[redacted]: “ctrace”

[redacted], [redacted]

1. Overview

ctrace is a raytracing engine written in C. It is capable of rendering a few simple primitives to an output file.

2. Basic capabilities

ctrace supports the following primitives; spheres, axis-aligned boxes, infinite planes, triangles, and planar polygons of arbitrary vertex count.

The rendering engine uses the Phong-Blinn illumination model, with shadows and specular highlights. It supports multiple point light sources at infinity, but no spotlights. Reflection and refraction are supported, with arbitrary recursion depth and blending levels.

Texturing is supported for infinite planes and spheres, with a choice of bilinear filtering or nearest-neighbour resizing. Anti-aliasing of the image is supported.

The engine is multithreaded, although no space partition schemes were implemented.

Constructive Solid Geometry (CSG) was partially completed. Arbitrary transformations were planned, however time constraints prevented their implementation.

A significant amount of customisation is available in config.h, to provide the user with a speed/quality tradeoff by selectively disabling engine features.

3. Overview of Extended Features

3.1. Other Primitives

ctrace supports both triangles and planar polygons of arbitrary vertex count using the even-odd method [3]. This method is as follows; flatten the polygon to two dimensions. From the point of intersection, draw a line off in any direction. If it passes through an odd number of lines, the point is inside the polygon; otherwise, the point is outside the polygon.

Ordinarily, this would be sufficient to implement both triangles and boxes. However, triangles have a more efficient implementation using the Möeller-Trumbore algorithm [4], and axis-aligned boxes are implemented using the Slabs method [1].

For ray/triangle intersection, the Möeller-Trumbore algorithm describes a fast way of translating and scaling the triangle and ray such that the ray is parallel to an axis. This allows for fast bounds checking simply by taking dot products of the ray direction with the edges of the triangle.

For ray/box intersection, the Slabs method is simple and well publicised; for each set of adjacent faces, assume them to be infinite planes and determine if the ray crosses the near or far one first.

Most examples of the Slabs method provide several conditional cascades [1], which would have made sense in earlier days of computing; however, with the growing pipeline length in modern processors, avoiding branch misprediction is a crucial

element for performance. Müller and Geimer [2] describe a method for unrolling the computation in a way that suits such devices.

3.2. Multiple light sources

ctrace supports a list of light sources.

3.3. Refractions

ctrace supports refractions. A material has an associated refractive index, as well as a blending factor.

3.4. Anti-aliasing

ctrace supports anti-aliasing. The primary technique for doing so is Full-Scene AA (FSAA), also known as Super-sampling: the scene is rendered at a full 4x resolution, and then a bilinear filter is used to resample it to the target resolution. This provides a significant quality enhancement at the cost of a 4x speed penalty; for most scenes this still gives acceptable performance owing to the choice of compiled language.

Anti-aliasing based on edge detection was researched, temporarily implemented, and was found to be noticeably more performant than FSAA. However, it was ultimately removed for its inability to comply with certain abstractions in the codebase, as well as the low-quality shadows it afforded.

3.5. Arbitrary transformations

Arbitrary transformations were planned, and significant provision was made for them in the code. However, they were not completed, owing to time constraints.

3.6. Textures

3D Textures (normal mapping / bump mapping) were not implemented. However, flat textures in PNG format are supported on spheres and infinite planes.

3.7. Constructive Solid Geometry

CSG was implemented. Part of the provision made for arbitrary transformations was the creation of a meta-object framework; a CSG object intercepts the collision and normal generation, and replaces it with the union, subtraction, or intersection of two other objects as appropriate. Using meta-objects in this way allows for CSG objects to be tested against other CSG objects, removing the restriction to only two primitives.

For both the two objects, a ray is traced toward the object. If there is an intersection, a second ray is traced in the same direction to find the exit point. This gives us a one-dimensional range of points considered ‘inside’ each object. Simple tests allow us to determine the intersection, union and subtraction of each range; these correspond to the CSG operations on the two objects.

Implementing CSG was a relatively simple affair, although it required some rearranging of abstractions in the codebase in order to minimise the amount of times the intersections were recalculated. Furthermore, time constraints meant that it was infeasible to complete the feature; as it stands there are some graphical issues related to determining the surface normals of CSG solids (see § 7.3 for details).

3.8. Acceleration algorithms

Although no space partitioning structures (such as a BSP tree, BVH or a kd-tree) were

implemented, the techniques were researched.

