Time Restricted Parallel Ray Tracing

Stephen Kelley and Edward Levie

April 24, 2008

Introduction

Ray tracing is certainly one of the fundamental algorithms in the world of computer graphics. Given enough time, processing power, and highly detailed geometry, the algorithm can produce wonderfully realistic renderings. However, these requirements, especially the first two, can cause a number of headaches for individuals interested in creating renderings.

This paper focuses on including the dynamic of time while ray tracing. Time is a restriction on everything in the natural world, and ray tracing is no different. One could forsee many circumstances in which an individual would want to produce an image within a certain duration. Real-time ray tracing for video games or interactive systems is one such use. In a video game situation it is conceivable that gamers may want a rendering system in which they don't have to set their graphics rendering options to attain a specific frame rate; instead the ray tracer would achieve whatever quality the computer system is capable of. We do not frame our exploration within any sepcific domain; rather, we simply examine the issues raised when guaranteed delivery of a rendering is required by a given time.

There are two ways to view the inclusion of a time restriction in a ray tracing scheme. The first is to view the time limitation as a guideline or a goal to achieve over the average case (if rendering multiple frames consecutively). This level of restriction gives more leeway to the algorithm, as it may adapt to the input geometry and spend more time on more complex scenes while spending less on simpler scenes. The second method, and the one which we have adopted for this text, involves setting a hard cap on the time required to ray trace a scene regardless of its complexity. While certainly less general and adaptive than the previously described method, the method is much simpler. Striving for an average framerate introduces questions regarding the tradeoff between saving time for a later run or perfecting the tiniest of shadings on the most insignificant of objects in a scene.

Previous Work

Though we have chosen the simpler of the two methods, both allow us to explore a very important question. If an image cannot render to completion due to a time constraint, how can the closest estimate of the image be generated in the least amount of time?

The easiest, and one would imagine increasingly more common, approach would be to use a grid or a cluster to run the ray tracing on many processors in parallel. This has been an area of research for quite a long time [1, 2]. Ray tracing is a particularly good algorithm to implement in parallel because of its pixel to pixel independence. For this paper, we chose to use a pre-existing parallel ray tracing program called Tachyon located at [3].

Tachyon Overview

Tachyon is a parallel ray tracer which provides many powerful features which make it a capable rendering system. Parallelism in Tachyon is achieved through the use of message passing libraries OpenMP and MPI and through multithreading pthreads, the Posix thread library. It is able to run on a wide variety of machines ranging from sequential single-processor computers to large distributed memory supercomputers due to its comprehensive compiler and platform support. Included in the ray tracer is a spatial decomposition scheme which greatly increases rendering performance by automatically partitioning a scene. While not as robust as other ray tracing implementations, antialiasing is achieved through perturbations of primary rays and averaged using a simple box filter. Tachyon includes support for both Phong and Blinn shading and its native scene description language has facilities for texture mapping. The scene description language provides many geometric primitives, but for complex scenes Tachyon also supports AC3D and NFF files. For a full list of Tachyon's features and capabilities as well as tips for running it, see [4].

Such an array of features allows for flexibility in rendering a given scene on the computing resources available to the user. It is possible to fine-tune most rendering quality options when creating a scene. Options such as scene resolution, number of antialiasing rays, depth of recursion, and specular shading model can be chosen to provide a desired image quality. The parallel nature of Tachyon requires options for managing distributed computation. Settings governing threading and message passing configuration are available as command line arguments. In sum, Tachyon's feature set makes it a versatile and flexible parallel ray tracer from which to build time restricted parallel ray tracing. A sample rendering from tachyon is shown in figure 1.

Theory

Now, having increased our computational ability through parallelization, one might assume that we have conquered issues of time. This is not the case. Consider the images in figure 2. Here, both of them are rendered incompletely

Figure 1: Sample rendering using an unmodified version of Tachyon. This image spent 12 seconds being raytraced by 2 threads

due to the time limitation. However, it can be argued that the image on the right does a better job of representing the true image because of a more uniform selection of display points is shown. This brings us back to the issue of how to get the best representation of the actual image the fastest.

