Final Project

In this project you will put the skills you have learned in this class to the test. You will be given a fairly compute-intensive ray tracing application. Your job is to improve its rendering performance by both parallelizing the code, and identifying bottlenecks and optimizing them.

Introduction

Snailspeed has recently acquired TurtleSoftware, a smaller competing software company. The acquisition was motivated by factors unrelated to Alyssa's projects. However, Alyssa's boss has learned that TurtleSoftware previously developed a simple ray tracing engine that can create realistic images of simple scenes. He would like to use the renderer to create an animation of a room of rippling water for his group's mobile phone screen saver app. Unfortunately, the renderer is excruciatingly slow, and thus cannot be practically used to render an animation. Your job is to help Alyssa improve the performance of the renderer so that it can be used for animation.

Getting Started

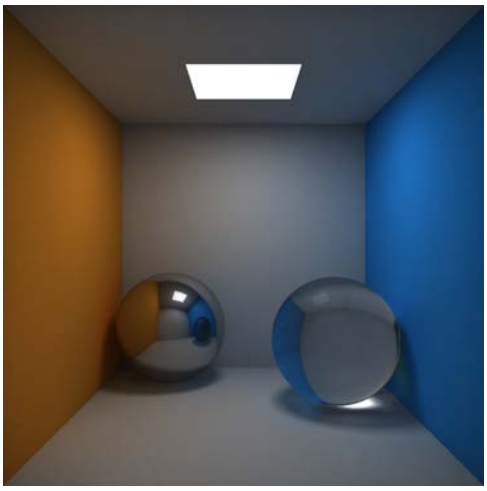
You will be working in groups of two or three. Please form a group by the next class and email the TAs. Students that do not form a group on their own will be assigned partners.

As before, you will be using Git. Use the following command to clone the git repository that has been set up for your group for the assignment:

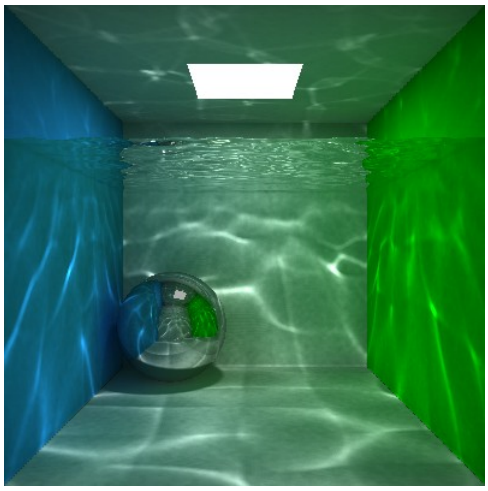
```
git clone \
/afs/csail.mit.edu/proj/courses/6.197/projects/project6/<group_name> \
~/project6
```

About the Renderer

TurtleSoftware's renderer 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:



Scene 1

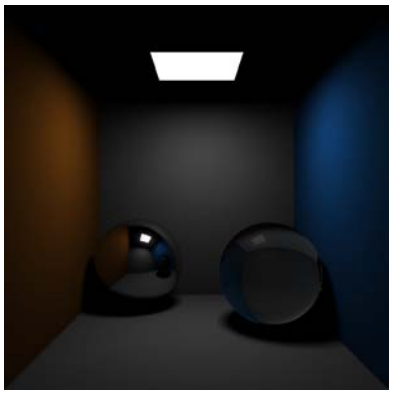


Scene 2

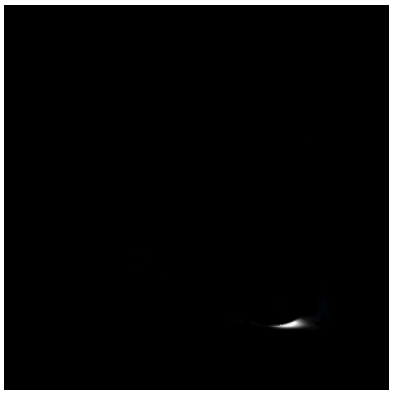
The ray tracer starts by tracing rays in the opposite direction of traveling light, starting at a camera focal point and tracing rays through each pixel of an image plane outward into a scene. When a ray intersects with a diffuse surface (a matte surface such as a wall), the ray tracer computes the amount 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.

