As an example, in [Reinhard and Jansen 1997], the nuances of division of labor in parallel ray tracing is explored, thus a distinction is made: *data parallel* ray tracing, where scene data is distributed across a data structure that the processors are also distributed over, allowing the algorithm to process a very large scene efficiently in regards to space at the expense of imbalanced load across the processors; *demand driven* ray tracing, where coherent rays are bundled together and prioritized by the processor originating these bundles, allowing intersection testing to be optimized and then cached for speed. Demand driven ray tracing scales much better, but necessarily requires data copying and communication of data across boundaries. A hybrid form can be used where the data parallel system is capable of performing the demand driven work, improving scaling compared to the very poorly-scaling data driven version, but plateaus when the processors are numerous enough to deplete the demand driven work at a faster rate.

Furthermore, in [Parker et al. 1999], Parker et al. describe in detail a ray tracer that can operate in either a synchronous mode, which behaves like a typical ray tracer and only updates the screen when all the pixels are traced, or an asynchronous mode where the screen is updated at fixed intervals with a static distribution of pixels to processor and the program simply reads from the buffer. The ray tracer in synchronous mode is described as processors receiving groups of rays sized to match the architecture, which are placed in a queue of descending size to balance reduction in synchronization overhead and efficiency losses from poor distribution of work. Accordingly, this may be classified as a demand based ray tracer.

Given the size of the data structures handled in homework 3, we determined a demand based ray tracer would not be constrained by memory limitations, even in the theoretical worst case where every processor had an entire copy of every data structure to itself. More specifically, if every Pixel object was copied to every process, since Pixels are compound POD types consisting of five Vec3fs which are each implemented as an array of three doubles, with the test machine's 8 bytes per double, then a 1920x1080 window would weigh approximately 250 megabytes per process. Given the specifications of the test system, consisting of a 12 physical, 24 virtual-core Ryzen 5900X with 32 GB of RAM, at design time this was considered an acceptable cost. In section HNGNGHL, we will discuss the differences in what we expected to result from the implementation compared with the actual result.

3 SYSTEM IMPLEMENTATION

First, in order to understand how the parallelism was implemented in the code, a brief overview of how the original homework 3, will follow in section 3.1. In sections 3.2 And 3.3, we will discuss the two different implementations of a demand demand based parallel ray tracer we implemented.

3.1 Original Homework 3

The key portions of homework 3 for this implementation are as follows: when the key to trigger the ray tracing animation to start or restart is pressed, the pixel buffers for both render buffers are cleared, along with the *refinement level* and the coordinate to processed for that refinement level. At a point in the near future, the looping Render function calls Animate, which calls DrawPixel up to 100 times, or until it returns false which occurs when it has refined every true pixel on its display. At each call, the RayTraceDrawPixel function will handle row rollover, column, and refinement level rollover for the refinement coordinates, then does a trace for the pixel's color, which is pushed to the currently active flip queue. Once either RayTraceDrawPixel returns 0 or 100 iterations pass, control flows back up out of Animate to the calling openGLRenderer, which calls allows the window to be drawn to 100 pixels at at time.

3.2 Mulithreaded Shadows

The first segment of the program we attempted to make parallel was calculating each light's contribution to a given traced point. That is to say that for each light in the scene beyond the first, a new thread will be spawned during the trace for the purposes of computing the amount of light contributed to an intersect point from that light. This was implemented using std::async and std:future. The rationale behind this was to make it possible to detect any issues in implementation before we had an expectation of what failure from algorithm would cause.

Within each call to RayTraceDrawPixel, in the shadows mode, a vector of std::futures are instantiated with calls

4 FUTURE WORK

There are a number of immediate low-hanging fruit one could finish with more time. For example, implementing SNell's equations. In addition, some time to get familiar with the PLL runtime to maintain more control over the parallel_for_each loop would undoubtedly prove useful. Perhaps add a layer of indirection over how the MPPU argument affects the Pixels vector size, such as a

scale factor calulated through the average time/pixel for the last K pixels or somesuch.

REFERENCES

Robert L. Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed Ray Tracing. *SIGGRAPH Comput. Graph.* 18, 3 (Jan. 1984), 137–145. https://doi.org/10.1145/964965.808590

Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. 1999. Interactive Ray Tracing. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics* (Atlanta, Georgia, USA) *(I3D '99)*. Association for Computing Machinery, New York, NY, USA, 119–126. https://doi.org/10.1145/300523.300537

Erik Reinhard and Frederik W. Jansen. 1997. Rendering large scenes using parallel ray tracing. *Parallel Comput.* 23, 7 (1997), 873–885. https://doi.org/10.1016/S0167-8191(97)00031-8 Parallel graphics and visualisation.

Turner Whitted. 1980. An Improved Illumination Model for Shaded Display. *Commun. ACM* 23, 6 (June 1980), 343–349. https://doi.org/10.1145/358876.358882