6 Ray Tracing

6.1 Introduction

This chapter discusses implementation of basic ray tracing capabilities for use by other subsystems of a production renderer. Although ray tracing is not required for a functional renderer, there are many effects that are easier or only possible with ray tracing, including:

- crisp shadows and shadows for transparent objects
- accurate simulation of reflection and refraction
- global illumination algorithms
- photon mapping

In a pure ray tracer, ray tracing is also used for visible surface determination from the camera.

The ray tracing facilities should be exposed in the shading language, adding to the toolset available to shader writers. The set of ray tracing functions may include functions that call the ray tracer more or less directly, along with functions that use the ray tracer to provide some other capability such as photon mapping. Useful ray tracing functions include:

```
color C=trace(point P, vector D);
```

This traces a ray from point P in direction D and returns the colour of the intersected object (if any).

```
float dist=hitdistance(point P, vector D);
```

returns the distance from P to the nearest object in direction D; a large distance is returned if no object is intersected.

```
color unshadowed=transmission(point P1, point P2);
```

returns the extent to which light is blocked between points P1 and P2 by opaque or semi-transparent objects.

6.2 Ray Tracing Basics

Ray tracing is largely a self-descriptive capability: it refers to the process of tracing the path of a ray from a point in a given direction and returning information about the primitive(s) intersected by the ray, if any. In most cases the caller of the tracing function will be interested in the first primitive intersected. A ray is defined by an origin and a (usually normalized) direction vector:

$$P(t) = origin + t * dir;$$

Frequently we will be interested in a ray segment defined by limiting t to an interval `[mind maxd]`.

Most programmers who have attempted to write a renderer have probably written a ray tracer of some sort. Ray tracing is popular as a rendering method because it is relatively simple and straightforward to implement a functional ray tracer. A naive ray tracer loops over all available primitives, testing each one for intersection, and returning the nearest primitive:

```
object *tracescene(point pos, ray; float mind, maxd; float &hitd)
  {
  nearestd=maxd;
  hitobject=NULL;
  for each object in scene
    {
    if (intersect(object,pos,ray, &hitd))
      {
      if(mind<hitd && hitd<nearestd)
        {
        hitobject=object;
        nearestd=hitd;
        }
      }
    }
  return hitobject;
  }
```

where the `intersect()` procedure returns a true result and the hit distance if the ray intersects the object. The intersect routine can be kept simple by restricting the supported object types to those that can be easily tested for intersection, such as convex polygons or even just triangles.

Turning the basic ray tracer described above into something usable in a high-end renderer involves overcoming two shortcomings:

1. The naive ray tracing algorithm tests each ray for intersection with every object in the scene, making it very slow for large scences. One of the first optimizations made to most ray tracers is to reduce the number of ray-object intersection tests by building a data structure that enables a scene to be searched for objects hit by a ray without testing every primitive. The next section of this chapter discusses the merits of several popular acceleration methods.