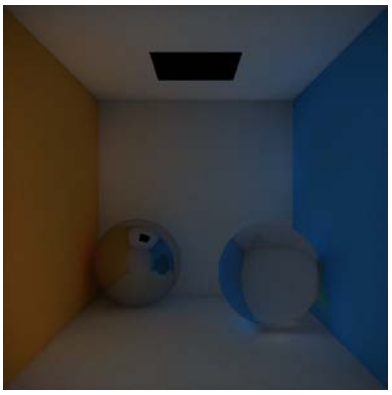
Scene 1



Direct Illumination

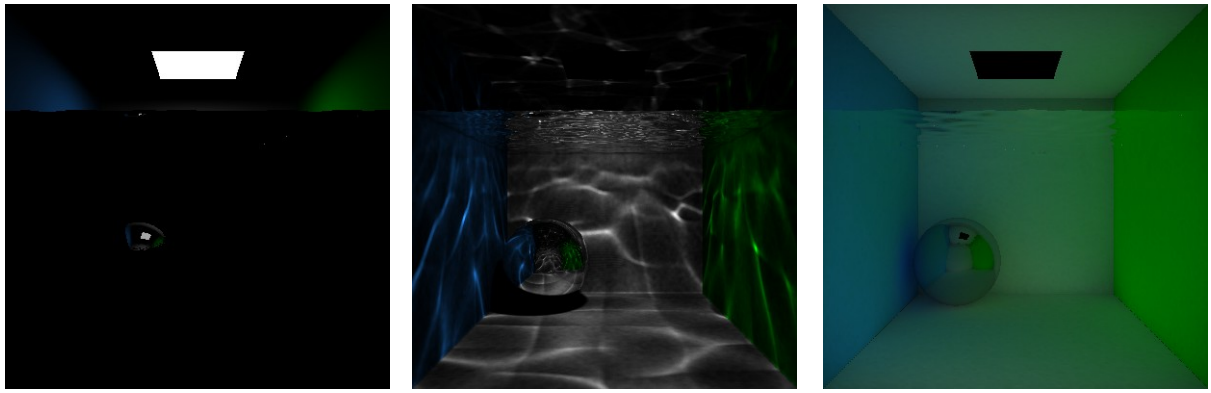


Caustics



Global Illumination

Scene 2



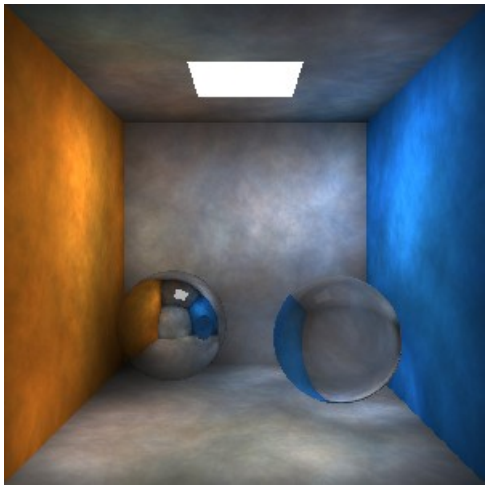
Direct Illumination | Caustics | Global Illumination

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.

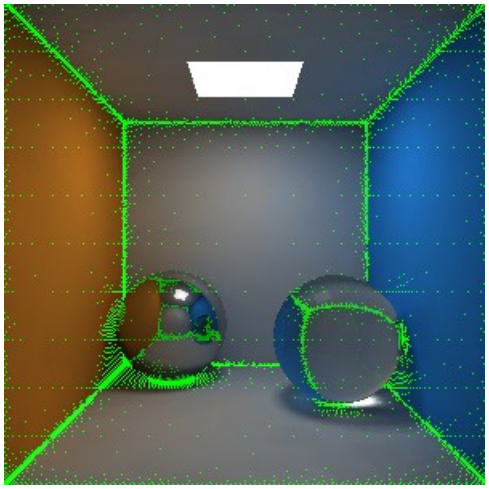
Caustic illumination occurs when light is focused by an object through reflection and/or refraction. Unfortunately, it cannot be computed in the reverse direction. This renderer employs a technique called photon mapping to compute the caustic effect. At the start of the rendering process, the renderer 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 r_0 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.



Blotchy rendering caused by visualizing the photon map directly



Location of cached irradiances in the icache. The rest of the points had their global illumination component interpolated.

Usage

Run `make` to compile the ray tracer. This will generate a raytracer executable that creates ray traced images.

The `raytracer` executable accepts the following command line arguments:

- `-s <1|2>`: Selects which scene to ray trace. Only two scenes are presently available. [default: 1]
- `-n <N>`: Sets output image size to be NxN. [default: 600]
- `-o <filename.bmp>`: Output image filename. [default: output.bmp]
- `-d <on|off>`: Toggles direct illumination. [default: on]
- `-g <on|off>`: Toggles global illumination. [default: on]
- `-c <on|off>`: Toggles rendering of caustics. [default: on]

Starting Out

Before beginning to optimize the ray tracer, you will need some intuition for how the ray tracer works. Go through the source code for the ray tracer and make sure that you have at least a high-level understanding for what is going on.

Profile the ray tracer in VTune and identify hot spots that are good candidates for optimization. To determine the effect of each type of illumination, try profiling the ray tracer with only one of the three types of illumination turned on. You may find VTune's Call Graph analysis to be more helpful than its Sampling analysis.

You are expected to submit a group progress report halfway into the project. The report should convey a strong understanding of the ray tracer, include profiling results, and discuss any progress you have made in making the code run faster.

