Ray Tracer – CmpE492 Senior Project
Midterm Progress Report

UFUK TİRYAKİ

Bogazici University, Computer Engineering
Department. Bebek / İSTANBUL

ufuk.tiryaki@gmail.com

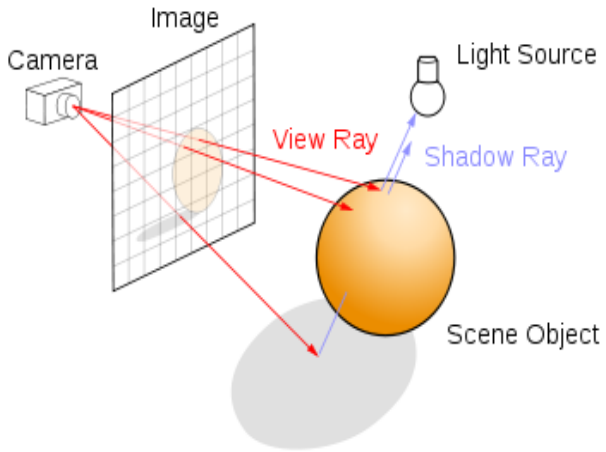
ABSTRACT

In this project, our aim is to prepare a generic scene renderer. Firstly, we have to mention about most popular render engines such as Pixar's renderman, Autodesk's mental ray and the last one V-ray. These are the ones that are frequently used for photorealistic renderings nowadays. Besides that, there are additional eligible engines. Now, we can begin to talk about the technological details of a good render engine. At this point, ray tracing is a key approach which enables us to generate photorealistic images through computers. In fact, physical simulation of the light's behavior needs very high computation power, today. Thereby, our main goal is to imitate real life light's physical behaviors. For the reason, we can evaluate the illumination models as a local and a global illumination. Ray tracing can provide us with both illumination models. Improvements in ray tracing techniques are the ones which stochastically simulate this real life situation as fast as possible.

INTRODUCTION

In this project, we will design and implement a physically based ray tracer. *In computer graphics, ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane. The technique is capable of producing a very high degree of photorealism; usually higher than that of typical scan line rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television special effects, and more poorly suited for real-time applications like computer games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and chromatic aberration .*

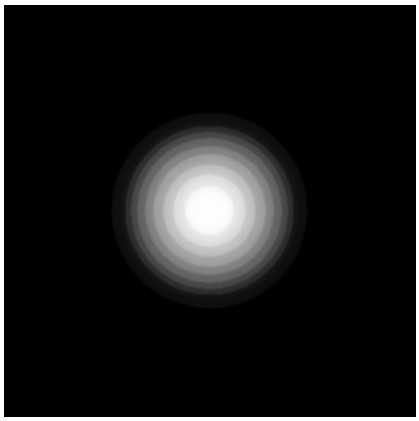
Optical ray tracing describes a method for producing visual images constructed in 3D computer graphics environments, with more photorealism than either ray casting or scan line rendering techniques. It works by tracing a path from an imaginary eye through each pixel in a virtual screen, and calculating the color of the object visible through it. Typically, each ray must be tested for intersection with some subset of all the objects in the scene. Once the nearest object has been identified, the algorithm will estimate the incoming light at the point of intersection, examine the material properties of the object, and combine this information to calculate the final color of the pixel. Certain illumination algorithms and reflective or translucent materials may require more rays to be re-cast into the scene. (R1)



The ray tracing algorithm builds an image by extending rays into a scene. (R1)

Ray Casting

We firstly implement a basic ray caster which generates only primary rays towards the scene and calculate the shading for given point light position. A triangle-mesh sphere is rendered through this technique. We used flat shading to calculate shaded region. Result can be seen below.



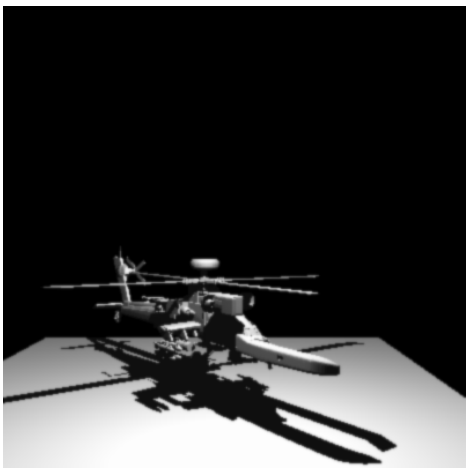
Ray Casting

The first ray casting (versus ray tracing) algorithm used for rendering was presented by Arthur Appel in 1968. The idea behind ray casting is to shoot rays from the eye, one per pixel, and find the closest object blocking the path of that ray – think of an image as a screen-door, with each square in the screen being a pixel. This is then the object the eye normally sees through that pixel. Using the material properties and the effect of the lights in the scene, this algorithm can determine the shading of this object. (R1)

In this project, we used ray/triangle intersection test to determine the triangulated surfaces. We used the *fast, minimum storage ray/triangle intersection* (R2) method that Tomas Möller has found.

