Masters Project

Ray Tracer

Vignesh Kumar Ramalingam
MSc Computer Animation and Visual Effects
2010-2011

August 2011 – National Centre for Computer Animation, Bournemouth University

Ray Tracer

Introduction

Ray tracing is a computer graphics technique for generating an image by tracing the path of light through pixels in an image plane.

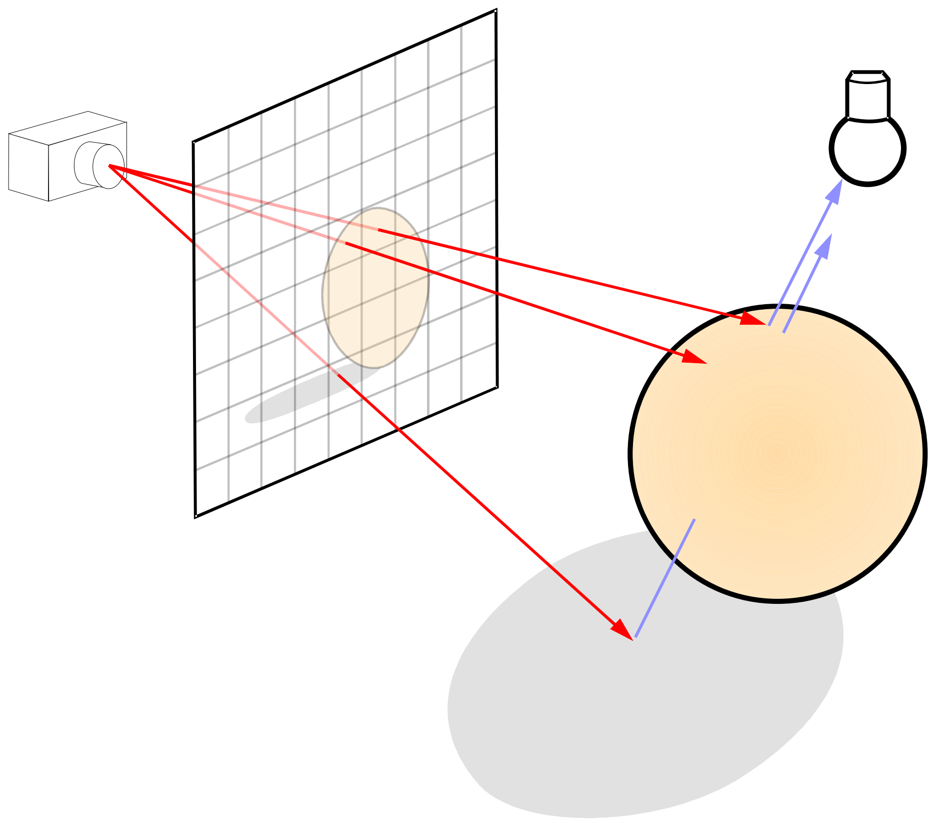


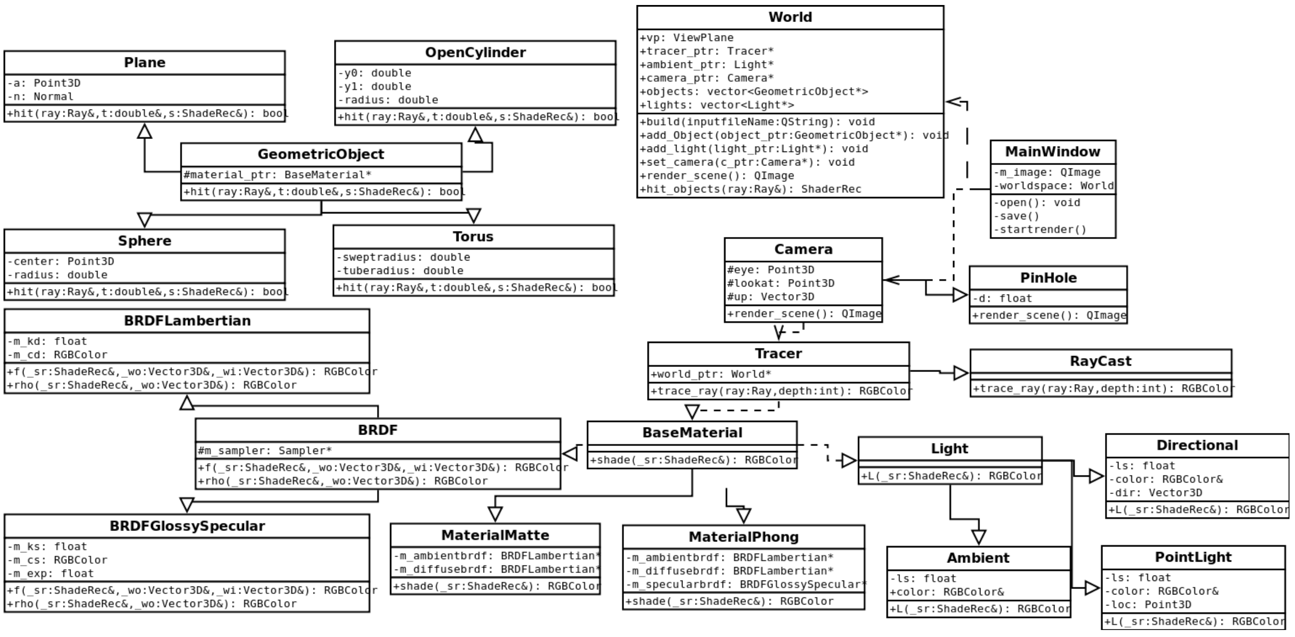
Figure 1.1 A simple scene for ray tracing
(Henrik,2008)

In the above figure, the scene consists of a camera, a view plane, a sphere, a light source. A basic ray tracing technique is where a primary ray starts from a camera, go through each and every pixel of view plane and tests whether an intersection has occurred between the objects (sphere). If a ray hits an object, the ray tracer calculates the colour of the pixel by determining how much light is reflected back along the rays. In an advanced ray tracing technique, if a ray hits an object and the objects has reflective material, the a secondary ray starts from the hit point and tests for an intersection with other objects.

Advantages of ray tracing technique:

- This technique is capable of producing a very high degree of visual realism, usually higher than that of typical scan line rendering methods, but at a greater computational cost.
- This technique is useful for applications where the images can be rendered slowly rather than real time.
- All 3D applications like maya, houdini, 3d max has a ray tracer embedded in it to produce the high quality images.
- Ray tracing is also useful to simulate a wide variety of optical effects, such as reflection, refraction and scattering etc.