In [5], the authors describe a number of approaches based on keeping a priority queue of rays based on some user specified priority function. Before getting into the function, it would be useful to cover the mechanism by which they convert a traditionally recursive operation or raytracing into one which can be broken into many components.

Using the Phong illumination equation, we know that the illumination of a pixel in the image plane is going to be equal to the combination of local (ambient and light source illumination) and global (reflected and transmitted rays) contrbutions. The local illumination values can be calculated by sending out shadow rays to all lights in the scene. In order to separate the calculation of different rays' contributions to the final illumination of the pixel, they define a weight on each ray. If a primary, reflected, or transmitted ray is traced and hits an object, it will spawn a reflected and transmitted ray whose weight is equal to the weight of the parent ray times the specular or transmittive coefficient of the surface. Consider a ray which has reflected off 6 different surfaces before hitting a pure diffuse surface. The ray's contribution to the original pixel that it passed through will be the illumination of the diffuse surface times its weight, which would currently be the product of all 6 reflective surfaces' specular coefficients. In this way, rays can be traced in any order, since they track their own contribution to the original pixels in the image plane that they passed through.

The authors also make an observation about shadow rays. Without allowing shadow rays to be cast whenever the user specifies, the idea of a priority based scheme is quite limited. In order to get around this, when calculating the local illumination of a point, they initially assume that no occlusions exist. Shadow

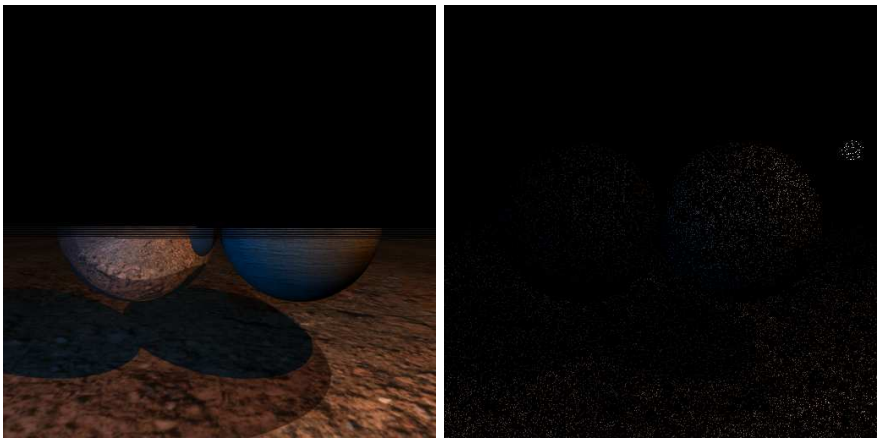

Figure 2: The image on the left is a result of running an unmodified version of Tachyon on 2 threads for 2 seconds. Note the systematic behavior of Tachyon by default. When time thresholds are small, much of the image is completely black. The image on the right shows the result of tracing with a random selection of initial pixels. Note that the general "sense" of the image is there. 2 spheres can be seen, and their shadowing can be made out.

rays are placed onto the queue with the weight of the ray which hit the point that they are testing for shadows. When the rays are popped from the queue, they calculate how shadowed they are and subtract this value times the current weight from the illumination of the pixel in the image plane.

In their paper, the authors test various priority mechanisms to determine which approach to employ to achieve the actual image the fastest. Some of the mechanisms were quite interesting. For instance, assign every object a priority and have any ray which hits the object inherit that priority. However, the best results were obtained with a fairly intuitive mechanism: selecting the ray which has the highest weight and therefore could affect the current image the most.

Implementation

For our implementation, the first task was to limit the execution of the code based on time. This was simple enough while code was loop and recursion based. We simply check a clock before tracing each ray, and if it is beyond the allotted time, all work stops. With the shift to a priority queue, time limitations remained easy. We simply popped rays off the queue until the limit was hit.

