Lab Project: Computer Graphics
Building a basic ray tracer
SS2017

Haralambi TODOROV

August 9, 2017

Advisor: Prof. Dr.-Ing. Matthias Teschner

1 Introduction

The main objective of the *Lab project: Computer Graphics* was to make the author familiar with the basic concepts of a ray tracer by implementing one. This lab report aims to present the gathered knowledge.

1.1 Report overview

The report is split into five chapters with, each explaining a certain part of the implemented ray tracer. The ordering of the chapters tries to follow the implementation order the author followed during the project.

Current chapter (1) is an introductory one giving details about the organisation of this report, motivation about why is ray tracing an important rendering technique and where does it find application nowadays, along with some information on the roots of ray tracing, the basic idea of the ray tracing algorithm and some implementation notes regarding the accompanying ray tracer.

Chapter (2) deals with one of the fundamental parts of a ray tracer, the ray-object intersection test. It starts with the geometric definition of a half-line (ray), goes on with the different type of geometries the ray tracer supports (spheres, triangles, axis-aligned boxes and triangulated meshes) and how the half-line intersects with these geometries.

Chapter (3) is concerned with how the camera and the image plane are modelled in the ray tracer. It explains the virtual pinhole camera model, how the image plane is mapped within the 3D scene and the two supported camera projections - orthographic and perspective along with their configurable parameters. The chapter goes on with how rays are shot from the image plane and how one can reduce aliasing artefacts by

using half-jittered sampling technique. It finishes with the use of gamma correction to enhance the visual appearance of the generated images.

Chapter (4) deals with the visual appearance of objects rendered in a scene. It begins by explaining how light sources are modeled, goes on with the concept behind the Phong illumination model and the physical motivation behind its components, how shadows are ray traced and lastly, it presents reflective and refractive materials and how these types of materials are ray traced.

Chapter (5) deals with transformations, explaining the motivation behind the use of homogeneous notation, the different type of supported transformations on objects, light sources and cameras and why the inverse view transformation is useful in a ray tracer.

Chapter (6) is concerned with how one can reduce the computation time per generated image by introducing acceleration structures. Firstly, it explains the concept behind axis-aligned bounding boxes and the benefits they bring, followed by a more advanced acceleration method, the uniform grid.

1.2 Why is ray tracing important

Ray tracing is one of the rendering techniques capable of producing images with a high degree of visual realism. It can naturally incorporate physically motivated visual effects such as reflections, refractions, caustics, soft shadows and others. This advantage makes the technique very attractive to movie and commercial studios, automotive industry (see figure 1a), as well as architectural design studios (see figure 1b) to simulate realistic illumination.

(a) ”Another R8” by Filip Sadlon rendered using Blender Render

(b) ”Mies Van Der Rohe Farnsworth House” by Alessandro Prodan using Mental Ray

Figure 1: Ray tracing used for visualisations by different industries

Although one can produce very stunning imagery with a ray tracing based render engine, this comes at a great computational cost, e.g. a frame from a recent computer generated Pixar’s movie takes between three and eight hours to render. [Goo14]
Despite great computation times, major animation studios seem nowadays to be fond of ray tracing and switch to ray tracing from scanline-based rendering approaches, such as

”REYES”, which have been proved stable and fast over the years to ray tracing. [Pix17] That points to the demands in the entertainment industry for more physically accurate imagery, but also pushes the boundaries of research in ray tracing and its computational efficiency. [ENSB13]

1.3 The roots of ray tracing

The first ray tracing algorithm was introduced by Arthur Appel in 1968 [App68], whose idea was to shoot rays from the eye (camera), one per pixel, and find the closest object blocking the path of that ray. Using the material properties and the effect of the lights in the scene, this algorithm can determine the shading of objects.
The next notable contribution in ray tracing was made by Turner Whitted in 1979 [Whi79], who introduced a technique to compute shadows, as well as the idea of recursive ray tracing to handle reflective and refractive materials. (see figure 2).
Other major contributions in the scene of ray tracing were made by Robert Cook in 1984 [CPC84] and James Kajiya in 1986 [Kaj86], introducing distributed ray tracing and the Rendering equation respectively. Because the accompanying ray tracer does not make extensive use of these concepts, they won't be discussed in detail.

