Implementation

Contents

1 Data Structures and Algorithms

1.1 Caustics

I achieved caustics via the photon mapping method described in detail by Jensen et al. in [1]. The method requires a forward ray tracing preprocessing step during which light-surface interactions are recorded. It also mandates modifications to lighting computations during the normal rendering step. As Jensen recommends, I used the k-D tree data structure for performance reasons. Also for performance reasons, I implemented projection maps for the preprocessing step.

Code related to the preprocessing step, radiance estimates, and projection maps is in `photonmap.[ch]pp`. My k-D tree implementation is in `kdtree.hpp`. Forward ray tracing code is `rt.[ch]pp`. In particular, see the `RayTracer::raytrace_russian` method.

1.1.1 Forward Ray Tracing

The photon mapping preprocessing step entails shooting a set number of rays ('photons') from each light in the scene and recording the interactions of the photons with diffuse surfaces. It is a Monte Carlo algorithm; photons are shot in random directions, and their subsequent behaviour is randomly decided (in the Russian Roulette style briefly discussed in class). The intensity of each photon must be chosen based on several factors, such as the user-specified unit distance, the number of photons to be shot, and the fullness of the projection map.

Determining the right intensity to assign to each photon was very difficult, and required me to fiddle with parameters and add new ones until it worked well in the general case. I compute the energy of each photon as

$$E = \frac{4\pi d_u^2 \cdot C_{\text{light}}}{f_{\text{light}} P}$$

where d_u is the user-specified unit distance (which depends on the scale of the scene); C_{light} is the intensity (or colour) of the light; f_{light} is the r^2 falloff coefficient of the light; and P is the user-specified number of photons to be shot. Note that $4\pi d_u^2$ is the area of a unit sphere around the light. Note also that if we use projection maps, then we must reduce the number of photons shot based on the occupied area of the projection map.

Instead of generating multiple recursive rays in the reflective and refractive directions at every specular event – as is done in backward ray tracing – we choose to have photons perform diffuse or specular interactions – or, alternatively, be absorbed – based on the attenuation that would be applied to the light in each case. For example, if a material has diffuse colour $(0.3, 0.3, 0.3)$ and specular colour $(0.1, 0.1, 0.1)$, then

$$P(\text{diffuse reflection}) = 0.9/3 = 0.3$$
$$P(\text{specular interaction}) = 0.3/3 = 0.1$$
$$P(\text{absorb}) = 1 - 0.3 - 0.1 = 0.6$$

Note that this imposes the constraint that the specular plus diffuse coefficients f a material cannot exceed 1; this is also a true in the real world.

Every time we hit a diffuse surface, we store the photon's intensity, along with the hit coordinates, in a k-D tree which is used later in the rendering phase. To simulate caustics, we are only concerned with photons that hit diffuse surfaces after going through at least one specular interaction; these are the only photons we store in the map.

1.1.2 Projection Maps

During the caustic photon map preprocessing phase, we spend time shooting a photon in a given direction only after checking that there are specular objects in that direction by querying the light's projection map.

Projection maps are bit arrays in which individual entries are mapped to areas on a conceptual sphere wrapped around the light. If an entry is set, then there are specular objects in that direction. The mapping uses polar coordinates (and so is non-uniform).

To fill in projection maps, we shoot rays in the directions corresponding to each entry and see whether they hit specular objects. All entries neighbouring an entry set in this way are also set to guarantee that we don't miss anything despite sparse sampling.

Projection maps were one of my most difficult features to implement, as projecting vectors onto a unit sphere and mapping the sphere to an array required several error-prone bi-directional trigonometric conversions. Take a look at my `ProjectionMap` class in `photonmap.cpp` to see what I mean. It was only after implementing a debugging feature to draw the outline of my projection map that I was able to get things right. Due to the non-uniformity of my parametrization of the spherical map, it was also difficult to compute the proportion of the occupied area of the map. This computation is required for proper caustic intensities (see section 1.1.1).

Here are the timings of the photon mapping proprocessing phase for a simple image with three specular objects with and without using projection maps (see `caustics1.png`). 10% of the light's spherical projection map is occupied in this scene.

