CS184 Final Project — Global Illumination and Photon Mapping with Extended Ray Tracer

cs184-ad: Gabe Fierro, cs184-ae Emerson Knapp, cs184-au Harry Zhu

15 December 2011

Abstract

Our original goal for our final project was to implement additional features for the ray tracer we created for **Assignment 4**, and then implement photon mapping to augment the scenes we could then create with the modified ray tracer.
We have a pair of "final" images: (image12.png) which is run with 200k photons, 200k caustic photons, 100 shadow rays and without smoothing. (8.png) is an example of the indirect illumination, and (9.png) is an example of indirect plus the regular ray trace (no caustics, but you can see the soft shadows).

Contents

1 Features

We implemented the following features for the raytracer:

SCENE FILE INPUT

We created our own file format for detailing scenes for our program to render. See `Appendix A` for a brief rundown of the file format we use. Underlined file names refer to images in the `images/` folder. If you're using a pdf reader such as Skim (and possibly Adobe Reader), then the underlined names should function as links to those images.

ANTI-ALIASING

We implemented both uniform and jitter anti-aliasing techniques. Specifying the `-a N` option sends N^2 rays to each point on the screen (as opposed to the default 1), regularly spaced around the unit square represented at that pixel and averages the results to specify the color for that pixel. Specifying the `-ja N` option sends N^2 rays to random locations within the unit square and again averages the results. In practice, $N = 3$ is sufficient for the 800×800 images we generate, and uniform and jitter achieve essentially the same effect.

Images: Compare (image1.png) this image with no aa to the same scene, but (image2.png) with nine rays per pixel (run with `-ja 3`)

REFRACTION AND REFLECTION

Both rays and photons correctly refract and reflect when intersecting with objects in the scene according to the object's refractive index and reflectivity, respectively (details for photons below). Objects can be both reflective and refractive and we see the correct result.

Images: You can see an example of refraction here: (image3.png)

EAR-CLIPPING

We specify vertices and faces in the manner of the OBJ file format: `v x y z` specifies a vertex at coordinates (x, y, z), and f $n1$ $n2$ $n3$ creates a face using vertices $n1$,$n2$ and $n3$. Using ear-clipping, we can specify arbitrary convex or concave polygons using the command f $n1$ $n2$ $n3$ $\cdots$ nk. The ear-clipping algorithm correctly identifies ears by differentiating between convex and concave vertices, and splits up the polygon into the appropriate triangles for rendering.

Images: (image4.png) This polygon contains both convex and concave vertices, and is specified by a single `f` line in the scene file.

PERSPECTIVE-CORRECT TEXTURE MAPPING

In the scene file, if a `.png` file is specified as a texture, then the objects in the scene use barymetric coordinates and linear interpolation (in the case of triangles) or uv-mapping using spherical coordinates (in the case of spheres) to use the texture as the surface of the object.

Images: Observe a black and white marble texture on a pair of triangles (image5.png) forming a rectangle, and a world map texture on a sphere: (image6.png).

AREA LIGHTS AND SOFT SHADOWS

We added a few features to simulate soft shadows. First, we added area light sources, which are are triangles that emit light. Then, each time we checked for shadowing, we sent multiple rays to random points on the light's surface, and scaled the contribution of that light by the number of shadow rays that did not reach the light. In order to optimize this procedure, we used "shadow photons", which were obtained by sending a new photon from each object intersection in the photon tracing stage straight through the object it hit. Then, by querying the shadow photon map, we were able to decide whether or not to send out shadow rays at all, which speeds up performance considerably for areas that are not shadowed.

Images: Here are examples of the scene with 1 (image7.png), 10 (image8.png) and 100 (image9.png) shadow rays.

OBJ FILE FORMAT AND MULTIPLE SCENE SUPPORT

Just as in an OBJ file, in our scene files, one can specify vertices and n-sided polygons (as seen above), specify materials (using `usemtl FILENAME.mtl`, where FILENAME.mtl is a material file) as well as specify other scene or obj files to be included in the scene being rendered.

Images: (image10.png) Here is an example of rendering the free `angel.obj` model in a scene of our own construction. We've made the angel refractive to demonstrate how we can integrate external obj files in our scenes.

KD-TREE AND GENERAL OPTIMIZATION

We construct a KD-tree for the polygons in order to reduce the lookup time for calculating ray intersection. For photon mapping, we also create separate trees for global photons, shadow photons and caustic photons in order to facilitate k-nearest neighbors lookups for estimating radiance, generating shadow rays and visualizing caustics, respectively.
Additionally, we used OpenMP (`http://openmp.org/`) to parallelize the pixel interpolation, ray intersection and occlusion detection loops, resulting in a drastic speedup. For basic rendering (simple directional light, with no reflection or refraction) of a 100,000 polygon object such as the `angel.obj`, we were able to reduce rendering time from 90 minutes to 15 seconds.