The engine is multithreaded, using the native Win32 threading API on Windows and pthreads elsewhere. The scene is split into one horizontal stripe per thread, in order to minimise inter-thread communication. This technique provides a near-linear speedup up to the number of cores available in a given machine.

For simple scenes like the test image, it is expected that the cost of generating a BVH could outweigh the cost of naïve image search. However, for more complex scenes, or for multiple renderings of the same scene, the benefits would quickly become apparent.

4. Performance

ctrace's performance is far from realtime (which could be attainable [7]) but more than acceptable for its purpose. Having completed the ray tracer up to Lab 4 in Python, performing a straight conversion into C with a hand-written bmp exporter cut the renderer's runtime from approximately 4 seconds to just 80ms, a 50-fold speedup - and this was prior to the implementation of multithreading.

In general, moderately complex scenes of under 50 objects render in seconds on modern hardware, with 4xFSAA anti-aliasing and multithreading enabled.

5. Compilation

ctrace is written in ANSI C and was developed with gcc 4.5.2. With some exception for comment styles, it compiles without warnings in strict C89 mode, although a minor performance loss was noticed compared to the more relaxed gnu89 default. Build options were tuned for speed, including -march=native, -O3 and -ffast-math. The suggestion on the Learn forums to use the flag -OFast was considered and rejected owing to it being a new feature in GCC 4.6 which, although available for my operating system, is not yet packaged by my preferred distributor.
(-OFast only adds -ffast-math anyway.)

A shell script was used to control building the project in lieu of a makefile, providing several simplicity advantages. Mercurial was used for local source code control.

There are several options available in config.h regarding speed/quality trade-offs that may be useful to those evaluating the software. For instance, time-consuming features such as anti-aliasing can be disabled while prototyping new features.

The third-party LodePNG [8] library is linked statically under the terms of the BSD License, to enable PNG file format support for exporting images and reading textures. This replaced a hand-written BMP library.

6. Summary

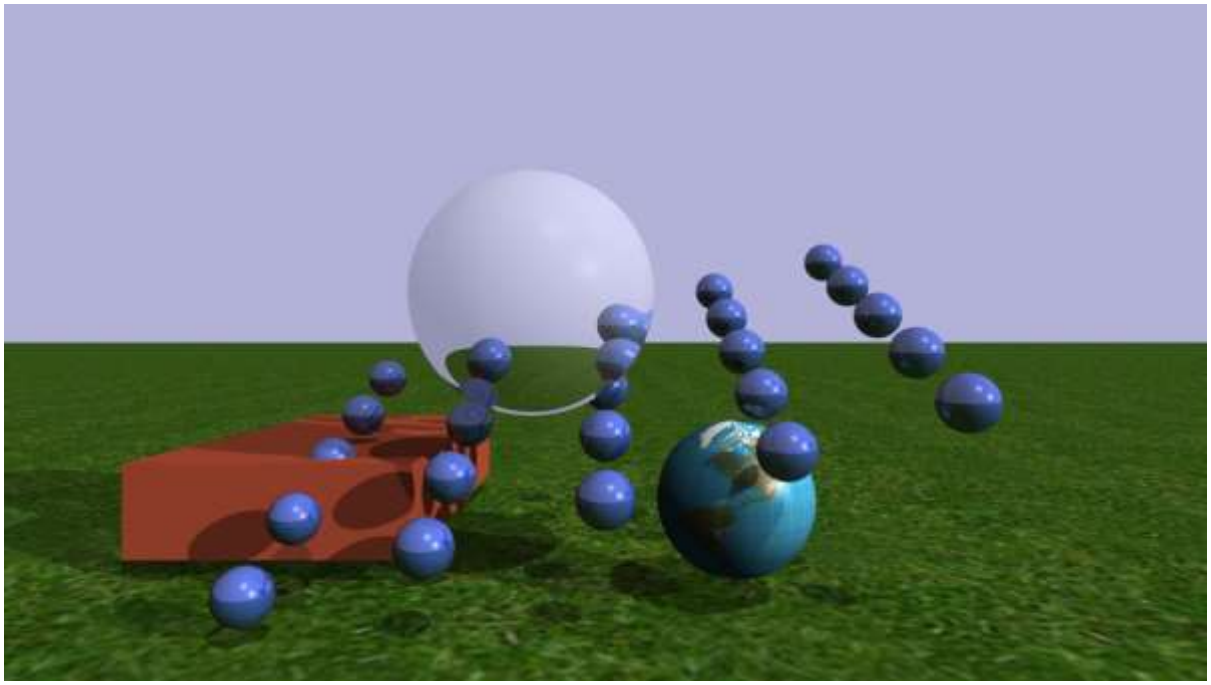
Overall, ctrace was enjoyable to write, and it meets the primary criteria for the assignment, as well as implementing several of the extension features. Porting the engine from Python to C resulted in a significant speedup, and allows the output binary to be small and self-contained.