System	Without map	With map	Speedup
FX-6100	32.5s	10.1s	3.22x
Core i5-3317U	26.4s	8.3s	3.18x

As you can see, the use of projection maps can significantly reduce the runtime of this preprocessing step. Often, though, the main rendering phase accounts for the brunt of the program's runtime and so this optimization makes little difference overall. The optimization is most useful when you want high-quality caustic effects, as these require large numbers of generated photons.

There is no benefit, or even a slowdown, if we use projection maps in sparse scenes, as it is less useful to avoid the ray intersection code. `caustics2.png` is such a scene. Here are its timings with and without projection maps:

System	Without map	With map	Speedup
FX-6100	21.6s	30.3s	0.71x
Core i5-3317U	25.6s	37.5s	0.68x

Even for sparse scenes, though, the projection map is beneficial, as it guarantees that specular objects get their fair share of photons.

1.1.3 Computing Caustic Radiance

We integrate caustics into our lighting model by simply adding 'radiance estimates' retrieved from our caustic photon map to our other diffuse surface colour computations (ie, the Phong model).

Given a point, we compute the radiance estimate for that point by querying our caustic map k-D tree for the N nearest neighbours, where N is a user-specified parameter. We then return the area-average of the results; that is, we sum the resulting intensities, and divide by the area of the smallest disc that includes all of the photons. Given that photon intensities are initially chosen based on the number of photons to be shot, this computation yields a reasonable estimate of the true caustic radiance.

1.1.4 The k-D Tree

For images with caustics or global illumination, the photon map k-D tree structure is often the major performance bottleneck (as high quality effects require that massive number of photons are stored and high numbers of neighbours are queried).

Before the ray tracer's rendering step, it must build the caustic photon k-D tree from the photons produced by the photon shooting preprocessing step. This involves repeatedly finding median points on the x, y, and z axes. My relatively naive implementation sorts the points to find the median, which results in an $O(n \lg^2 n)$ algorithm.

Finding the N nearest neighbours of a point in a k-D tree entails a recursive algorithm that traverses the binary tree, checking at each level whether the node's point is closer than the current candidates to the target, and deciding to visit children nodes based on whether the children on that side of the split can possibly be closer than the current candidates. Usually, we'll need to visit many leaf nodes before we find the desired number of points. The resulting algorithm has complexity $O(k \lg n)$, where k is the number of neighbours to be retrieved.

1.2 Constructive Solid Geometry

Our constructive geometry algorithms iterate through pairs of lists of line segments (one list for each operand), determining where the next segment begins or ends and pushing new segments into an output list based on the operation type (union, difference, or intersection) and the interactions between the lists. The relevant code is in `csg.[ch]pp`.

The nature of our proposed final scene required us to implement interfaces between materials other than air for the purpose of refraction (for example, we needed to be able to accurately model transmission of light from a glass to a liquid inside the glass and back). We found that the constructive solid geometry union function provided us with a perfect opportunity to acquire the necessary information about adjacent mediums.

Thus, our constructive solid geometry union function returns information about a 'from' medium and a 'to' medium. In particular, the function records an interface between two materials when

- A line segment for the second operand begins during a line segment for the first operand

- A line segment for the second operand ends during a line segment for the first operand
- A line segment for one operand ends within ϵ units of the beginning of a line segment for the other (for some small ϵ)

Note that the resulting semantics have the second union operand "override" the first in places where they overlap. We did not augment the constructive solid geometry difference and intersection functions in this way, as coming up with consistent and intuitive semantics (and writing code for them) was difficult; moreover, the union functionality was sufficient for our purposes.

Note also that this feature has no effect whatsoever unless the materials involved are transparent.

To accomodate this feature, our ray tracing code had to be modified to be aware of the possibility of a 'from' medium.

1.3 Reflection and Refraction

Reflection and refraction both involve shooting recursive rays in specific directions. In the case of refraction into a coloured medium, the light intensity is attenuated as a function of the distance travelled in the medium.