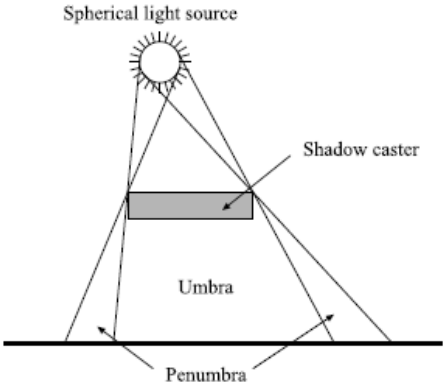
Shadows

After we implemented ray casting algorithm, we started to work on secondary ray calculations. Shadows naturally have sharp edges or soft edges depending on the light or surface's indirect illumination. We assume that our light source is a point light which emits light in all directions. We then implement an additional routine which shoots a shadow ray to test whether the point is in shadow region or not to our ray casting algorithm. A helicopter model is rendered through this technique and the result is seen below.



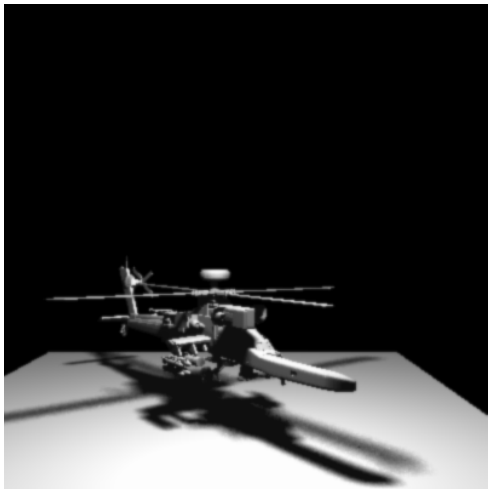
Helicopter is rendered with shadow rays.

We then started to work on soft shadow implementation techniques. Since, we only shoot one shadow ray per pixel to test whether the point is in shadow region or not, shadow borders will not be realistic when the area light comes into play. The following technique solves this problem.



Above, we have a shadow caster and a light source which has a surface area greater than zero. It creates a penumbra on the shadow region. This new sub-region is a transition from light to shadow so that it has an average radiance between full light and full shadow. To calculate this new region properly it is not enough to send one shadow ray per pixel thereby we have implement a technique which enables us to calculate this

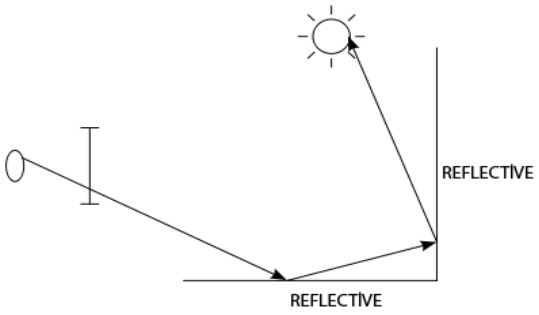
region. The algorithm can be described as follows: we shoot a primary ray to the scene and if the ray intersects a triangle then we shoot more than one rays stochastically to the light source and calculate intensity for each ray then we take an average to get the result. This method is performed for all pixels in the scene so that computation time is very long. We improve our algorithm that it only works for penumbra so that we need to determine full light region and full shadow region. For example, in the first case, we use 128 shadow rays to calculate the shadow. It means that for the 600x600 render, we send 600x600x128 rays to the scene which is very high number of ray intersection. This thoroughly slows down render performance. On the other hand, we only send 16 shadow rays for each pixel whether the region is penumbra or not, if the region is penumbra then we send additional 128 shadow rays to calculate the transition intensity. Below, you may see the result.



Helicopter is rendered with soft shadow.

