CS488 Final Project Proposal

Daniel Burstyn (20206120)

Nov 20, 2009

Final Project:

1 Purpose

To implement a number of advanced ray tracer features and model an attractive scene.

2 Statement

The scene I plan to render is that of the top of a marble bathroom sink. On the sink there will be a clear glass containing a toothbrush, and sink stopper. The faucet will be running, and the sink will be full of water. The wall behind the sink will have a simple tile texture, and a mirror.

The sink stopper is a cut off cone that will require a cone primitive and CSG to make. A torus will also be used as the metal loop at the top of the stopper. The glass and toothbrush will require refraction.

The water in the sink will be rippled and require bump mapping. The sink and faucet will be constructed with CSG. The mirror and sink will both be reflective, and texture mapping will be used for the tiles, and running water.

3 Technical Outline

3.1 Additional Primitives

The additional primitives I plan to add are cones, cylinders, and tori. Cones and cylinders are conceptually quite similar to spheres as we can calculate the intersection of the ray using their simple geometric formulae.
Cylinders: $x^2 + z^2 = 1$ (Oriented on the y axis)
Cones: $x^2 + z^2 = y^2$ (Oriented on the y axis)
Cylinders and cones also need an additional check on the values since they are not supposed to be infinite. The equation for a torus is much more complex.
Tori: $(\sqrt{x^2 + y^2} - R) + z^2 = r^2$ (Oriented on the z axis)
Tori are quartic and can have up to 4 intersections with a ray, so we must use a quartic root finder to find the intersection points. Additionally, finding the normal of a torus is quite complicated, but it can be done by finding the tangent vector along the torus circle, and the tangent of the cross-section circle, and the cross product of those vectors gives the normal.
The lua parser needs a simple extension to be able to construct these new primitives, but will be very similar to the existing cube and sphere ones.

3.2 Texture Mapping and Bump Mapping

We will first need to extend the lua parser to allow materials to have a bump map and a texture map as part of them. These are both images, and are used in different but similar ways.

To do either mapping, we will need a way to take an intersection point on a primitive and map it to a specific pixel on the map image. This is simple for polygons, but much more difficult for primitives like cones and spheres. Each primitive will need a function that can do this mapping. To map a point on a conic into x, y coordinates for the map, the angles between a set vector, and the intersection point vector, and some tricky trigonometry and arclength equations can be used.

Once we have x and y coordinates, we can lookup that coordinate in the maps. For texture mapping, we simple set the colour of the point to be that of the colour in the map. For bump mapping, we use the RGB values of the bump map to perturb the normal of that point in x, y, and z.

3.3 Constructive Solid Geometry

Once again we must define new functions for our lua scripts so that we can make boolean operation nodes that are the cornerstone of CSG. The boolean nodes will take two nodes as children and perform a boolean operation on them - either intersection, union, or difference.

In order to do a ray intersection with boolean node, we must do a ray intersect with both children, but instead of returning a single intersection point we return an entire line segment (or multiple segments in the case of a torus), then the boolean node will perform the boolean operation on the segments to determine the final intersection point of the object.

We also have to be careful because in the case of a difference, we may have to flip the direction of the normal for shading.

3.4 Glossy Reflections

Mirror reflections can be implemented be recursively ray tracing. When a ray hits a shiny object, we cast a ray in the mirror direction and add some fraction of those results to the lighting calculation of the object originally hit.

Glossy reflections are a simple extension to mirror reflections. Instead of casting the secondary ray in the mirror direction, we first perturb the ray slightly at random. The result is a glossy looking reflection.

3.5 Refraction

Refraction is implemented somewhat similarily to reflection. When the ray hits a refractive object, we cast a new ray in a slightly modified direction. The new direction vector can be computed with snell's law: $n_1 \sin\theta_1 = n_2 \sin\theta_2$. This equation is performed for each dimension, relative to the normal of the object at the intersection point.

Since we have an epsilon to check that our shadow rays don't intersect with the object they are on, we also need to make sure that this won't affect our refracted rays. The refracted ray will intersect with the refractive object again when it leaves, so we should make sure that this will still work for very thin clear objects.

3.6 Hierarchical Bounding Boxes

Hierarchical bounding boxes are an extension to the bounding boxes used in A4. Instead of having a non-hierarchical axis-aligned bounding box around primitives, we can extend this to have bounding boxes placed arbitrarily throughout the scene graph. Further, these boxes can be rotated, and have non-uniform scales applied to them.

Since we can place these bounding boxes arbitrarily, we can have them contain a number of primitives and not just a single one. We can also expand them to produce bounding boxes for objects made through CSG. These bounding boxes reduce the number of intersections that need to be done in the usual case that a ray wouldn't hit any part of the object.

Since we don't want the user to have to specify all the bounding boxes, we will instead place bounding box nodes in the scene hierarchy by doing a pass of the tree before we start rendering. It would seem logical to place bounding boxes above SceneNodes, Polygon Meshes, and CSG objects. It is obvious that bounding boxes for meshes and CSG will reduce the number of intersections that need to be done. Placing bounding boxes around SceneNodes is done since it is common practice to construct complex objects out of primitives, and put them together using a SceneNode so they can be transformed all at once, and instantiated multiple times. It is logical that this grouping of objects would likely benefit from a bounding box.

3.7 Photon Mapping with kD-Trees

Photon mapping is done by, before rendering, casting many rays—photons—away from the lights in the scene. The photons bounce off reflective surfaces until they reach a diffuse surface where they are stored in

a photon map. The photons are also obviously affected by refraction as explained above. Photon mapping, although computationally intensive, can produce effects that regular raytracing cannot.

As mentioned above, when the photon hits a diffuse surface, it must be stored in a photon map. This photon map is a spatial data structure in nature and for the purposes of ray tracing, we want fast insert and fast find-density. To do this we will implement a kD-tree which can efficiently find the density of photons in a close area.

A kD-tree is a k-dimensional tree that subdivides space using hyperplanes. At each level of the tree, space is divided by a hyperplane along an axis, where the axis of division rotates at each level of the tree. To find which plane is used to divide the space, a nieve method can be used to determine the middle of the space with respect to all of the objects in that space already.

3.8 Caustics

Now that we have constructed a photon map, we can use it do get some neat lighting effects. For this project, I plan to do caustics. Caustics are the bright spots of light that appear when light is shone through a curved transparent object like a cylinder or a sphere. This is one of the effects that can not easily be achieved through normal ray tracing.

In order to get caustics, we will modify our lighting calculations by adding in an irradience estimate based on the density of photons in the photon map at the point of intersection for the ray. Since the photon map is stored as a kD-tree, finding the density of photons at any given point is a relatively cheap operation.

4 Bibliography

1. Most information will come directly from the relevant chapters of the provided course notes.
2. http://en.wikipedia.org/wiki/Procedural_texture
3. people.scs.carleton.ca/ mould/courses/3501/procedural.pptx
4. www.cs.ucdavis.edu/ amenta/s06/findnorm.pdf

Objectives:

Full UserID:_____ **Student ID:**_____

___ Additional primitives cylinder, cone, and torus are added.

___ Texture mapping is added.

___ Bump mapping is added.

___ Constructive solid geometry for primitives is added.

___ Glossy Reflections are added.

___ Refraction is added.

___ Hierarchical bounding boxes for improved efficiency.

___ Photon mapping with kD-trees.

___ Caustics using photon mapping.

___ A final scene is modelled as described above.