CS 418 Program 5: Ray

(revised April 29, 2003)

out: Tusday, April 15, 2003
due: Monday, May 5, 2003

In this assignment you will extend the modeler that you wrote in Programs 3 and 4 by adding the ability to make nicer renderings of the scene using a ray tracer. Your ray tracer will live off the same scene data structures that the modeler uses, so that you can use your modeler to set up a scene and camera, then render a high-quality image.

Your renderer will support all the materials and geometry types used by the modeler, with a basic algorithm that supports shadows and mirror reflections. You will also need to implement at least one feature from the list of extensions below; you may implement multiple extensions (or add other features) for extra credit.

Required features

The user runs your ray tracer by setting up a scene, moving the perspective camera to the desired view, and choosing a "Render..." command. This brings up a window containing controls to set the rendering parameters (at a minimum, the image size, maximum recursion depth, and number of samples per pixel), an image area where the result will appear, and a "Render" button. It's not required to make the image appear bit by bit while it is being computed, but you will make using the program much more convenient for yourself if you do. At the minimum you should display a progress bar; the user needs to know whether the rendering will finish in 10 minutes or 10 weeks.

Your ray tracer must implement all these features:

- Geometry support for spheres, cubes, cylinders, cones, and triangle meshes. You should treat generalized cylinders as triangle meshes, but the more basic shapes should be intersected exactly.
- Support for the perspective camera (you do not need to handle the orthographic case).
- Antialiasing by supersampling on a regular grid with box filtering.

- Shaders for the Lambertian and Phong materials. These shaders should match the behavior of the hardware lighting, and the shader for the Phong material should also include a mirror reflection component that is controlled by the specular coefficient (that is, the reflection you see in a surface is k_s times the reflection you would see in a mirror). These shaders should compute shadows.
- A basic ray-tracing structure that calls the shaders for surface intersections and controls the recursion depth using a simple cutoff based on the number of bounces. The maximum number of bounces should be a user-adjustable parameter.

For this assignment you will also hand in three images to demonstrate your ray tracer. You should make the best looking images you can, technically and aesthetically. There will be awards for the best renderings. Be sure to leave enough time to run the final images at high resolution and with good quality settings!

Extension features

You must implement one of the following extensions. You may implement more for extra credit (5 points each).

- Improved materials. Implement a shader for dielectric materials (like glass or water) that produces refracted and reflected rays with the correct Fresnel factors. A shader that uses two recursively computed rays means that your renderer will generate a tree of rays, which needs to be pruned to keep the program from becoming too slow. In addition to the maxiumum-depth cutoff, you should also implement a maximum-attenuation cutoff by keeping track of how much a given ray will contribute to the image (i.e. what is the factor it is being multiplied by before it is added to the image). When that factor drops below a user-determined threshold, you should terminate recursion.
- Distribution ray tracing. Use randomly distributed rays to produce soft shadows and glossy (as opposed to mirror-like) reflections from Phong materials. This means that each light source will need to have a radius, and it obviates the separate highlight and mirror reflection terms (both will be handled by the same Phong lobe). You should treat light sources as flat circles that always face the point being shaded (this approximates spherical lights but is much easier to implement). The light sources do not need to act as geometry—that is, a light source does not have to occulde an object that's behind it (although it's fine if it does).

  Note that this option does not involve very much code but it does involve more challenging math than the other two.
- 2D raster surface textures. Implement texture mapping based on images. This involves defining texture coordinates for all the geometry types and reading in texture

maps that then modulate the diffuse component of whatever shading model you are using. All of the geometry types other than triangle meshes can have their own native (u, v) coordinates; for triangle meshes you should implement sphere projection with the center at the average of all the vertices in the mesh.

The texture coordinates should be as follows:

- Cylinder and cone: u goes from 0 to 1 as you go counterclockwise around the surface (viewed from the direction of the +y axis), and $v = (y + 1)/2$. For the end caps the texture coordinates can be anything between 0 and 1. Note that for the cone the texture will be very compressed near the tip.
- Sphere: u is longitude, going from 0 to 1 as you go counterclockwise around the equator, and v is latitude, going from 0 at the $y = -1$ pole to 1 at the $y = +1$ pole. (Note that v should *not* be $(y + 1)/2$).
- GC: u is the transverse direction and v is the radial direction. Each should run from 0 to 1 across the whole shape (not for each spline segment).