Reflection

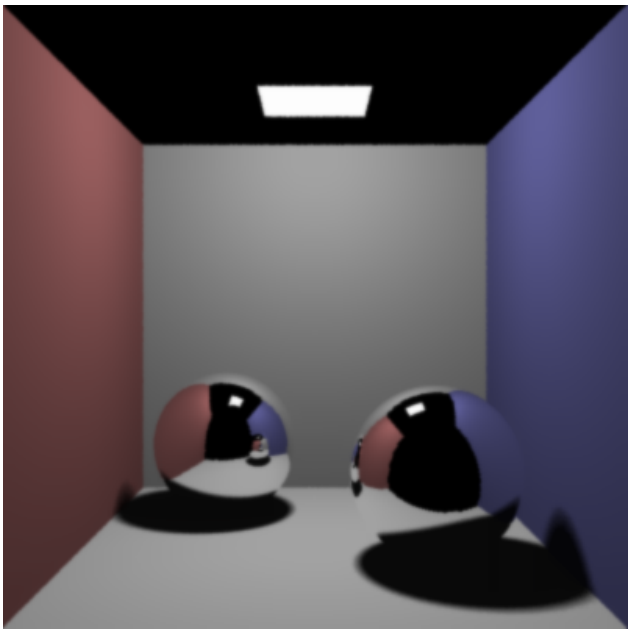
First, a ray is created at an eye point and traced through a pixel and into the scene, where it hits a reflective surface. From that surface the algorithm recursively generates a reflection ray, which traced through the scene, where it hits another reflective surface. Finally, another reflection ray is generated and traced through the scene, where it hits the light source and is absorbed.



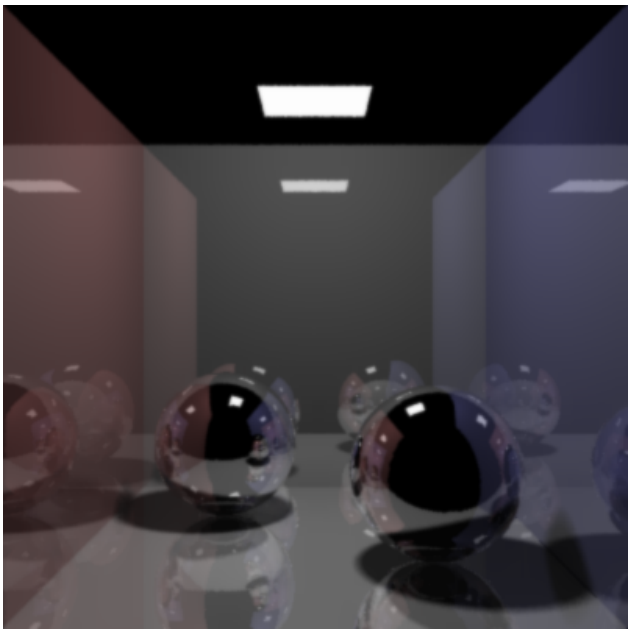
A material has a diffuse color, specular color and ambient color. It also has reflection coefficient and diffuse coefficient where the former plus the latter is less or equal to one. In the following lines you may see pseudo code for the ray trace loop.

```
Radiance illumination(Ray r, Intersection i, int
depth)
{
   Radiance diff = diff_color(r,i);
   Radiance reff;
   Material m = i.hit_point().material();
   if( depth < 5)
   {
        If(m->reflective() == true)
        {
            Ray rr = r.reflect();
           if(intersect(rr,i) == true)
             reff = illumination(rr,i,depth+1);
           else
           {
             if(light->intersect(rr) == true)
               reff = white;
             else
               reff = background;
           }
        }
   }
  return diff*diff_coeff + reff*ref_coeff;
}
```

Next page contains results showing surface reflections and soft shadow together.



Fully reflective balls in a Cornell box.



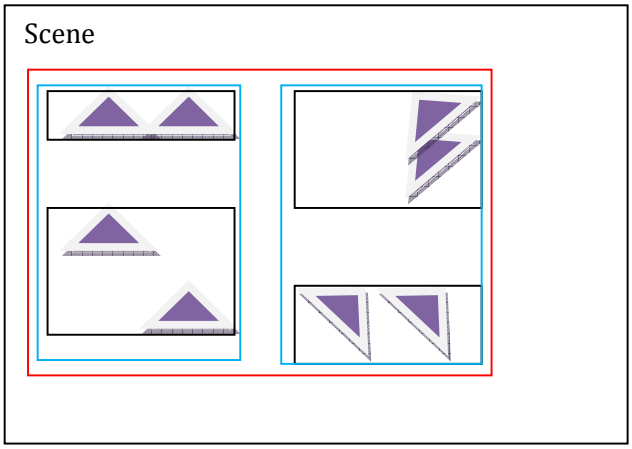
Fully reflective balls in a Cornell box which has half reflective walls.