Class design


World
+vp: ViewPlane
+tracer_ptr: Tracer*
+ambient_ptr: Light*
+camera_ptr: Camera*
+objects: vector<GeometricObject*>
+lights: vector<Light*>
+build(inputfileName:QString): void
+add_Object(object_ptr:GeometricObject*): void
+add_light(light_ptr:Light*): void
+set_camera(c_ptr:Camera*): void
+render_scene(): QImage
+hit_objects(ray:Ray&): ShaderRec
Plane
-a: Point3D
-n: Normal
+hit(ray:Ray&,t:double&,s:ShadeRec&): bool
OpenCylinder
-y0: double
-y1: double
-radius: double
+hit(ray:Ray&,t:double&,s:ShadeRec&): bool
GeometricObject
#material_ptr: BaseMaterial*
+hit(ray:Ray&,t:double&,s:ShadeRec&): bool
MainWindow
-m_image: QImage
-worldspace: World
-open(): void
-save()
-startrender()
Sphere
-center: Point3D
-radius: double
+hit(ray:Ray&,t:double&,s:ShadeRec&): bool
Torus
-sweptradius: double
-tuberadius: double
+hit(ray:Ray&,t:double&,s:ShadeRec&): bool
Camera
#eye: Point3D
#lookat: Point3D
#up: Vector3D
+render_scene(): QImage
PinHole
-d: float
+render_scene(): QImage
BRDFLambertian
-m_kd: float
-m_cd: RGBColor
+f(_sr:ShadeRec&,_wo:Vector3D&,_wi:Vector3D&): RGBColor
+rho(_sr:ShadeRec&,_wo:Vector3D&): RGBColor
Tracer
+world_ptr: World*
+trace_ray(ray:Ray,depth:int): RGBColor
RayCast
+trace_ray(ray:Ray,depth:int): RGBColor
BRDF
#m_sampler: Sampler*
+f(_sr:ShadeRec&,_wo:Vector3D&,_wi:Vector3D&): RGBColor
+rho(_sr:ShadeRec&,_wo:Vector3D&): RGBColor
BaseMaterial
+shade(_sr:ShadeRec&): RGBColor
Light
+L(_sr:ShadeRec&): RGBColor
Directional
-ls: float
-color: RGBColor&
-dir: Vector3D
+L(_sr:ShadeRec&): RGBColor
BRDFGlossySpecular
-m_ks: float
-m_cs: RGBColor
-m_exp: float
+f(_sr:ShadeRec&,_wo:Vector3D&,_wi:Vector3D&): RGBColor
+rho(_sr:ShadeRec&,_wo:Vector3D&): RGBColor
MaterialMatte
-m_ambientbrdf: BRDFLambertian*
-m_diffusebrdf: BRDFLambertian*
+shade(_sr:ShadeRec&): RGBColor
MaterialPhong
-m_ambientbrdf: BRDFLambertian*
-m_diffusebrdf: BRDFLambertian*
-m_specularbrdf: BRDFGlossySpecular*
+shade(_sr:ShadeRec&): RGBColor
Ambient
-ls: float
+color: RGBColor&
+L(_sr:ShadeRec&): RGBColor
PointLight
-ls: float
-color: RGBColor&
-loc: Point3D
+L(_sr:ShadeRec&): RGBColor

Design

This ray tracing application has been split into four major components,

- User Interface
- World Scene
- Camera
- Lights and Materials

User Interface

When the application is run, the user has to give a input xml file which will be used by the World component to build a scene which has to be ray traced.

SamplexXml format:

```
<!--
This is a sample xml file
-->
<!DOCTYPE RAYTRACERXML>
<raytracer>
  <viewplane>
    <hres>400</hres>
    <vres>400</vres>
    <pixelsize>1</pixelsize>
    <samples>10</samples>
  </viewplane>

  <camera>
    <eye x="25" y="200" z="100"></eye>
    <lookat x="-0.5" y="0" z="0"></lookat>
    <viewdistance>8000</viewdistance>
  </camera>

  <lights>
    <ambientlight>
      <scaleradiance>1.0</scaleradiance>
    </ambientlight>
    <pointlights>
      <pointlight>
        <location x="1" y="5" z="0"></location>
        <scaleradiance>3.0</scaleradiance>
        <shadows>true</shadows>
      </pointlight>
    </pointlights>
  </lights>

  <materials>
    <mattes>
      <matte>
        <name>material1</name>
        <ka>0.25</ka>
        <kd>0.75</kd>
        <cd red="1" green="1" blue="0"></cd>
      </matte>
      <matte>
        <name>material2</name>
        <ka>0.15</ka>
        <kd>0.5</kd>
        <cd red="0" green="0.4" blue="0.2"></cd>
      </matte>
    </mattes>
  </materials>

  <objects>
    <triangles>
```

```
      <triangle>
        <name>triangle1</name>
        <vertexposition1 x="2" y="0.5" z="5"></vertexposition1>
        <vertexposition2 x="2" y="1.5" z="-5"></vertexposition2>
        <vertexposition3 x="-1" y="0" z="-4"></vertexposition3>
        <material>material1</material>
      </triangle>
    </triangles>
  </objects>
</raytracer>
```

This application also has a GUI which is used to display the rendered images. I have added some functionalities for the user like, saving an image, opening an image, zoom in and out of the image etc.