In our implementation, a single colour dictates the intensities of reflection and refraction for a material (though either feature can be individually disabled). The relative proportions of this coefficient that contribute to reflection and refraction, respectively, are determined by the *Fresnel equations*. In particular: given an incident angle away from the normal of θ_i, a transmitted angle away from the normal of θ_t, and refractive indices n_1 and n_2 for the incident and transmission materials respectively, the proportion of specular light which is reflected – the rest is transmitted – is given by

$$\frac{1}{2} \cdot \left(\left(\frac{n_1 \cos\theta_i - n_2 \cos\theta_t}{n_1 \cos\theta_i + n_2 \cos\theta_t} \right)^2 + \left(\frac{n_2 \cos\theta_i - n_1 \cos\theta_t}{n_2 \cos\theta_i + n_1 \cos\theta_t} \right)^2 \right)$$

We were motivated to use the Fresnel equations since our important features as well as our final scene were heavily dependent on refraction, and the Fresnel equation avoids the discontinuities that arise in simple refraction models when total internal reflection occurs.

Our primary reference on the Fresnel equations was de Greve [2]. The relevant code is in `rt.cpp`. In particular, see the `fresnel` function.

1.4 Parallelism

In assignment 4, I took advantage of the "embarassing parallelism" of ray tracing by making my ray tracer multithreaded. In particular, I spawn a number of threads, each of which repeatedly queries a threadsafe queue data structure. This structure doles out rows of the image being rendered, and each thread, upon receiving a row, renders it before asking for another. The result is a significant speedup.

I have not parallelized the photon mapping preprocessing stage, due to both time constraints and the fact that the rendering stage is the main bottleneck.

Here are some timings for different systems using different numbers of threads while rendering my `threading.png` scene:

Core i5-3317U (two cores hyperthreaded)

# threads	User time	Clock time	Speedup
6	127.1s	33.1s	2.51x
4	127.0s	32.8s	2.53x
2	84.3s	42.4s	1.96x
1	83.3s	83.1s	1.00x

FX-6100 (six cores)

# threads	User time	Clock time	Speedup
6	28.9s	4.9s	4.59x
4	27.3s	6.8s	3.31x
2	23.4s	11.7s	1.92x
1	22.5s	22.5s	1.00x

As you can see, multithreading my program increased its performance considerably. Moreover, the improvement is mostly scene-independent.

2 Methodology and Tools

2.1 Language and Style

While writing my ray tracer, I made extensive use of C++11 features. The program is compiled with gcc's `-std=c++0x` option. Of particular note is my use of `std::function` and C++11 lambdas in my ray tracing and primitive ray intersection functions.

I made use of standard library data structures and algorithms whenever appropriate. Dynamic memory allocation was never a performance issue.

2.2 Configurability

The ray tracer has on the order of 20 different user-specifiable options that govern the appearance of generated images. These include things like the sampling resolutions for various features, the number of photons to shoot for photon mapping, whether to do special renderings meant to demonstrate certain features, et cetera.

There are two ways to control every option: you may specify them on the command line, or you may specify them in a LUA script via the `gr.option` call. `gr.option` accepts as its argument a string which is interpreted as a sequence of command-line arguments.

Having `gr.option` is advantageous because it frees the user from having to remember the desired command-line parameters for a given image. Requiring the user to pass a command-line string to `gr.option` (rather than having dedicated LUA functions for every option) gets around the need to add boilerplate to `scene_lua.cpp` every time we want to add a new option.

All of the code related to program options and command-line arguments is in `cmdopts.cpp`.

2.3 Performance

I spent a good deal of time attempting to improve the performance of my ray tracer in ways unrelated to my project goals. Currently, the biggest performance bottleneck for caustic-enabled scenes is the k-D tree N nearest neighbours function, which is usally invoked with values of N on the order of 100 and must search through a binary tree containing hundreds of thousands to millions of photons.

I was able to significantly reduce the time spent in the nearest neighbour search by adding a maximum distance to the search; beyond this distance, we simply give up and return whatever we have. This allows us to return early when we search for photons in dark areas of the scene rather than wasting time gathering photons which will be too far away to influence the radiance.