You can find lots of nice textures on this web site:

http://astronomy.swin.edu.au/ pbourke/texture/

For these extension features you will need to extend the file format to allow the extra information (new material types, radii for light sources, texture images) to be stored in the input file. You should define the extensions however you like, and implement them by adding to the interfaces of the parser. When you do this, be careful to only add attributes (don't remove any or modify the meaning of existing ones) and to define defaults so that basic files still can be read and will work correctly.

Framework code

Because intersecting every ray with every primitive is very slow for large numbers of primitives, all useful ray tracers incorporate spatial data structures of some sort. This "acceleration structure" is used to efficiently intersect a ray with the scene by quickly discarding all but a few primitives that are near the ray, and intersecting the ray only with those.

Because triangle meshes can contain large numbers of primitives, but implementing an acceleration structure is too much work for this project, we are supplying an implementation of a common acceleration structure for you to use (see below).

See the web site for the code and details about the API.

Implementation notes

You should be able to use the same scene data structure for your ray tracer that you are already using for the modeler, though you may find it convenient to store some extra transformations. You can and should intersect the individual primitives in their canonical coordinate systems (i.e. intersect rays only with the unit sphere, etc.) by transforming the ray by the inverse of the object's transform, then transforming the results back into world space. Don't forget that normals need to be transformed by the inverse transpose of the matrix you use to transform points and vectors.[1]

We are providing an acceleration structure, and this affects the design of your ray tracer. You provide surface classes that can compute bounding boxes and intersect rays, and the acceleration structure supports two operations: creation from a list of surfaces and intersection of a ray with those surfaces. In the creation phase it will get the bounding box from each surface and use those boxes to construct the acceleration structure; in the intersection phase it will intersect the ray with a subset of the scene that is near the ray, returning the closest surface along the ray.

One implication of this setup is that a surface needs to have some way to get a hold of its transformation in world coordinates (i.e. the concatenation of the transforms of its ancestor groups).

A debugging hint: if rays seem to erratically miss things they should hit, suspect a problem with your bounding boxes.

Ray tracing can use a large amount of CPU time. If your program is only usable in its usual interactive mode, you will only be able to render while tying up a display. If you hope to apply a lot of cycles to rendering your final images, you will want to implement a batch mode so that you can use CSUGLab's Linux compute servers, each of which has 1 GB of RAM, and which together have 12 CPUs ranging from 1 to 2 GHz. It should be very simple to implement a mode that takes a list of filenames from the command line and hands them to the parser, then renders an image using the resulting camera, scene, and lights. One idea for setting the rendering parameters (image resolution, output filename, ...) is to have the first command line argument be a special parameter file and the rest be input files.

This may be obvious, but when you are debugging your code you should use low image resolution, a small number of lights, and one sample per pixel, because that will make your ray tracer finish quickly.

Groups

This project is to be done in groups of two. In cases where groups have been re-formed

[1] Note that despite the presence of the word "normal" in the documentation for `vecmath.Matrix4d.transform(Vector3d)`, the transformation applied by that method is just the usual matrix multiplication with zero in the w component; it is not appropriate for transforming surface normals. It is appropriate for ray directions, though.

because of students who dropped the class, any or all of the code that was part of either partner's previous assignments is fair game to build on.

Extra credit

As always, we encourage you to add extra features for extra credit. The ground rules are that (a) a program that scores below 95/100 on the basic requirements cannot earn any extra credit and (b) no program can have a score greater then 120/100. Here are some suggestions that would be worth between 2 and 5 points out of 100 if implemented well:

- Implement a shader for Ward's isotropic reflectance model.
- Adaptive sampling in image space. This should control the density of samples used to do antialiasing, but it should also allow for subsampling (fewer than one sample per pixel) in smooth areas.
- Real time image display. Display the image as it is being computed, using multi-threading so that the program is still responsive. Provide a "Cancel" button so that you can stop part way. Doing this in conjunction with the previous item, you could display a nice adaptive refinement where you first display a low-resolution image, then refine it as more pixels come in.

  Unless you take remarkable measures to allow concurrency you will need to block editing the model while the ray tracer is running.
- Procedural solid textures. Allow for 3D texturing and implement a few variations on a procedural turbulence texture.
- Improved antialiasing. Implement Gaussian filtering, and randomly jitter the sub-pixel samples to further reduce artifacts.
- Depth of field. If you implemented distribution ray tracing, add the option to simulate a camera lens that focuses at a particular distance.
- Bump maps. If you implemented 2D texturing, add bump mapping based on a grayscale image that represents a height field.