First render, there are two full reflective balls with diffuse walls. Rays hitting the ball are fully reflected and some of them hit the wall and some of them hit the light directly and rest of them hit nothing. Thereby, we easily can see the reflection of the walls on the balls. Latter image, walls are half reflective with full reflective balls. In this case reflection coefficient of a wall is 0.5 whereas reflection coefficient of a ball is 1.0. In both images, light type is area so that the shadows have soft edges.

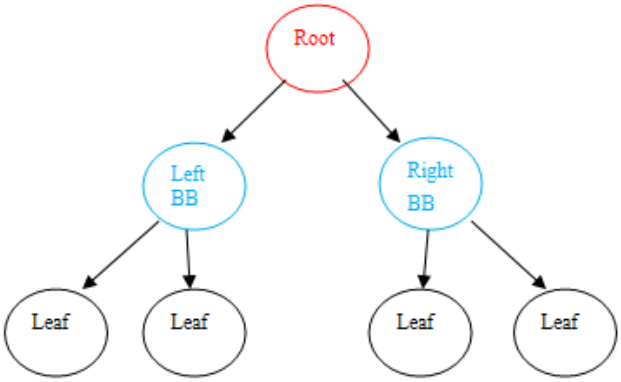
Bounding Volume Hierarchy

Ray intersection can be computationally huge when there are many test objects in the scene. In this project, I prefer triangle mesh models to parametric primitives such as sphere at the start of the project. On the left, spheres in the Cornell box are triangle mesh and each of them contains about 65K triangles. Thereby, the scene consists of nearly 130K triangles. This number shows that, when you send a primary ray to the scene, you need to test 130K triangles to find the accurate triangle. This is computationally expensive so that we have to find another way to detect the primitive in the scene. We need to change the asymptotic behavior of the algorithm from O(n) to O(log(n)). Thereby, we use bounding volume hierarchy to manage the spatial positions of the faces. In the following lines, you may see the approach that we use in this project.

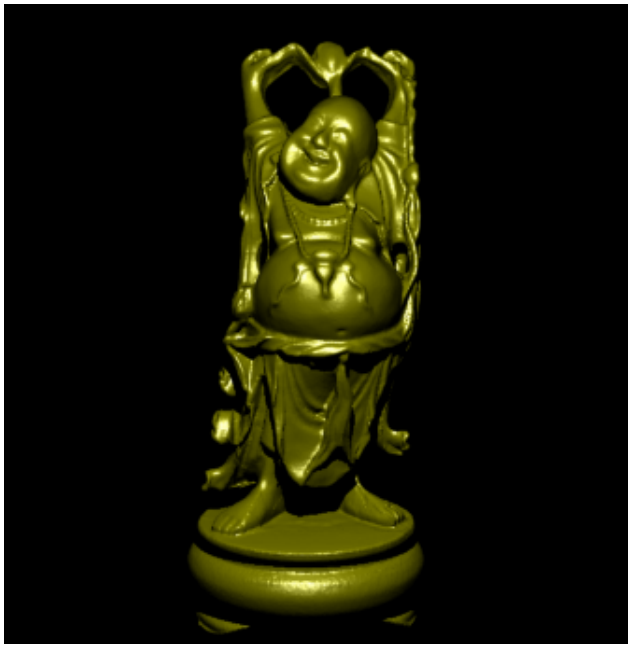
The whole scene consists of triangles having spatial centers. In our acceleration method, we create bounding volumes with at most 2 triangles if the tree depth is less than 20. These volumes are hierarchical from largest bounding volumes to the ones having only triangles. These boxes are also leaves of the binary tree.



You can see the binary tree of the scene. Root is the largest bounding box that contains the whole bounding volumes.



Since we can traverse the tree in a logarithmic time, we can render complex scenes with very high number of triangles. Below, the result can be seen.



Georgia Institute of Technology. Large geometric model archive. Happy Buddha – 1,087,716 faces. It is rendered in 3 seconds with shadow rays (600x600). Phong shading is applied. Intel core2duo T5500, 1GB RAM.

Project Design

After we solve basic problems, we started to work on general structure of the project. We firstly define the scene structure. A scene has the following entities. Set of primitives, Light, Camera.