The priority queue implementation is located in pqueue.h and pqueue.c.

This code was found online at [6] which saved us quite a bit of work. The main thrust of the work was converting the full_shade() function into a series of functions that would handle reflected, transmitted, primary, and shadow rays as well as do phong shading on surfaces. With the shift from a recursive tracing scheme to a queue based scheme, we had to enhance various data structures throughout the program. For instance, each ray now had to track which pixel in the image plane it was to update. Any ray spawned then had to be passed the location of these pixels as well. We also had to modify the initial weights of the primary rays from 1, as stated in the paper, to $1/(aasamples+1)$. This allows antialiasing rays to contribute directly and immediately to the image buffer rather than the recursive approach of finding the illumination of the pixel with respect to each perturbed ray and then averaging them together. Without this modification the antialiasing rays would each contribute the full illumination, thus oversaturating the image.

Testing was accomplished by generating images using unmodified Tachyon and the corresponding image using our time restricted rendering. Doing this allowed us to pinpoint how our renderings were progressing and whether or not they were correct. Using a debugging tool was not realistic given the nature of the code. The reasons for this are elaborated on in Difficulties, Bugs, and Blunders. Testing our method was also performed by running the code on varying hardware configurations from a single-core Pentium 4 to an eight-core Xeon using multiple threading options from a single thread to 8 threads. This allowed us to verify our results under limited resources and under conditions where computing power was bountiful.

Results

Our results were a bit disappointing. Our modifications routinely performed worse than the recursive version of the program. Comparison output is shown in Figure 3. Both images were generated using 3 second limits, but the recursive version ended up finishing in a mere 1.7 seconds. Figure 4 also shows a side by side comparison of the two algorithms. These runs were allowed to run without regard for time in order to get a sense of how far behind our implementation actually was. The final running time for unmodified Tachyon was 5.7s while our version finished in 28.1s.

I would guess that this increase in speed has to do with our version not taking advantage of some of the short cuts in the software. Also, the priority queue implementation might be a little bit slow. A major problem that would keep this from being useful with a small number of threads is the memory required to store the required priority queue. Many times, our processes running sequentially allocated over 1Gb of RAM. Part of this is due to inherent memory requirements of keeping the queue, but part of it is also due to us not wanting to modify too much of the underlying structure of the program, and as a result having some pieces of information duplicated or stored for no reason. We found that modifying the Tachyon source was a delicate process; the parallelism introduces

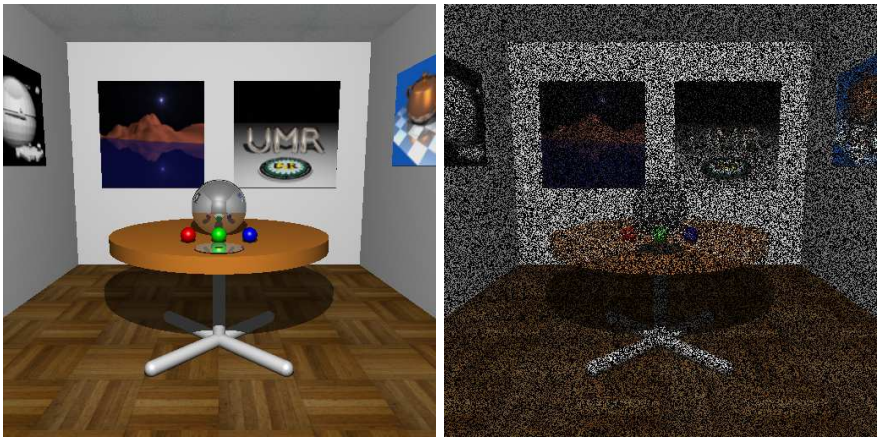

Figure 3: The image on the left is the result of running on an unmodified version of Tachyon on 2 threads for 3 seconds. The actual execution took only 1.79s. The image on the right is the result of running on our priority based version on 2 threads for 3 seconds.