Useful Tips:

- Start by optimizing Scene 1. Scene 2 will take significantly longer to render so there is no point starting with it until you have reduced the render time of Scene 1.
- Render small images. Waiting for a 600x600 image can needlessly take a lot of time if you just want to check that something still looks correct.
- Optimize each of the three illumination types separately. Your render times will be significantly shorter and your profiling results more informative.
- Parallelize the code earlier rather than later. Ray tracing contains lots of parallelism so you should easily be able to obtain a 7x or more speed up by parallelizing the code. This is a big speedup, so the earlier you do this, the less time you will spend waiting for the renderer when working on other optimizations. Once again, focusing on one illumination type at a time decreases the number of data races that you will encounter and have to solve at any one time.

  Hint: You will require locks to protect certain data structures from race conditions. Since cilk_locks are not available in the Linux version, use pthread mutexes.

Possible Optimizations

Here is a list of optimization ideas to get you going:

- Pre-compute commonly executed functions. Some parts of the code do not require an exact return values for certain functions. Approximations based on pre-computed values are often good enough.
- Search for short commonly used methods and move them into the header files so they can be inlined.

- Improve the ICache class. Currently, the irradiance samples are stored in a linked list. Consider changing this data structure to something more efficient.

  Hint: An octtree (similar to quadtree but in 3D) could be a good data structure for this. However, you have to pay special attention to the fact that each entry in the cache is valid for different ranges (depending on the value of r_0). You may need to employ a similar trick to the one you used in Project 4.

This list is not complete. You should explore the application code and profile it repeatedly to find new bottlenecks.

Rules

- You may not modify any of the parameters in the config.h file. We will be writing over it with the original when we evaluate your work.
- You may not perform any optimizations that would prevent the ray tracer from rendering other scenes.
- You are also not allowed to change the behavior of the code based on the command line arguments. Your optimizations should be as general as possible.

Evaluation

Your grade will be calculated as follows:

- Correctness (15 points). To receive full points your code should:
  - Render images that appear visually similar to the images rendered by the unoptimized staff-provided code. We will be inspecting 600x600 images of both scenes.
  - Contain no data races for a 100x100 rendering.

  You will receive partial credit for each of these requirements.
- Performance (35 points). Your performance grade will be computed by comparing the performance of your submission to the performance of other submissions in the class.

  Note that we will be executing your submissions on 16-core machines. You only have access to 8-core machines. Avoid making optimizations that are specifically tailored for the 8-core machines. Your goal is to write code that scales beyond what is immediately available to you.
- Style (10 points). Your code should be decomposed into functions and classes and should use as few global variables as possible. It should have plenty of comments throughout, describing your optimizations in low-level detail.

- Progress Report (10 points). The progress report should be a brief 3-page writeup. It should include the following:
  - Describe any observations you have made while profiling the application.
  - Describe the optimizations you have implemented.
  - List the performance improvements you see over the original code.
  - Outline additional optimizations that you plan on exploring.
  - Provide a brief breakdown of who did what work.
- Final Writeup (30 points). Your final writeup will be a group submission that discusses the work you performed. Your final writeup should include the following:
  - Detailed discussion of experimentation and all optimizations performed.
  - Description of parallelization strategy and implementation.
  - Justification of the choices you made.
  - Discussion of what you decided to abandon and why.
  - A breakdown of who did what work.

Bonus

For an extra 10% on the project, generate a 300x300 pixel 20-second animation of the 2nd scene where the surface of the water is in motion. When rendered with caustics, the moving water surface should create some interesting lighting effects in the scene.

To generate the 20-second animation, use your ray tracer to generate a sequence of frames where the water surface moves slightly between each frame. Label your frames numerically (i.e. img1.bmp, img2.bmp, img3.bmp, etc). These frames can be combined into a single animation using the following command:

```
ffmpeg -f image2 -r 15 -i img%d.jpg video.avi
```

Note that the above command will output a video that plays at 15 fps. To generate a 20-second animation, you will need 300 separate frames. You do not want to be generating these 300 frames by hand. Modify the ray tracer and write shell scripts as necessary to automate the process.

One source of difficulty is smoothly animating the surface of the water. If the surface of the water does not move smoothly, then the resulting animation will seem jerky

and unrealistic. For example, you cannot simply generate a random water surface for each frame as the resulting animation would seem like a jumbled mess.

To create a smooth animation, you should adjust the `time` variable defined in the `DisplacedSurface.makeSurface()` function. This variable is used as a third dimension to the 3d perlin noise generator and can be used to generates a water surface that changes smoothly with time. You may need to experiment with different period sizes to find one that yields a smooth animation.

Finally, you are not allowed to render the animation using more than one cagnode computer. The machines are shared with graduate students so please be respectful of the resource.

6.172 Performance Engineering of Software Systems

Fall 2009