+ Primitive

Each geometric object in the scene is represented by a Primitive, which collects a lower-level Shape that strictly specifies its geometry, and a Material that describes how light is reflected at points on the surface of the object (e.g. the object's color, whether it has a dull or glossy finish, etc.) All of these geometric primitives are collected into a single aggregate Primitive, aggregate, that stores them in a 3D data structure that makes ray tracing faster by substantially reducing the number of unnecessary ray intersection tests. (R3)

+ Aggregate

Aggregate class can hold many primitives. Although this can just be a convenient way to group geometry, we use this class to implement acceleration structures, which are techniques for avoiding the linear complexity of testing a ray against all n objects in a scene. (R3)

+ Light

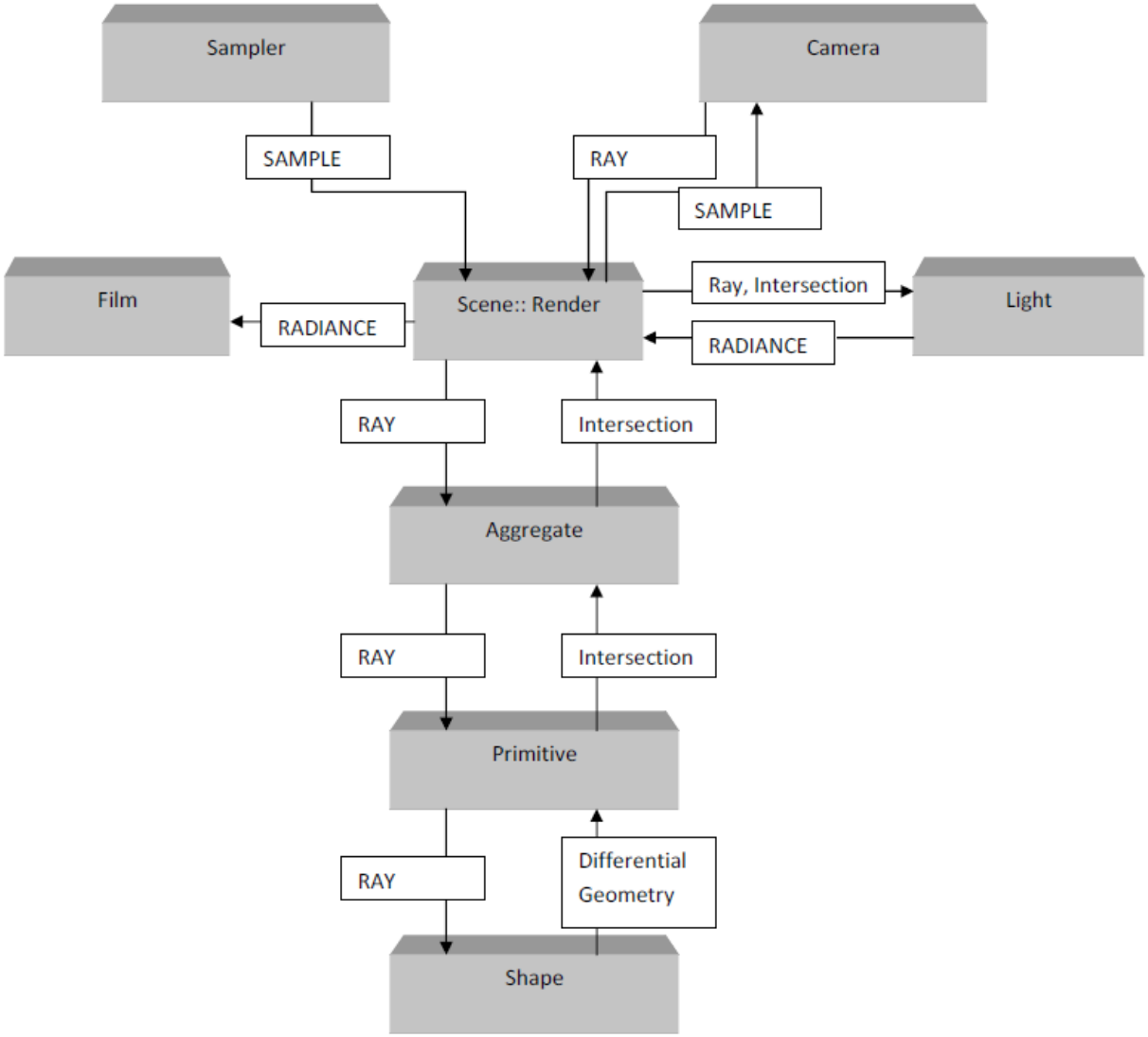
Each light source in the scene is represented by a Light object. The shape of a light and the distribution of light that it emits have a substantial effect on the illumination it casts into the scene. A single global light list holds all of the lights in the scene using the vector class from the standard library. While some renderers support light lists that are specified per-geometric object, allowing some lights to illuminate only some of the objects in the scene, this idea doesn't map well to the physically-based rendering approach, so we only have this global list. (R3)

