Raytracer

Joe Doliner

March 18, 2010

1 Introduction

For my project I designed and implemented a toy raytracer. It does diffuse and specular lighting (per pixel of course) It also supports raytraced transparency, refraction and reflection. And radiosity texturing for bidirectional tracing.

2 Scene Language

Scenes are described using xml files. The scene schema has the following constraints:

- Scene files have one top level node called a scene node
- a scene node contains 0 or more light nodes, 0 or more geometry nodes, exactly 1 camera node, exactly 1 settings node
- a light object must contain a color node, an intensity node, a position node and a look at node
- a geometry node contains one primitive node specifying the shape of the object, 1 translation node and a material node, a material node specifies a diffuse color, reflactance, transparency, and refraction values.
- a camera node has a position node a look at node, an up node, a focal length node, a width node, and a height node.
- a settings node contains a background color node, radiosity node (enabling or disabling radiosity) and radiosity accuracy node.

Scene parsing leverages libxml2 which made the process pretty painless and reads the scene file into a Scene t struct which contains an array of pointers for geometry and light objects (as well as how many of each we have) a pointer to a camera object a pointer to a settings object.

3 Primitives

My raytracer is pretty limited in termis of primitives right now, it only parses and stores boxes, tori, spheres and planes. And only planes and spheres have intersection functions written which are needed to really do anything. On the other hand primitives are really easy to add due to the geometry abstraction.

4 Intersection

The heart of the program is of course the intersection functionality. I've implemented an Intersect Scene which takes a ray (the same structure we used for the other projects) and a Scene t * and returns the first thing the ray hits as an Intersection t. An Intersection t struct contains, a pointer to the geometry object it hit, the normal of the intersection the point in world space at which the intersection occurred, the parameter of the ray at which the intersection occurred, and u and v parameters for the textures of the objects (used for radiosity).

5 Tracing

Using the intersection we can get a color value for each ray we shoot, then we can use that to calculate diffuse and specular intensity using fragment shading calculations. Also for each light we can shoot a ray at it and see if we get an intersection. If we do then we don't use that light in calculating diffuse and specular values since the fragment is in shadow. Then we can look at the geometry's material and if it has reflectance we reflect our original ray over the normal of the intersection, shoot again and blend the colors. Similarly if we have transparency we can do a snell's law refraction and reshoot the ray.

6 Radiosity Textures: the good stuff

If a scene enables the radiosity option then before we do any tracing every geometry object has a radiosity texture computed for it. Radiosity textures are computed using a monte-carlo technique, for each light, rays are shot randomly into the scene. When a ray intersects with an object we increase the value in the radiosity texture where the light ray hit. With enough rays the radiosity textures give a smooth accurate image of the illumination in the scene. Furthermore, everytime a ray hits a surface it also bounces off and gets shot back into the scene. This allows geometry to pick up on light not coming directly from light sources but bouncing off of other geometry first.

The paper I based my implentation on: `http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=C492EA2FEBCDC524D296C1C8F68F9353?doi=10.1.1.84.4862&rep=rep1&type=pdf`