Constructive solid geometry objects significantly impacted performance before I implemented hierarchal bounding boxes (which are now automatically placed around CSG primitives). Here are some timings with and without these bounding boxes for my `csgbb1.png` script:

System	No bounding boxes	Bounding boxes	Speedup
FX-6100	35.7s	3.8s	9.39x
Core i5-3317U	54.6s	5.8s	9.41x

Clearly, this optimization has a significant impact on performance in scenes with characteristics.

Perhaps unsurprisingly, I doubled the speed of my sphere intersection function by converting it from a generalized quadric intersection function into one specific to spheres.

There was one case where initializing a `std::function` with a C++11 lambda caused a dynamic allocation in a hot loop, and initializing the function outside the loop improved performance by about 10%. In general, though, dynamic memory allocation was not a significant contributor to performance issues, despite my frequent use of standard library containers.

2.4 Tools

I made use of the following tools:

- I used Valgrind's callgrind tool in order determine the best targets for optimization (given reasonable k-D tree sizes, the program is CPU-bound).
- I used GNU's gdb whenever I needed to debug a crash (although I generally used print statements for other types of debugging).
- I used GIMP, the GNU image manipulation program, to generate several textures that I used for testing and in demo images.
- I used Git for version control. For general interest, I had made 68 commits in my project `src` directory at the time of writing.
- I used `bitbucket.com` for free private Git hosting.

3 Bugs and Assumptions

I have run into bugs with CSG objects being clipped. This may be a bug in my CSG hierarchal bounding boxes. The bug rarely surfaces, and so I haven't devoted much of my precious time to fixing it. Nor have I disabled CSG bounding boxes, as they provide an excellent speedup.

4 Future Possibilities

Several extensions to photon mapping are described in great detail by Jensen in [1]. I am most interested in experimenting with the use of photon mapping for global illumination, as the images produced through the use of global illumination techniques can be very impressive. There is also a discussion of the use of photon mapping to render participating media such as fog; smoke would be a great addition to my final scene. Subsurface scattering is another possible extension of photon mapping.

I have already put some effort into achieving global illumination, as there is very little I need to implement in addition to what I already have for caustics. The resulting images are promisig, though not polished yet (see for example `gi1.png` and `gi2.png`).

5 Acknowledgements

- I received help from Lucie Lemmond with improving my final scene.
- One of the textures I used in my final scene was a public domain image I retrieved from the Wikimedia Foundation [3].
- I'd also like to point out my dependence on the GIMP's excellent features for texture and bump map generation.

6 Code Map

What follows is a brief description of what you can find in the various source files, ordered by approximate importance. I have omitted files which were present in the assignment 4 base code and which haven't been significantly changed.

- `a4.cpp` – `a4_render`, the main render function.
- `rt.cpp` – The `RayTracer` class, which encapsulates most of the logic for recursive ray tracing (including reflection and refraction).
- `primitive.cpp` – Primitive definitions, including intersection testing functions.
- `lightingmodel.cpp` – Lighting computations, including soft shadows (in particular, see the `occlusion` function).
- `csg.cpp` – Code for constructive solid geometry.
- `photonmap.cpp` – Photon map creation, radiance estimation, and projection map code.
- `kdtree.hpp` – The `kdtree` template class.
- `textures.cpp` – Texture and bump map definitions; support for PNG image textures and bump maps.
- `texdefs.cpp` – UV Remapping code and procedural textures and bump maps.
- `intersection.hpp` – Some fast intersection functions that are used in several places.
- `threading.hpp` – Multithreading support.
- `timer.cpp` – The `ProgressTimer` class, which prints rendering time and progress.
- `stats.cpp` – Support for recording and printing statistics.
- `cmdopts.cpp` – Program options and command-line argument parsing.

7 References

[1] Henrik Wann Jensen, *A practical guide to global illumination using ray tracing and photon mapping.* ACM SIGGRAPH 2004 Course Notes, 2004 [exceptionally detailed, thorough, and readable SIGGRAPH course notes on photon mapping and its extensions]

[2] Bram de Greve, *Reflections and Refractions in Ray Tracing.* November 2006 [information on using the Fresnel equations for reflection and refraction]

[3] Wikimedia Commons (unknown author), *Graph of the Growth of Boston Schoolchildren in Height and Weight*. 1907 [public domain image used in my final scene]