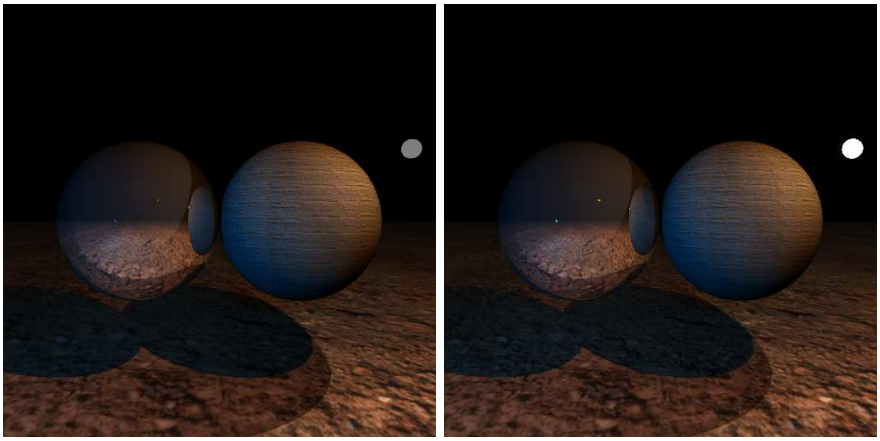

Figure 4: An unbounded run for each algorith with respect to time. The image on the left is from the unmodified version of Tachyon. It finished in 5.7s while the priority queue based version finished in 28.1s

sensitivies in memory management.

Difficulties, Bugs, Blunders

Overall, I'd say that the hardest part about this project was simply dealing with someone else's code. You can see from the results section, Tachyon is very fast. Unfortunately for us, this means that the code is very optimized and very hard to understand. We spent hours and hours poring over very strange errors in the code. Tachyon is written in C and the necessity of manual memory management, pointer manipulation, and simple data structures resulted in very low coding productivity.

By far the biggest time sink for us was tracking down a segmentation fault. Tachyon is actually compiled as a library and for performance uses heavy compiler optimizations. The combination of these factors makes debugging via gdb almost impossible. We ended up debugging via printf's scattered throughout the code. This introduced yet another set of complications since our time bounding caused the program to complete when much output had to be written. The bug ended up having to do with a bitwise AND used to check a flag storing the ray type. This issue highlights how difficult it was to work with the Tachyon source; the number of flags and low-level logic operations to manage them was dumbfounding.

In terms of bugs, generally speaking we advise against running with antialiasing off. It tends to produce inconsistent results that we were not quite pinpointed yet. We believe that we understand the cause of speckles in the shadow on the left of Figure 5 (they may be due to floating point errors when removing the illumination that should be occluded), but we were not able to determine why the table appears transparent in the right image.

Future Work

There are many possible avenues to extend all of this work. First, all of the code needs to be cleaned up. We might have been better off starting with a less optimized, simpler base library to modify, but using Tachyon gave us threading which can be extremely complicated to implement and debug. A very interesting avenue to explore would be adaptive antialiasing dependent on detail of features in the image plane. If a mechanism was developed to detect sharp shadow edges, it should be possible to cast more rays in those areas in order to create smoother gradients. The priority queue implementation heretofore described could aid in this process by prioritizing primary rays and "cleaning up" only once a sufficient representation of the image has already been produced.

