Ray Tracing in Parallel

Jordan Sorensen

18.337 Final Project Report

Introduction

The computer graphics community has developed a variety of techniques for simulating light transfer in order to generate images of imagined three dimensional worlds. From polygonal techniques capable of running in real time to physics-based methods that can take minutes or hours to render a single frame, these techniques run the gamut from fast but low quality to slow and photorealistic. The recent explosion of parallel computing hardware has given ray tracing – a high quality rendering technique which can be easily parallelized – a leg up. After implementing a completely serial ray tracer for two computer graphics classes at MIT, I decided that I'd like to bring my ray tracer up to speed by making it parallel.

What is a ray tracer?

A ray tracer is a program which tracks the path light takes through a scene in order to generate an image of that scene. It assumes that light travels in a straight line – a ray – and that it changes direction when it strikes an object. It simulates light travel by creating a ray which matches the direction that light is traveling in and testing all objects in the scene to see if they intersect that ray. It then takes the first intersection in the scene, finds its position, and begins a new ray from this position in a new, "bounced" direction to simulate light bouncing off this object. A number of extensions are possible to this scheme – refraction (for transparent materials like glass and water) can be achieved by casting rays into and through objects, and more complex surface materials can be simulated by bouncing multiple rays off each object. The basic idea, however, is simple – generate many rays of light and find their intersection with objects in the scene.

Why is it slow?

Each time a ray is cast, we must test for its intersection with each object in the scene. With a large or complex scene, this can require a lot of processing power. On top of this, however, many many rays of light must be simulated – at least as many as the number of pixels in the generated image (which is usually on the order of tens of thousands). These computations add up quickly, resulting in rendering times of minutes or hours.

How is it parallelized?

Luckily, ray tracing is an embarrassingly parallel computation. All computations work on the same set of inputs – a ray and the scene description. The scene description is a static input – it stays fixed throughout the computation – and the rays are generated as the program runs. Luckily for us, no intercommunication is necessary between computations – the data remains completely separated until the very end, when results are compiled into an output image.

Project Goals

My development notebook contains a dual core processor. However, while running my ray tracer in serial, only one of these cores is utilized. I wanted to make sure that my parallel ray tracer would be able to use both cores. In addition, the computer market is now incredibly parallel, and there's a good chance that computers I purchased in the future would contain even more processors. As

a result, I decided that my parallel ray tracer should be able to handle an arbitrary number of processors so that it would scale with this increasing computer power.

In addition, I often imagined my ray tracer being used for even bigger projects. For example, it might be used to generate short animated movie clips. This would require thousands of ray traced images (about thirty images for each second of video) and would take an enormous amount of time on a single computer, even if that computer had four or eight cores. To render these images, I might use a cluster of computers, and send work computation that needed completed to each one. To fill these needs, my parallel ray tracer would need the ability to be parallelized across a network as well. I could even imagine trying to harness other types of computing power, such as GPUs. In order to meet all these requirements – scale to an arbitrary number of processors on a single computer, use processors scattered across a network, and provide the ability to "plug in" a variety of other computing power, my design needed to be extensible.

Design

With these goals in mind, I drew out a design that I felt would be able to accommodate these requirements. In my design, a single thread was the "Master" thread. This thread didn't do any work on the actual scene, it simply delegated work to others. It maintained a list of work that needed to be done (a list of rays which needed to be traced) as well as a list of computing power sources. These computing power sources could be any of the things listed above – local processors, computers across a network, a GPU – as long as they met the requirements outlined in a specification. This specification said that the computing power source must provide some function which took in a ray and returned the result of casting that ray into the scene. The specification did not, however, say how this function must work. It might start up another thread on the same computer. It might connect to an external server and send the ray to the server to compute. By specifying only the interface between the computing power source and the Master thread and not the implementation of this specification, I was able to make my design extensible. Anything that I could write code to meet this specification with could be plugged into my program as a computing power source.

I mentioned above that the specification stated that each computing power source must provide a function which takes a ray and returns a result. But what kind of result would it return? It couldn't be the actual result of the computation – say a color, or exact number – because that would mean that the computing power source must compute the result before returning control to the Master thread, essentially serializing my design. Instead, I created a "DelayedResult" object. This object had two properties. One, IsResultAvailable? told whether or not the result had yet been calculated. The other, GetResult, got the result once it had been calculated. Using this scheme, a computing power source could allow the Master thread to continue execution before it finished its computation, but the Master thread still had a handle- a way to get a hold of the result later one once the computation was complete.

The embarrassingly parallel nature of ray tracing also allowed me to make a few optimizations. Most importantly, I decided that local processors should all share the same copy of a scene. Since they all had access to the same memory, they could all use the same copy in order to save data transfer time. Because all each processor did was read information from the scene – they never modified the scene as a result of their calculation – they did not need an independent copy just to avoid stepping on each other's toes.