User has to click the Render menu → start render to start the rendering process. This start render method creates an empty world object and calls the build method with user input xml file which constructs the scene to be rendered. We get the pointer to the camera from the world and call render_scene method.

World Scene

This component is the world class which contains the

view plane object which contains the pixel which the primary ray is must pass through ,

tracer object to keep tracking the ray and check for intersections,

a container for lights where all the lights in the scene are stored,

a container for geometric objects where all the primitive objects like sphere, plane, triangle, torus and cylinder are stored.

Ray Object Intersection

The tracer objects has an hit_an_object method which call a hit method for all the geometric objects. This hit method returns true or false based on whether that object has been hit or not.

The method to check whether that object has been hit or not can be easily done by using mathematical formulas. I will give these formulas below,

Plane:

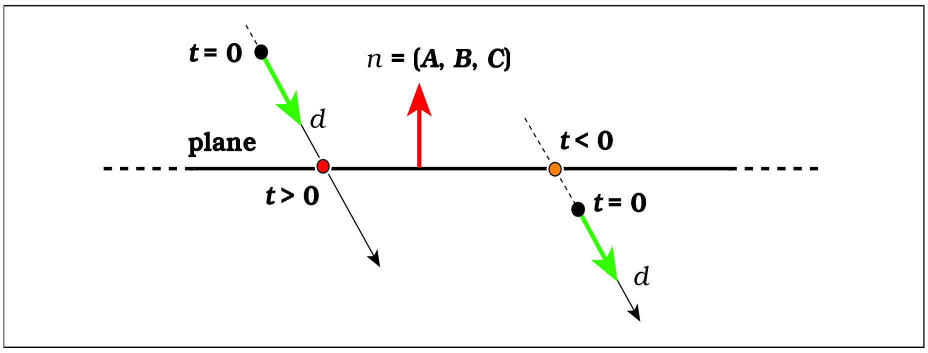


Figure 1.2 ray plane intersection (Suffern, 2007)

formula:

If t > 0.0001 then ray hits the plane where t is calculated from

*t = (a – o) n / (d*n) ,*

In the above formula,

a is a point that lies on the plane,

o is the origin of the ray,

n is the normal to the plane,

d is the direction of the ray.

(Suffern, 2007)

Sphere:

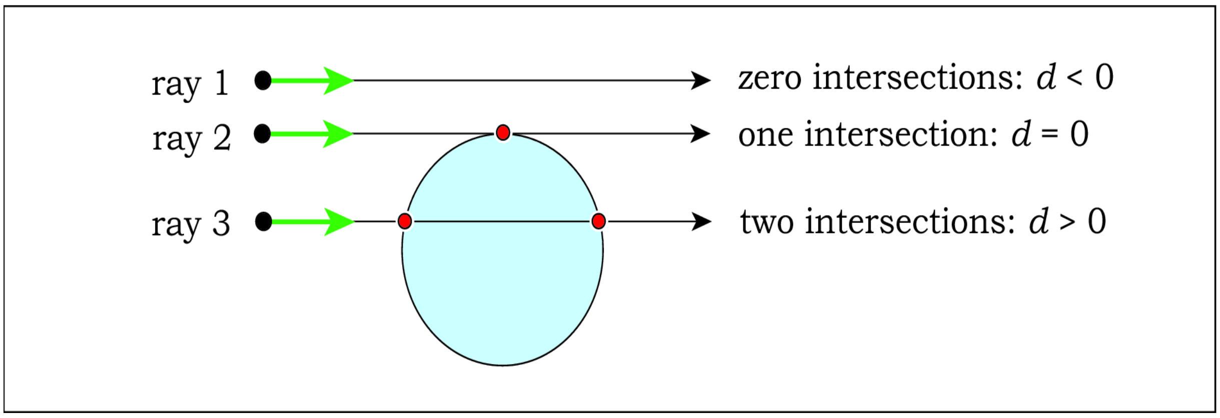


Figure 1.3 ray sphere intersection(Suffern, 2007)

Now consider ,

*a = d * d, where d is the ray direction,*

*b = 2 (o – c)*d, o is the ray origin, c is the center of the sphere,*

c = (o – c)(o – c) - square (r), r is the radius*

d = square(b) – 4ac;

if d < 0 , then ray doesnt hit the sphere.