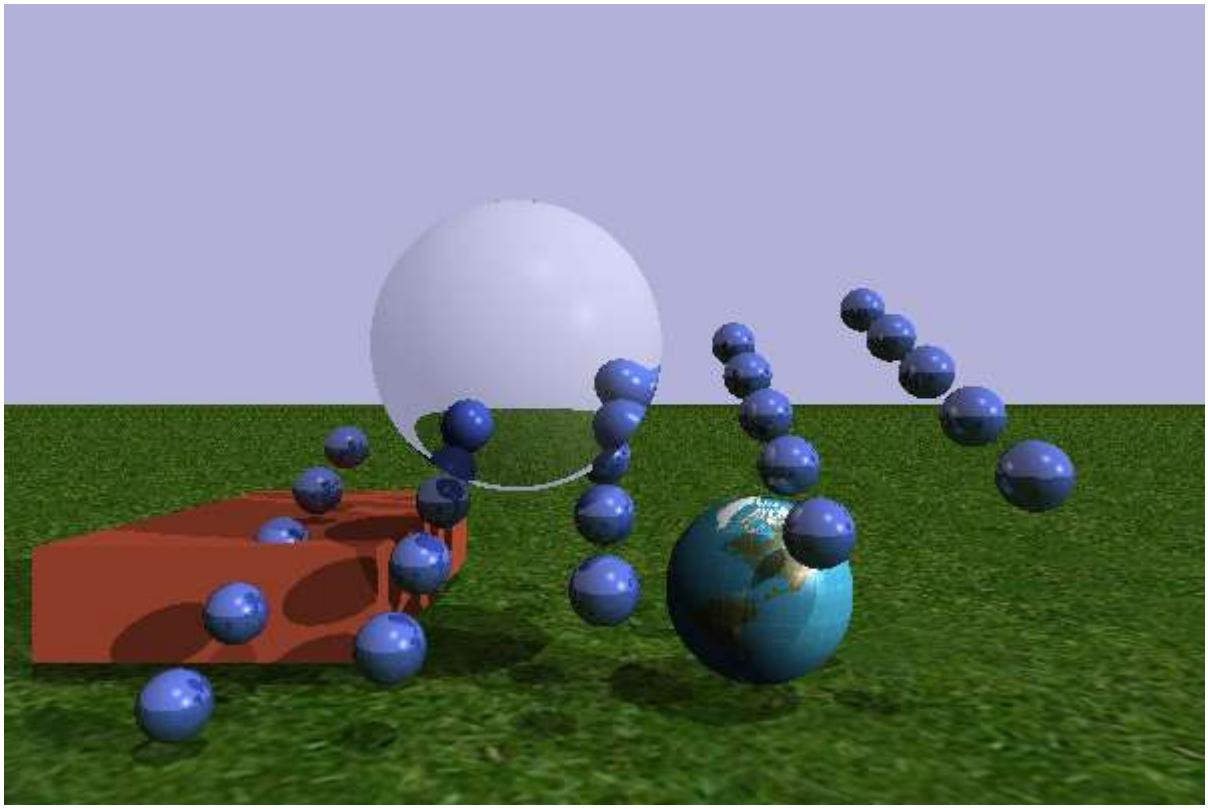
Although there was insufficient time to complete some of the more exotic features such as CSG, 3D textures, spotlights, soft shadows, transformations and space partitioning algorithms, it is still capable of producing a visibly pleasing output.

7. Example Output

7.1. Figure 1.

The test scene used when developing the raytracer, demonstrating some core features. **Above:** rendered at 1920x1080 with 4xFSAA, bilinear texture filtering, and 4 levels of reflection depth. This render took 38.7s to complete. **Below:** rendered at 600x400, bilinear texture filtering, and 1 level of reflection depth. This render took only 0.7s to complete.