Images: (image11.png This image, consisting of almost 100,000 unique polygons, took about 15 seconds to render. We've titled it "Weeping Angel," after the alien race in the BBC show Doctor Who.
By the way, this is what happens if you incorrectly compute the bounding boxes....

2 Photon Mapping

2.1 Implementation Details

In order to implement photon mapping, we first determined how we were going to generate the photons that would propagate through the scene. For a point light, we generate random directions on a unit sphere centered at the light's position and generate N photons in those directions (N is specified b the `-ph X` option, where $N = X \cdot 1000$). In order to make sure that we maintain the same amount of illumination for the scene regardless of how many photons we send out, we specify the power for the light in the scene file and weight the photons accordingly so the sum total of their flux is that of the light. [For directional lights, we follow a similar process].
In order to correctly model the photons "bouncing" around the scene, we use the Monte Carlo method known as **Russian roulette** to use the power of the photon and the attributes of the materials in the scene to probabilistically determine whether a given photon is absorbed, has a diffuse reflection, has a specular reflection or is transmitted (i.e. through a material with a different index of refraction).

2.2 The Rendering Equation

For the purposes of photon mapping, we can consider the rendering equation as having four parts: direct illumination, specular reflection, caustics and indirect illumination. For each point x in the scene, we calculate each of these four terms and add them to get the color at point x.

DIRECT ILLUMINATION

Direct illumination is the radiance for a location x given the occlusion of other objects in the scene and the light from a source that hits that location. We used normal ray tracing to estimate this integral, but for debugging purposes (using the `-r` option) we performed a k-nearest neighbors search at point x to gain an estimate for the direct illumination.

SPECULAR REFLECTION

Photon mapping can be used to estimate the specular reflection for at a point x, but we need a good deal more photons in order to produce that effect (see notes for caustics below). So, as with direct illumination, we use the normal ray tracing to calculate the specular reflection at a point x.

CAUSTICS

With the global photon map, because we get a relatively even distribution of photons, we get an estimate of the caustic for refractive objects. We send an additional number of photons toward each refractive object in order to increase the number of photons taken into account for estimating the caustic, which we estimate by doing the k-nearest neighbor operation and averaging the photon flux (which will be more accurate because there's a higher concentration of photons).

At a given point x, we have to determine how much light comes from surfaces near x. To do this, we send out a certain number of "gather rays" from point x in random directions on a unit hemisphere with the same normal as x's surface. For each of these "gather rays" that intersect with a surface, we perform a k-nearest neighbors lookup to get the surface radiance at that point and scale that radiance according to the steradians of the solid angle formed by the approximate surface area of where we gather the photons at that intersection. The sum of the radiance returned by these "gather rays" is normalized to 2π, the surface area of the hemisphere in steradians.

Appendix A

"#"comment comments are not parsed into the scene
`v xcoord ycoord zcoord`: vertices for polygon rendering; requires x,y,z, coordinates as arguments
`vt xcoord ycoord zcoord`: texture support for polygons; arguments are between 0 and 1 that each map to a point on the texture file
`vn xdir ydir zdir`: normal support for polygons; requires a vector as the normals direction as an argument
`f vertex1/vt1/vn1 vertex2/vt2/vt3 vertex3/vt3/vn3 ...`: obj file format of polygons(vt and vn arguments are optional); supports more than 3 vertices due to earclipping
`s radius`: support for rendering spheres; requires a radius as an argument
`pl xcoord ycoord zcoord r g b power`: support for point lights; requires the position, intensity, and power level as arguments
`dl xcoord ycoord zcoord r g b`: support for directional lights; requires the direction and intensity as arguments
`tl vertex1 vertex2 vertex3 r g b power`: support for area(triangle) lights; requires a set of vertices(similar to f) as well as color intensity and power as arguments
`usemtl pathtomtlfile`: sets the material properties to the ones specified in the mtl file in the argument
`ka r g b`: support for ambient lighting; requires RGB color as argument
`kd r g b`: support for diffuse properties of objects; requires RGB color as argument
`ks r g b`: support for specular properties of objects; requires RGB color as argument
`kr r g b`: support for reflective properties of objects; requires RGB color as argument
`sp power`: support for the specular exponent of objects; requires the specular exponent as an argument
`ri indexofrefraction`: support for refraction materials; requires index of refraction as an argument
`cam`: sets the camera with the current transformations set before
`translate xtrans ytrans ztrans`: sets the translation matrix with the three coordinates in the arguments
`rotate xrot yrot zrot`: sets the rotation matrix with the three arguments
`scale xscale yscale zscale`: sets the scale matrix with the three arguments
`ct`: clears all previous transformations set
`tex pathtotexturefile`: sets the texture to be the file specified as an argument; texture must be a .png file
`ctex`: clears the previous texture specified, must use if you dont want a texture for the next renderable object
`load pathtofile`: support for multiple scene files; able to load in extra scenes from within scene files themselves (even supports multiple obj files)