If d > 0, calculate the t

t = (-b + squareroot(square(b) - 4ac)) / 2a *or*

t = (-b - squareroot(square(b) - 4ac)) / 2a

if t > 0.001 then ray hits the sphere.

(Suffern, 2007)

Triangle:

Triangles are defined by three points a, b, c and has constant normal. Intersection of the triangle is done by first checking whether the ray hits a plane first and if it hits, then check whether its the triangle. This is solved using a bay centric co – ordinates (alpha, beta, gamma) and is explained in detail in the Ray Tracing from ground up book (Suffern, 2007).

t = (a(fl – hj) + b (hi -el) + d (ej -fi)) / (a (fk – gj) + b(gi -ek) +c(ej-fi)

Cyclinder:

Now consider ,

*a = dx * dx + dz * dz, where dx is the ray direction x value, dz is the ray direction z value*

*b = 2 (ox*dx + oz*dz), ox is the ray origin x value,*

*c = ox*ox + oz * oz - square(r) where oz is the ray origin z value*

formula for hit intersection is,

*a*square(t) + b * t + c = 0;*

d = square(b) – 4ac;

if d < 0 , then ray doesnt hit the sphere.

If d > 0, calculate the t

t = (-b + squareroot(square(b) - 4ac)) / 2a *or*

t = (-b - squareroot(square(b) - 4ac)) / 2a

if t > 0.001 then ray hits the sphere.

(Suffern, 2007)

Torus:

formula for hit intersection is,

c4*pow(t , 4) + c3*pow(t , 3) + c2*sqr(t4) + c1*t+ c0 = 0;

where,

c4 = sqr (dx dx + dy* dy + dz* dz) where dx , dy, dz are ray direction x, y, z value.*

c3=4 (dx* dx + dy* dy + dz* dz)*(ox*dx + oy*dy + oz*dz) where ox , oy, oz are ray origin x, y, z value.*

c2=2(dx* dx + dy* dy + dz* dz)*[ox* ox + oy* oy + oz* oz - (a*a + b*b)] + 4*sqr((ox*dx + oy*dy + oz*dz)) + 4 * sqr(a) * sqr(dy), where a is swept radius and b is the tube radius.*

c1=4[ox* ox + oy* oy + oz* oz - (a*a + b*b)]*(ox*dx + oy*dy + oz*dz) + 8*a2oydy*

*c0 = 4*sqr[ox* ox + oy* oy + oz* oz - (a*a + b*b)] – 4*sqr(a)*(sqr(b) - sqr(oy))*

if t > 0.001 then ray hits the sphere. (Suffern, 2007)

Camera

I have created a very simple perspective camera called pinhole camera which takes the following values as input,

- eye_point is an the camera location in the world.
- look_at_point is the view direction.
- up is a vector value for the orientation about the view direction.
- d is the view plane distance from the camera.

Camera class is the one which starts the whole rendering process. Once the world object is built properly, it will have the camera object with the above said values.

Overall ray tracing process

In the main window object, after calling build method for the world object, we get the camera from the world object and call render_scene method available in the camera.

In the render_scene method, we get the resolution of the output image.

```
Loop through the resolution
{
        get the pixel point from the view plane,
        create a ray which starts from the eye_point and goes through each and every
pixel given by view plane.
        Call the trace_method available from the tracer object obtain from the world.
The RayCast tracer object calls hit_objects ( present in the world object) method to
check whether any intersection has happened between the ray and the objects in the
```

scene. If a ray hits any object, we call the shade method available from the material, which will return the radiance (colour) of the pixel.

Save the radiance of the pixel to an image file (Qimage)

return the image.