+ Camera

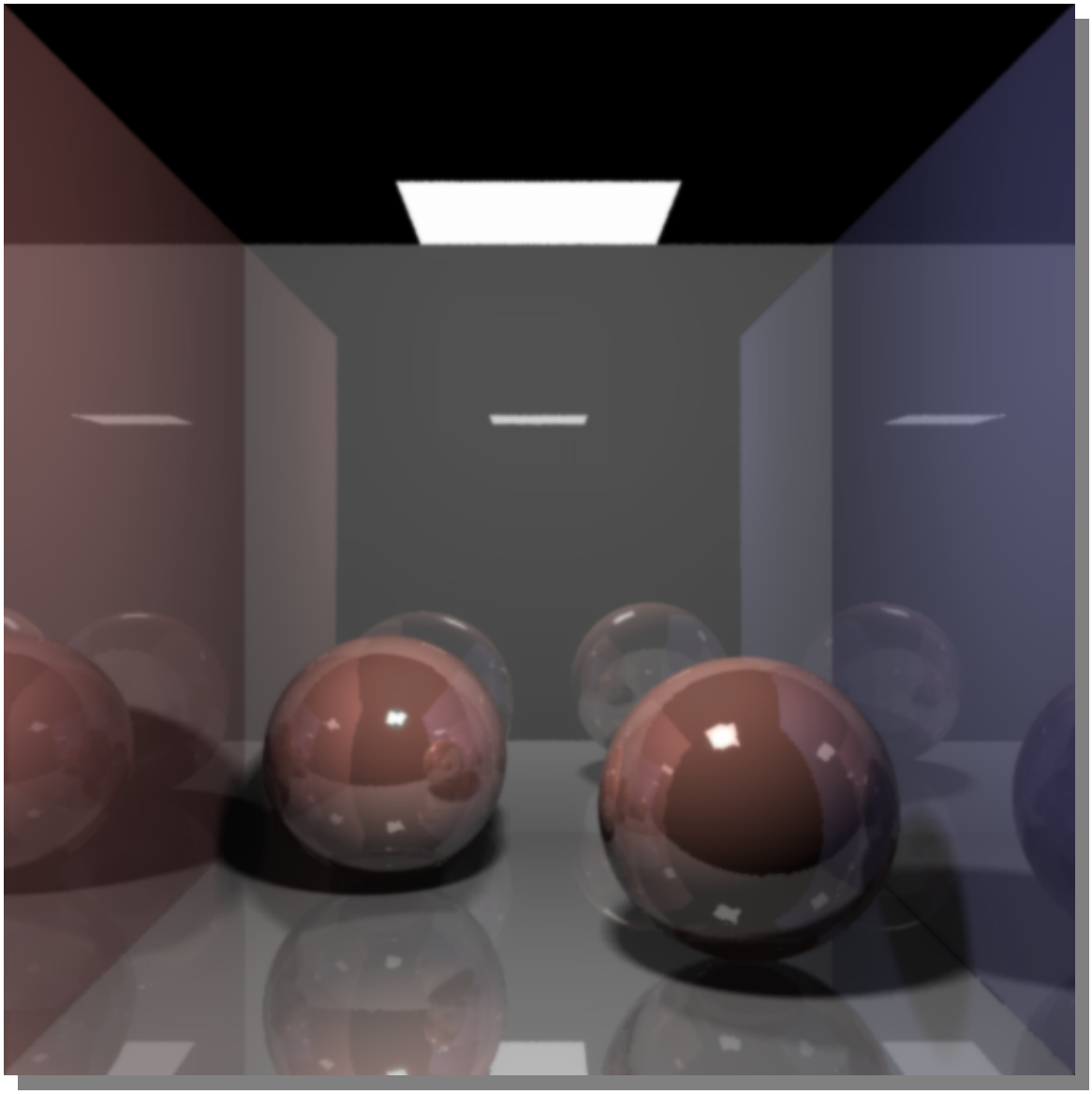
The camera object controls the viewing and lens parameters such as camera position and orientation and field of view. (R3)

Below, you can see the rendering pipeline. First of all, Sampler generates a point on the viewing plane. Through this sample, camera creates a ray beginning from eye point passing through the viewing plane on sample point. After that, ray is casted into the scene then aggregate traverse the binary tree to find out whether the ray hits a triangle or not? If the triangle is found a differential geometry is filled by intersection routine.