Results

With this design, I could begin actually building the framework on which my ray tracer would operate. I began by writing the code for the Master thread which would handle work management. This thread maintained a queue of work which needed to be processed, as well as a list of available processors. It also had a set (but adjustable) buffer size. This buffer size set the number of work items which a processor should be allowed to have unfinished at any given time. The Master thread

continuously looped, giving work to all processors which had fewer outstanding computations than the buffer size. By keeping a large enough buffer, we guarantee that no computing power source ever runs out of work to do, guaranteeing good load balancing. If we keep too large a buffer, however, load balancing will be off as we near the end of the computation, since the Master might run out of work to give one processor because it has given another too much (in order to fill its large buffer).

In my implementation, when the Master thread gives a work item to a computing power source, it doesn't remove that item from the queue entirely. Instead, it pushes it to the end of the queue. My hope was that this would make my ray tracer more robust to a variety of computational power sources. For example, a computer across a network might go down or become temporarily disconnected from the network. Using this queue model, any work which was previously given to a computing power source will eventually be reassigned to another source if that one never returns a result.

After I finished implementation of this Master thread, I moved on to begin implementing things which would meet the computing power source specification. First, I implemented a plug in for local processors. This turned out to be a much more difficult task than I had anticipated. I was using a library (pthreads) in order to put each computing power source on a different thread (which would allow them to be on different processors) which I was unfamiliar with, which added learning time to my development time. In addition, after finally writing the code, a mysterious error began cropping up. It didn't appear every time the program was run, but a high percentage of the time, and it would cause the program to crash. Finally, after many days of searching and debugging, I found the source of the error. The important assumption I had made regarding scenes – that they were read-only – was actually slightly off. Although the high level concept of a scene didn't change as the computation took place, the scene object was implemented using a low level piece of software which did change state as it was accessed (it contained objects which used reference counted pointers, a programming construct used to make sure memory is allocated and deleted when it needs to be). Two different threads working on the same scene would both try to modify this low level object at the same time and cause the program to crash. As a result of this discovery, I had to create a copy of the scene for each processor.

Finally, I had multiple processors up and running. I added in a timer to see how much parallelization had helped and ran it on a simple scene. Much to my dismay, the parallelized version actually ran *slower* than the serial version. After using a profiling tool to collect data, I finally determined the source of the problem – the code that managed work spent more time handing out work, checking to see what work was done, etc than parallelization saved. This made some sense – I had used as input a very simple scene, and with such a simple scene, the cost of dividing up work was actually more than the cost of the work itself. I tried again with a much more complex scene, and got slightly better results – the parallelized version ran ~25% faster.

Analysis

Still, after doubling the processing power accessible to my application, 25% is not a great increase. Some of this cost came from my design – since I had tried to make my Master thread robust to many different potential kinds of processing power, it was not lightweight, and took up a decent amount of available processing power without achieving any actual work. In addition, since I had to copy the scene file for each processor, there was additional data flow cost for each processor.

On the bright side, I feel my design met my goal of extensibility. I was not able to implement the user of computational power across a network – after dealing with so many unexpected issues when implementing parallelization across local processors, I simply didn't have time to do this work. However, I feel some degree of certainty that my design would extend to this type of computational power relatively easily. In addition, I was able to implement parallelization across local processors, and my current program will easily scale to any number of processors on a single machine.

I feel my experience parallelizing this program has reminded me that it's much more effective to

design a system in parallel from the ground up than to take a serial system and parallelize it. For example, it was very convenient to assume that my scene objects were read only, but it turned out that at the lowest levels, they were not. If I had designed these constructs from the ground up with the parallelization in mind, I could have strictly enforced the fact that they were read only and not needed to make multiple scene copies. I see many programmers today (this project included) taking the "parallelize" approach – they take a serial application and try to make it parallel, rather than designing systems in a parallel way. For example, a lot of current research in compilers regards performing a complex data flow analysis on a serial program to determine what can be done in parallel. However, this parallelize approach seems like it can't succeed beyond the transition phase – it may be a good way to take existing programs and make them fit into a new system, but when the new system is the norm, successful programs will be those developed with this new paradigm in mind.

Future Work

Although I hope to eventually finish the goals I originally set out for this project (including networked parallelization, and GPU use) I'd first like to start my ray tracer over again, this time designing it knowing that it will be executed in parallel. I hope that this will help me achieve a nearly 100% increase in speed (corresponding to 100% increase in available processor power) when adding a second processor, as opposed to the 25% increase I currently experience.

In the interest of decreasing the ratio between actual work and "maintenance" overhead, I might also consider changing the level at which I am parallelizing. For example, instead of giving each computing power source a ray to trace, I might give it an entire block of pixels to trace rays through. This way, the computing power source has a larger amount of work to do before it must report its results back or get more work from the Master.