}

Rays

The ray object has an origin and direction. The origin is the eye_point of the camera and the direction is calculated by the following formula,

d= xv * u + yv * v – dw where u,v,w are the orthonormal vectors.

xv = s * (c – hres/2 + px)

yv = s * (r – vres/2 + py) where s is the pixel size, px, py are the current pixel position and c,r are the current value of the image pixel.

A colour of the pixel on the image is calculated by sampling a number of pixels.

Lights and Materials

Lights

I have a Light class which is a base class which is inherited by Ambient, Directional and Point Light.

The lights are defined by a colour and a radiance scaling factor. So, its easy to get a radiance flux by changing the scaling factor.

Ambient light

This light is used to provide some illumination (very little) to the objects which doesn't have any illumination from the other lights.

Directional Light

Light rays from the directional light are parallel in nature. A direction vector is mandatory for creating a directional light.

Point Light

Light rays from the point light radiates from a specified point in the scene.

Materials

I have a Material class which is the base class for all the material used in this application. The material class is inherited by Matte material class and Phong material class.

Each material class has a shade method, which is used in raycast tracer object to calculate the radiance coming from the light at the hit point of the object. This shade method is not used in Ambient light.

BRDFs

The bidirectional reflectance distribution function (BRDF) provides the way to precisely calculate how the light is reflected at the surfaces. This application has BRDF base class which are used by the materials. It has two methods, f and rho both returns colour with its respective formulas.

Lambertian or Perfect diffuse BRDF

Perfect diffuse reflection is the reflection of light from a surface such that an incident ray is scattered equally in all directions.

Glossy specular BRDF

Glossy specular reflection is where the incoming light is reflected in a broad range of directions.

Matte material

Matte is the simplest material used to shade the ambient and diffuse colour of the object. Matte material has three attributes

ka which is ambient refection coefficient,

kd which is diffuse refection coefficient,

cd which is the diffuse color.

This material creates two Lambertian BRDF with the above values.

Phong Material

Phong material reflects light that concentrated around the direction of mirror reflection. This glossy specular reflection results in specular highlights on the surface, which are the smeared out reflections of light sources. It gives the object an appearance of shiny plastic. Phong material has five attributes

ka which is ambient refection coefficient,

kd which is diffuse refection coefficient,

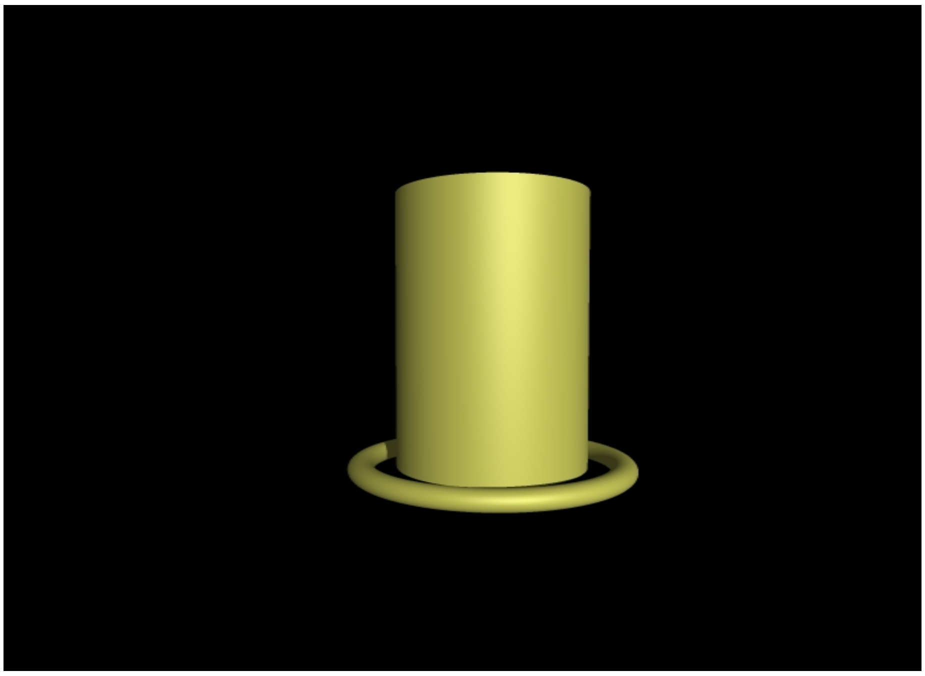
ks which is specular refection coefficient,