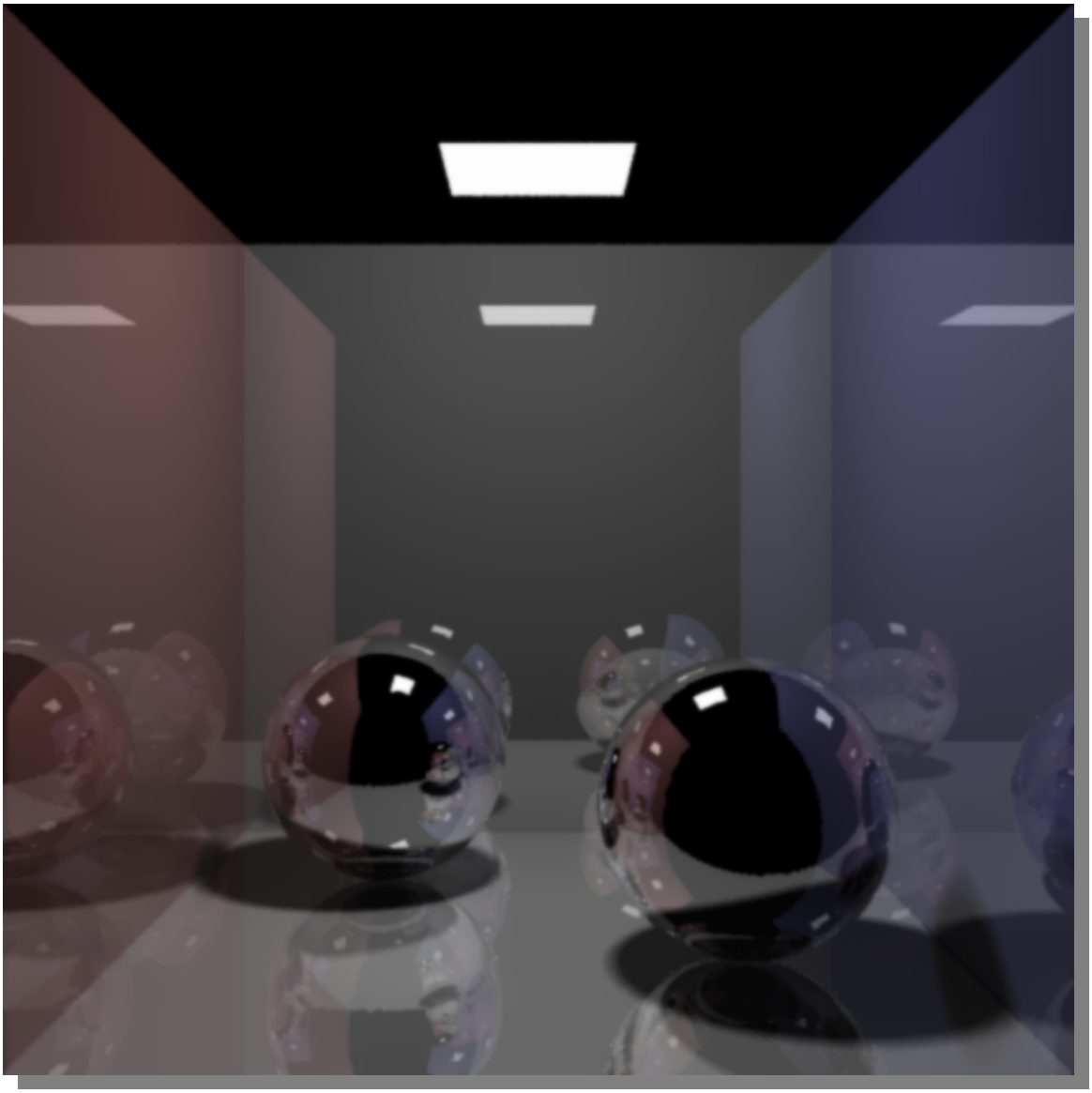
This structure contains local normal and spatial position. After primitive gets the local geometric information it attaches material info to the intersection and send back to the scene::render. When the intersection comes back, it is sent to light to get the radiance. Finally the radiance is used to generate the image buffer.