Another interesting area to explore was attempted in this project but not fully implemented. The idea is another possibility for implementing time restrictions. Instead of a priority queue, each primary ray is given a start time and a deadline. The deadline for a ray would be:

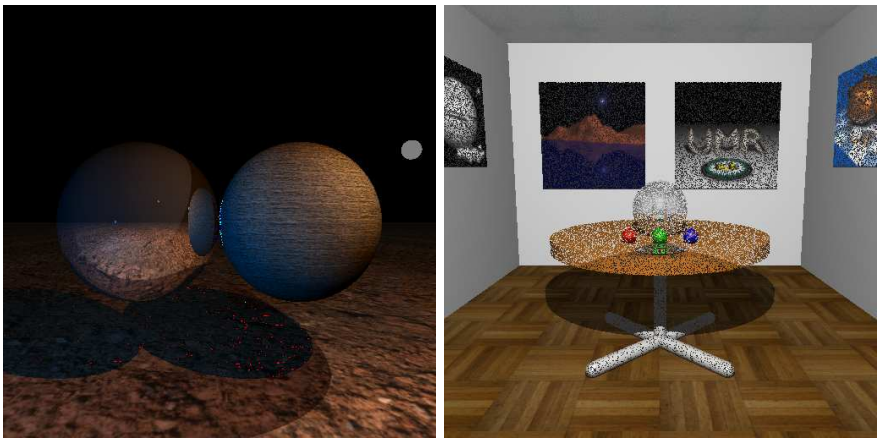

Figure 5: 2 blunders which only appear when antialiasing is turned off

```
ray.start_time + (1/FRAME_RATE)/(scene.h_res * scene.v_res)
```

Basically each primary ray gets an equal fraction of the time restriction for rendering the ray. As soon as the deadline is reached the ray would be stopped and most likely set to have hit the background. The complications of implementing this in Tachyon is that there are many points at which rays are created and the function call hierrarchy for a given ray is complex and heterogenous. Primary rays follow a very different path than reflective rays, which are different from shadow rays. Admittedly this approach would have its own set of complications. The quality of the rendering in this case would not be optimal. Many rays in a scene never intersect any object but these would be given the same amount of time as a reflective ray traversing a potentially deep recursive path. The result of this would be that image quality would be lower if the rendering completed without reaching the time limit; many rays would have been stopped before necessary. An adaptive method to vary deadlines would be applicable here.

Conclusions

We have described our attempts at extending a parallel ray tracer, Tachyon, with the ability to perform time restricted rendering. Tachyon presented many challenges to implementation of our ideas. We were however able to incorporate a priority queue scheme to limit the amount of time spent rendering an image. Overall we both learned a great deal about the intricacies of parallel ray trac-

ing and the concerns that must be made when converting a complex recursive process to support truncated processing based on time.

We did not keep close watch on the amount of time we spent coding though we probably should have. A very rough estimate of the amount of coding time would be 30 to 40 hours. As discussed previously much of our time was spent not coding but simply elucidating the complexities of the existing Tachyon codebase. Stephen found and incorporated the priority queue initially. Ed extended the data structures in use to allow for immediate updating of the image buffer. Stephen and Ed both spent many hours tracking down a segmentation fault. Ed rendered many of the scenes early on in order to find some reasonable test data. Stephen wrote the first revision of the project report. Ed edited and added content to the report prior to its finalization. Stephen generated the example images in the report. Stephen and Ed found articles about previous work in this space. Stephen heroically stayed up over 24 hours (not all of it coding) to make shadows work properly!

References

[1] Muntean, T., Waille, P.: A massively parallel approach for the design of a raytracing oriented architecture. (1991) 41–51

[2] DeMarle, D.E., Parker, S., Hartner, M., Gribble, C., Hansen, C.: Distributed interactive ray tracing for large volume visualization. In: PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics, Washington, DC, USA, IEEE Computer Society (2003) 12

[3] (http://jedi.ks.uiuc.edu/ johns/raytracer/)

[4] (http://jedi.ks.uiuc.edu/ johns/raytracer/papers/tachyon.pdf)

[5] Yagel, R., Meeker, J.: Priority-driven ray tracing. The Journal of Visualization and Computer Animation **8** (1997) 17–32

[6] (http://www.sbhatnagar.com/SourceCode/pqueue.html)