Performance Engineering of Software Systems November 14, 2010
Massachusetts Institute of Technology 6.172
Professors Saman Amarasinghe and Charles E. Leiserson Handout 15

Ray Tracing Performance Derby

Final Project

1 Introduction

In this final assignment, you will put the skills you have learned so far to the test. Each group will start with the same program—a ray tracer. This program renders two scenes in a manner similar to the one described in the ray tracing primer. The code you are being given is complete and correct; however, it is rather inefficient. Your job is to apply the techniques you learned from this course to speed up the program.

This project simulates the situations you will encounter in real life. After designing and implementing a fairly sizeable program, you find that it doesn't run fast enough. At this point, you have to figure out what is slow, and decide how to fix the performance issue. We've left in some low-hanging fruit, but you'll find the need to make design changes to the raytracer for substantial performance improvement.

2 Background

2.1 Raytracing theory

The provided program is a photon mapping ray tracer that can render scenes composed of basic primitives such as spheres, cubes, and displaced surfaces. The ray tracer is coded to render one of two hard-coded scenes: a red and blue box with two spheres, and a blue and green box half-filled with water.

The ray tracer starts by tracing rays in the opposite direction of traveling light, starting at a camera focal point and tracing rays through each pixel of an image plane outward into a scene. When a ray intersects with a diffuse surface (a matte surface such as a wall), the ray tracer computes the amount of light being emitted by the diffuse surface at that location, which in turn is used to compute the color of the associated pixel in the image. If a ray interacts with a reflecting and/or refracting surface, new rays are created to continue tracing the path that light would have taken on its way to the camera.

In order to obtain a realistic effect, this ray tracer models three different sources of illumination to compute the amount of light that falls on the diffuse surface at the point of a ray intersection: direct, caustic, and global. The following images display the illumination-type breakdown for the two scenes rendered by the ray tracer.

As the name suggests, direct illumination is light that comes directly from a light source (i.e. the surface is in the line of sight of a light source). It is cheap to compute because it can be performed in the reverse direction by checking whether a ray originating from a surface location has an unobstructed view of a light source.

Caustic illumination occurs when light is focused by an object through reflection and/or refraction. Unfortunately, it cannot be computed in the reverse direction. This renderer employs a technique called photon mapping to compute the caustic effect. At the start of the rendering process, the renderer emits thousands of photons from the light sources and tracks their loss of energy as they bounce around the scene. The information is used to create an irradiance map (implemented as a kd-tree) that can be queried to compute the light being emitted by a diffuse surface at any location. By storing only photons that traveled directly through a reflective/refractive object to the diffuse surface, the photon map can be used to determine the component of radiation at any point along a surface that resulted from caustic illumination.

Global illumination is light that has repeatedly reflected off of many diffuse surfaces until a fixed point is reached. It is the most expensive component to calculate. Like caustic illumination, it is calculated with the help of a photon map. However, unlike caustic illumination, the global illumination component is not computed directly from the photon map. Doing so would result in a blotchy rendering (shown below) because of the finite number of photons stored in the photon map. Instead, the renderer adds an extra level of indirection, sending out an additional 312 rays for every visible point on a diffuse surface to determine the amount of global illumination shining onto that point. These rays in turn interact with the scene and eventually intersect with other diffuse surfaces at which point the photon map can be used to determine the amount of light being emitted by those surfaces. This extra level of indirection is very expensive. Fortunately, the global illumination component does not vary much along a surface and can therefore often be computed via interpolation from neighboring points in the scene. The renderer uses an irradiance cache (or icache) to cache previously computed irradiances that can be used for interpolation. Each entry in the cache includes an r0 parameter that is computed from the distances of the rays used to compute the irradiance. This parameter is used to determine the size of the region for which the cached irradiance can be used to accurately compute interpolated irradiance values.

2.2 Compiling and Running

You can use `make` to compile the code, which generates `raytracer` and `all_tests` binaries. It accepts the following arguments (the default value being shown):

- `-s 1`: Render scene 1. Currently, only scenes 1 and 2 are defined
- `-n 600`: Renders the image at resolution 600*x*600 pixels. Set this size smaller to render a faster preview.
- `-o output.bmp`: Outputs the image to the file called `output.bmp`
- `-d on`: Turn on direct illumination. (Hint: `off` is the opposite of `on`)
- `-g on`: Turn on global illumination
- `-c on`: Turn on caustics

We've also provided you with several different build configurations which you should find useful. For each mode, you can enable it by adding `MODE=1` to your make command line. The Makefile is written so that if you change modes, it will properly rebuild everything. Note that you can mix and match these modes.

- `DEBUG`: Turns off optimization and enables assertions.
- `PROFILE`: Disables frame pointer omission. If you want to generate a callgraph with `perf` or `gprof`, you will need to use this mode, because both tools take stack snapshots by walking the stack using the frame pointer. This mode implies `NOCILK`.