Renders

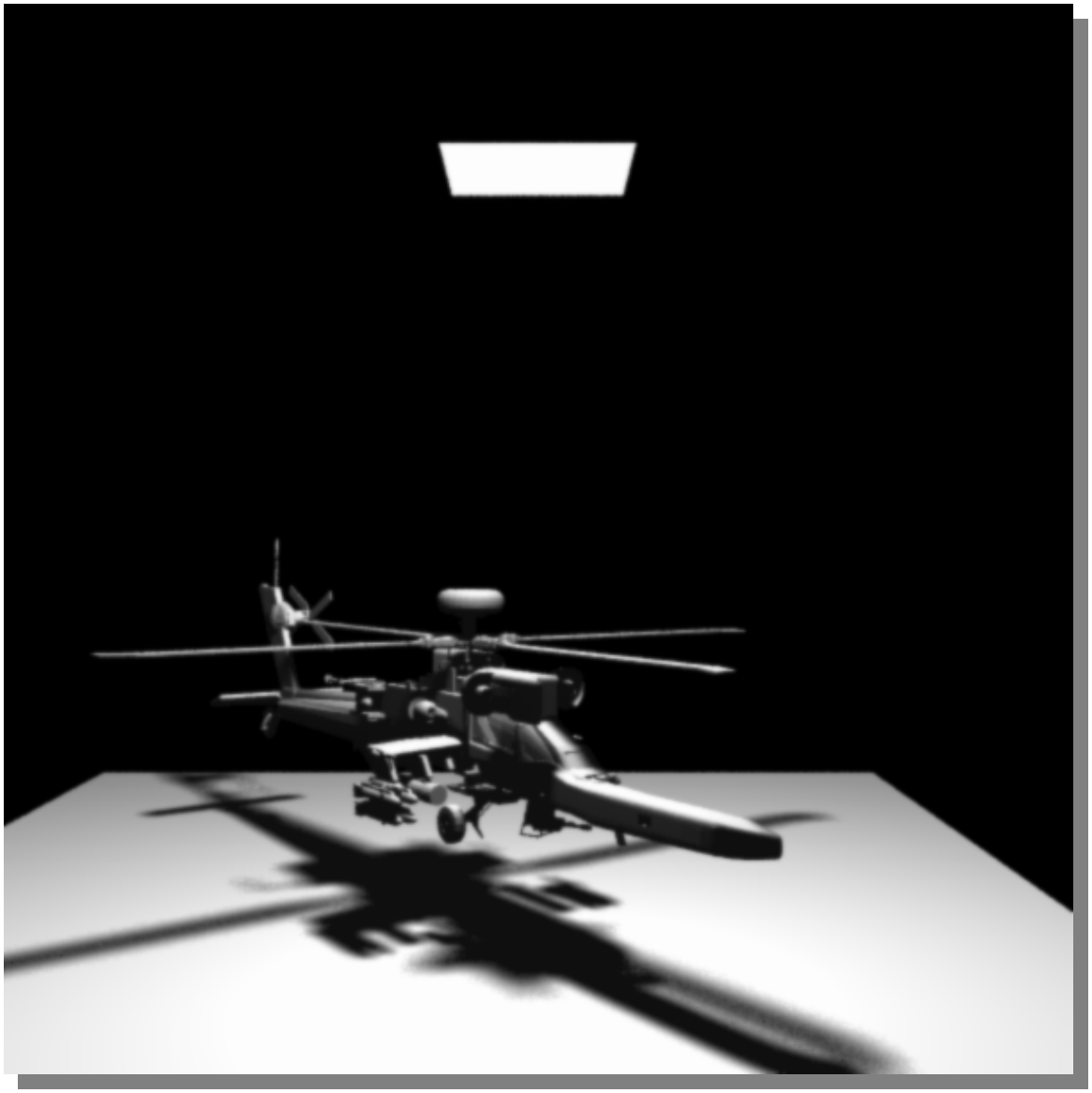


Render Time : 6 min
Face Count : 130K
Light : area light
Anti Aliasing : none
Material
Reflectance Coefficient : 0.5
Transparency Coefficient : 0.0
Diffuse Coefficient : 0.5



Render Time : 12 min
Face Count : 130K
Light : area light
Anti Aliasing : super sampling (2x)
Material
Ball
Reflectance Coefficient : 1.0
Transparency Coefficient : 0.0
Diffuse Coefficient : 0.0

Wall
Reflectance Coefficient : 0.5
Transparency Coefficient : 0.0
Diffuse Coefficient : 0.5



Render Time : 32 min
Face Count : 160K
Light : area light
Anti Aliasing : super sampling (16x)
Material
Reflectance Coefficient : 0.0
Transparency Coefficient : 0.0
Diffuse Coefficient : 1.0

Improvements

We will add the following features.

- Refraction
- Texture Mapping
- Ambient Occlusion
- Anti Aliasing (Efficient method)
- …

REFERENCES

- **R1-**http://en.wikipedia.org/wiki/Ray_tracing_(graphics) - 27th of march 2009
- **R2-** Fast and Minimum Storage Ray/Triangle intersection - Tomas Moller, Ben Trumbore
- **R3-** Morgan Kaufmann - Physically Based Rendering From Theory to Implementation.