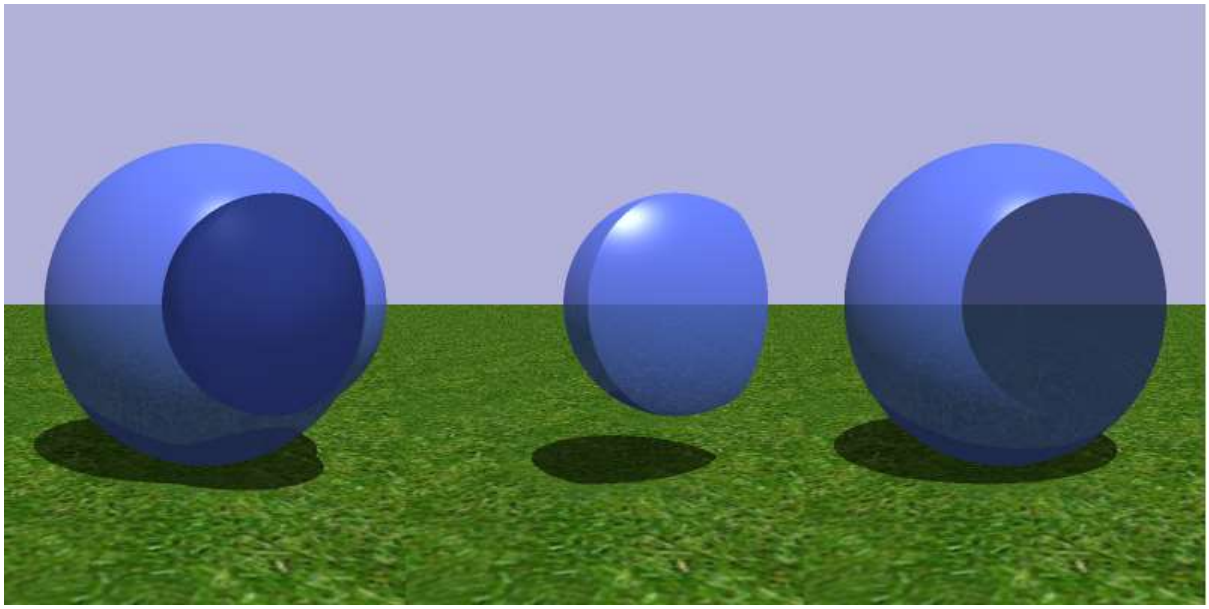


7.2. Figure 2.

A hollow Utah Teapot was modelled in Autodesk 3DSMax and exported to the ASE format. The vertex data was then manually extracted and converted to a set of triangle objects. Here, ctrace is rendering 1024 triangle objects at 600x400, with no anti-aliasing or reflections. This render took 38.6s to complete.

7.3. Figure 3.

Constructive Solid Geometry (CSG) was implemented, although time constraints prevented the correction of some graphical errors regarding the surface normals. Here, ctrace is performing three renderings of the union, intersection, and subtraction respectively of two reflective spheres, each at 800x400 with 4xFSAA. Rendering time was on the order of 0.5s per scene.



7.3. Hardware.

Where noted, the times are as performed with multithreading enabled on an early-2007 model Intel Q6600, running at its stock speed of 2.4GHz. During rendering, all cores appear to be at 100% utilisation.

Times were initially measured using the standard UNIX time command; however when PNG support was added for loading textures and exporting the rendered scene, it was considered that this added a measurable delay in execution. Hence an internal timer was added, the results of which will be visible if ctrace is ran from a command prompt.

8. Citations

The internet was, as always, a valuable resource whilst developing ctrace. The lecture notes were also obviously indispensable. A list of references follows.

[1] “Ray-box intersection”, G. Scott Owen (1998)
http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm

[2] “Branchless Ray/Box intersection”, Thierry Berger-Perrin
http://ompf.org/ray/ray_box.html

[3] “PNPOLY - Point Inclusion in Polygon Test”, W. Randolph Franklin (2006)
http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html

[4] “Fast, Minimum Storage Ray/Triangle Intersection”, Möller and Trumbore (2003)
http://www.cs.virginia.edu/~gfx/Courses/2003/ImageSynthesis/papers/Acceleration/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

[5] “Raytracing: Textures, Cameras and Speed”, Jacco Bikker (2005)
http://www.devmaster.net/articles/raytracing_series/part6.php
Note that this article has a bug; see ctrace/object.c:324 for details

[6] “Ray Tracing: Graphics for the Masses”, Paul Rademacher
http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html

[7] Arauna, a realtime raytracing engine
http://igad.nhtv.nl/~bikker/

[8] LodePNG, a PNG image decoder and encoder
http://lodev.org/lodepng/