Simple Ray Tracer

Course: CS3241 – Computer Graphics
Students: Qin Yan, Wu Yinghui
Instructors: Dr. Tan Tiow Seng, Dr. Teh Hung Chuan
School of Computing
National University of Singapore
Lower Kent Ridge Road
Singapore 119260

Abstract

The assignment aims to extend a ray-casting program to a simple ray-tracer. The functionality that is extended includes reflection, refraction, and shadow effects.

Introduction

A ray-tracer is a program that generates images of three-dimensional scenes by trying to simulate the behaviour of light rays in the real world. This algorithm is usually more suitable for relatively highly reflective scenes. The drawback of this algorithm, however, is that it is usually too computationally intensive to be implemented for real-time performance.

Implementation

There are four major parts that needs to be extended in order to turn the provided ray-caster skeleton into a simple ray-tracer:

(i) Parsing of scene script files;
(ii) Calculation of reflected and transmitted ray, and colour intensity;
(iii) Calculation of ray-sphere and ray-box intersection;
(iv) Implementation of recursive ray tracing algorithm.

1. Scene script file parsing (`script.cpp`)

After a keyword of "SPHERE" is encountered and the radius of the sphere is read successfully, a temporary *sphere* object is created with the specified values. This temporary object is then added to the object list with *type* = OBJ_SPHERE.

After a keyword of "BOX" is encountered and the *min*, *max* vectors are read successfully, a temporary *box* object is created. This temporary object is also added to the object list with *type* =OBJ_BOX.

A *do* loop is added to the *readline* method to skip the comment lines in the scene script file which start with a hash sign ("#").

2. Reflection vector, refraction vector, and colour intensity calculations (`calculation.cpp`)

The reflection vector is calculated from the formula

$$newRd = Rd + 2 \cdot (N \bullet (-Rd)) \cdot N \qquad (1)$$

where *Rd* is the direction vector of the incident ray pointing towards the incident point. Furthermore, the condition $Rd \bullet N > SMALL$ is checked beforehand to test whether the ray is shot from inside the object or outside. If the ray is from inside, the normal *N* must be inverted in order to get the correct reflection vector *newRd*.

The refraction vector *newRd* is calculated from the formula

$$n = n_i / n_r,$$
$$newRd = n \cdot Rd + \left[-n \cdot (N \bullet Rd) - \sqrt{1 + n^2 \cdot ((N \bullet Rd)^2 - 1)} \right] \cdot N \qquad (2)$$

where *n* is the relative index of refraction of the two media, *Rd* is the direction vector of the incident ray pointing towards the incident point. Similarly, the coefficient $Rd \bullet N$ should be tested before being used for calculation. Another condition, whether or not the value inside the square root is less than *SMALL*, is tested for complete reflection cases. If so, the method will return 0.

A loop is used to add up all contributions from all the lights to the intersection point. If a point is back facing a light (tested by examining $Rd \bullet N$) or totally in the shadow, the light's contribution to that particular point is not counted. Otherwise, the Phong illumination model is used to calculate the contribution of the light.

3. Ray-sphere and ray-box intersection calculation (`objects.cpp`)

All ray-object intersection calculations are carried out after transforming the ray from global coordinate into local coordinate system.

Ray-sphere intersection is calculated by examining the roots of a quadratic equation constructed as following:

$$A \cdot t^2 + B \cdot t + C = 0,$$
$$A = Rd \bullet Rd, \quad B = 2 \cdot Rd \bullet Ro, \quad C = Ro \bullet Ro - r^2. \qquad (3)$$

where *Rd* is the direction vector of the incident ray, *Ro* is the origin of the ray, and *r* is the radius of the sphere. If there are two real roots for this equation, the smaller positive root will be taken as distance to calculation the near intersection point.

As ray-box intersection is just checking with the three pairs axis-aligned faces of the box repeatedly, an additional method *RayIntersectHelper* is added. This helper method is invoked for three times for each box. If none of the invocations returns 0, the smaller positive coefficient *t* is taken to be the distance in the equation $IntersectionPoint = Ro + t \cdot Rd$.

4. Recursive ray tracing implementation (`ray.cpp`)

When a ray hits an object, reflection vector is first calculated. If the object is transparent, the refraction vector is also added. The new rays are assigned an attenuation factor according to the material defined for the object surface. They are then shot from the incident point but calling the *ShootRay* method recursively. This process will not stop until the tracing has reached the desired ray-tree depth.

Discussion

A number of problems was discovered during the implementation of this simple ray-tracer.

1. Handling complete reflection

When a ray is shot from inside an object, complete reflection may occur when the incident angle is relatively large. In this case, we can no longer calculate the correct refraction angle by applying formula (2). So, the return value of *CalcRefractionVector* method is checked to handle this kind of situations explicitly.

2. Negating the normal as necessary

When a ray shoots from inside an object and is coming out of the object's surface again, it's reflection and refraction rays will not be correct if the surface normal at the incident point is not negated. Therefore, the value $Rd \bullet N$ is checked to ensure a correct normal for the calculation.

3. Round-off errors of floating point calculations

Calculation involving floating numbers are inexact and the round-off error can be quite significant in some cases. For example, when checking a normal's direction against a incident ray, *SMALL*, which is a small positive coefficient, should be used instead of 0. Otherwise, the error can result in lots noise points on the surface transparent spheres.

4. Multi-threaded rendering

The original program appears "not responding" to the operating system during its rendering process. The reason is that the computationally intensive part of rendering stops other components of the program from responding to system messages. Thus, the rendering process is put into a separate thread in order to improve the responsiveness of the application while rendering.

Conclusion

The implemented *Simple Ray Tracer* is based on the provided skeleton code for ray-caster. Improvements are made to support more primitives from the script file, and to generate more realistic ray-traced three-dimensional scenes images. Furthermore, the rendering process is now put into a separate thread than the application so as to resolve the "not responding" error during the rendering process.

Appendix

(1) Source code of the four major parts
(2) Sample scenes rendered