- `NOCILK`: Disables the cilk runtime. Use this build mode for debugging, profiling, and running valgrind.
- `GPROF`: Adds `-pg` to `CFLAGS` and `LDFLAGS`, so you can use `gprof`.

3 Administrivia and Deliverables

Compared to previous projects, this project is loosely structured in order to cut the overhead and give you the most time to work on this project.

3.1 Groups

You will be expected to work in groups of **two or three**. You may work with anybody in the class, including previous project partners. We suggest you choose and form your group as soon as possible. Run the project-setup script with the names of your partners like so:

```
$ project-setup -p final partner1 partner2
```

You can omit partner2 if you want to be in a group of 2.

3.2 Preliminary Report

Halfway into the project, we expect you to produce a preliminary report on your optimization plans and progress. At this point, you should have a strong understanding of the ray tracer's structure and performance characteristics. You should also have made some progress with optimizations, but most importantly, you should have a plan for what optimizations you are going to implement. There are many potential changes you could make, but you should focus on the areas that will give you the best return on time spent. This report will be submitted on Stellar, and you will be meeting with your MITPOSSE mentor. This is a valuable opportunity to receive feedback both from the course staff and an industry expert regarding your plans for the final project. To summarize, your report should contain:

- Profiling data on the reference implementation
- The changes you've implemented so far, and their impact (supported by profiling measurements)
- Optimizations you plan to make (and how you prioritize them), supported by profiling data. You should also estimate the impact of each optimization based on your profiling.
- A work breakdown of how your team plans on dividing the work. This should convince the course staff that your group members are performing equal work, and everyone has a sufficient amount of work worthy of a final project for 6.172.

3.3 Final Turn-in, Derby

There is no beta submission. Past the preliminary report, the next and final deadline is to deliver your code by the beginning of the final class. We will be running all of your projects and displaying the results **live**, so come and see how you and your peers did!

3.4 Final Report

To help us with grading your submission, please submit a final project report that briefly outlines:

- The optimizations in your submission.
- Optimizations that you tried, but didn't work.
- The work breakdown within your group.
- Any extra information you think would be helpful for us in assigning a grade.

This report can be turned in by 11:59PM on the last day of class.

4 Additional Information

4.1 Starting Out

The raytracer is a lot of code – about 5000 lines in fact. You will have to decide how to use your limited time to achieve the best results. Here's some suggestions from us about how to begin:

- Make sure you are comfortable with the concepts behind how a raytracer works. Without this background, it will be difficult to think of substantial optimizations or troubleshoot unexpected artifacts in your rendering.
- Make sure you understand how the code is organized. While it's probably a waste of time to read through every line of code before beginning, make sure you understand the big picture components of the code and how they interact.
- Use the profiling tools you've learned in this class to identify hotspots. This should reveal all the low-hanging fruit.
- Never lose sight of the big picture. Once you find one hotspot, it's easy to get sucked into optimizing it to death. You can make a career out of optimizing, for example, the trigonometric functions. Recognize when you've made sufficient progress and move on!
- Minimize time spent waiting for computation results. Try to:
  - Cache the output of important images. We will be providing you a 600x600 reference image for the two provided scenes, so don't waste time running the reference raytracer to make these.
  - Render small images – there's no need to render 600x600 images for everything. Smaller images are faster to render and equally useful for profiling, debugging, and checking correctness. The reference implementation can render a 10x10 image of scene 1 in seconds.
  - For scene 2, the best way to get this to run in a reasonable amount of time is to edit `config.h` to reduce the number of triangles in the water. Don't forget to set it back to the default.
- Make incremental changes. Avoid structuring your changes such that you cannot compile/run your code for extended periods of time. Performance surprises tend to be unpleasant in nature.

- Parallelize! Raytracing contains a lot of parallelism, so on the cloud systems there's the potential for a massive 12*x* speedup. As you familiarize yourself with the code, keep in mind the ideal locations for parallelism. Introduce parallelism earlier, not later.
- Keep an open mind. When looking at pre-existing code, it's easy to be lured into following the footsteps of the original authors. You should ask yourself whether the data structures and algorithms chosen are appropriate – is there a faster (or more parallelizable) method?

4.2 Rules and Fine Print

We will be setting some ground rules:

- You may NOT change parameters in the config.h file, or otherwise ignore performance-sensitive parameters in your implementation!
- You may not perform optimizations specific to the distributed scenes. Your raytracer should perform well at rendering other scenes. We will be running your raytracer against a secret, new scene that will be revealed some time shortly before the derby to give you an opportunity to correct any bugs it may reveal.
- You may not change the behavior of your program based on execution-time parameters such as the size of the image, name of scene being rendered, time, date, user running the binary, etc.
- You may not rely on other people's code or ideas for any significant functionality in your submission. Reference materials and borrowed code snippets should be cited in your final writeup.
- Multiprocessing should be achieved through the Cilk++ runtime. You may not, for example, use a distributed computing package to source external computational resources.

When in doubt, don't hesitate to ask the course staff for clarification.