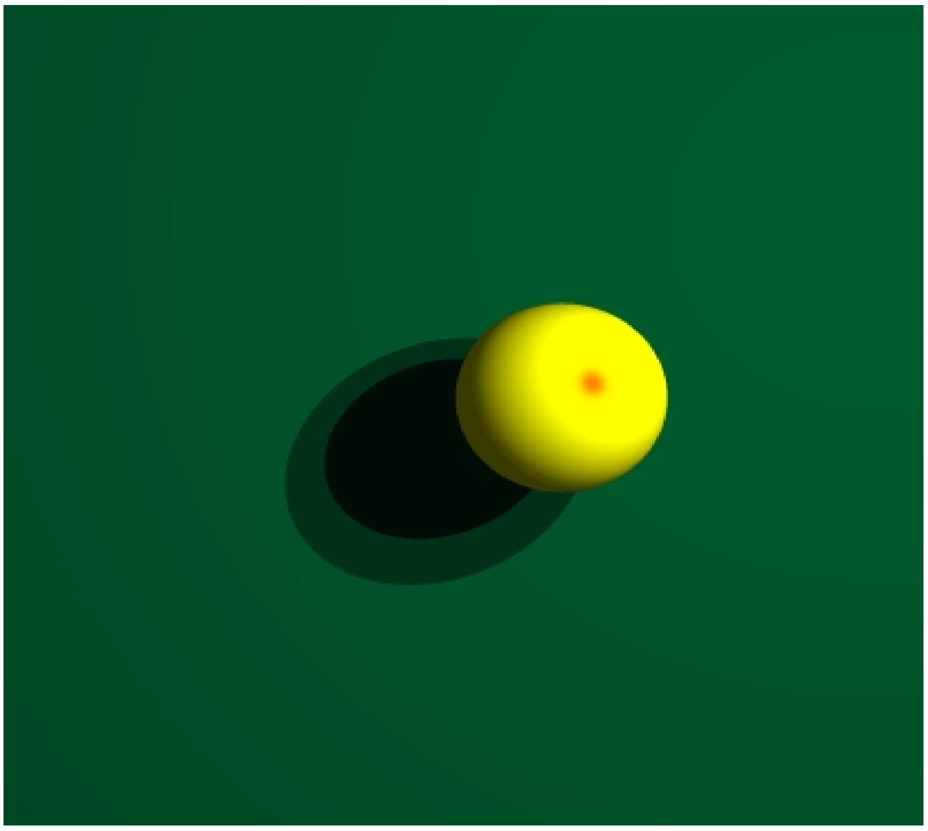
cd which is the diffuse color,

cs which is the specular color.

This material creates two Lambertian BRDF with the values ka, kd, cd and one GlossySpecular BRDF with ks, cs.

Sample Results





Problems faced

The main problem I faced doing this application was ray tracing an object with the triangle meshes. I spent too much time building a structure a to hold the meshes and do the ray hit intersection test. The result was an infinte loop, where it calculates the color for each pixel but never displayed it, So I have to revert back to code without triangle meshes.

Conclusion

The aim of this project is to develop a perspective ray tracer which can support a framework to add any number of lights, materials, geometric objects easily and to implement the basic perspective ray tracig technique.

Future Work

The two main things I want to implement triangle meshes and texture mapping. I would also like to add more primitive objects, materials and lights. I want to also work on Global Illumination, Caustics.

References:

Books:

1. Suffern, K., 2007. *Ray Tracing from the Ground Up.* Wellesley: A K Peters

Web pages:

1. Henrik, 2008. “Ray tracing”.Available from: http://en.wikipedia.org/wiki/Ray_tracing_(graphics) [Accessed 15th August 2011].