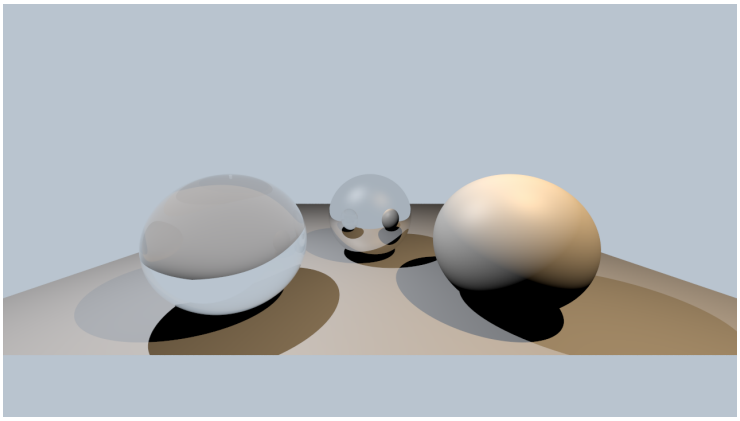


Figure 2: A Whitted-like scene with reflective (back) and refractive (front left) spheres rendered in the accompanying ray tracer

1.4 Basics of the ray tracing algorithm

For the following explanation we make the assumption we have a camera with perspective projection. The algorithm then works by tracing a path from an imaginary eye (camera) through each pixel in a virtual image plane (in front of the camera) and calculating the radiance 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 (see figure 3).

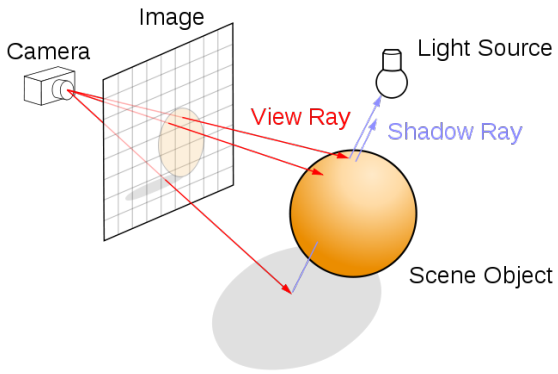


Figure 3: Image depicting the concept behind the ray tracing algorithm. Wikipedia

1.5 Implementation notes

blabla

2 Ray-object intersection tests

The main objective of the current chapter is to present the concepts along with some worth mentioning implementation details about the ray-object intersection tests used in the ray tracer.

To explore ray-object intersections, one has to first define what is an intersection. An intersection is point along a half-line (from now on I'll use the word *ray* meaning half-line) on which given ray intersects certain object. Mostly one is interested in the nearest intersection point along a ray.

2.1 Ray

A ray is geometrically defined using two vectors: a point o, which is called *origin* and a normal vector $\hat{d}$, which is gives the *direction* of the ray. To be able to give the exact location of an intersection point p along the ray, one introduces a scalar value t, which is the distance between the ray's origin o and the intersection point p. One is mainly interested in intersection points "in front" of the ray, so t should have positive values. The parametric equation for a point along a ray looks like: [SH14]

$$p(t) = o + t\hat{d} \tag{1}$$

Within the implemented ray tracer a ray has five data members:

- *origin*
- *direction*
- *inverse of the ray's direction* ($1/\hat{d}$)
- *sign of the ray's direction*
- *type component:* ***primary***, ***shadow***, ***reflection***, ***refraction***

The author has decided not to have an intersection point data member stored within the ray for memory efficiency. Depending on the scene a (big) part of the rays could not have any intersection points at all.
The *inverse of the ray's direction* and *sign of the ray's direction* data members are used to optimize the performance of the ray-axis-aligned bounding box intersection test. Details will be provided on the following section on the topic.
The type component is used for distinguishing between different types of rays and further giving statistic for a given rendered scene.

2.2 Sphere

blabla...

A sphere is defined by two components: a vector c, representing the sphere's center and a scalar value r, representing the sphere's radius. In the implemented ray tracer the author introduces a third component r^2 which is precomputed when constructing the sphere or when applying transformations which alter the sphere's radius. The parameter is later used to optimize ray-sphere computation efficiency.
Figure 4 is of graphical assistance for the algorithm's concept and the author will use the same notation as given on the figure. The goal of the algorithm is to compute the intersection point $p(s - q)$ if the point in front of the ray's origin or $p(s + q)$ if the ray's origin is inside the sphere. On the way the algorithm can terminate, if its sure, that there is no intersection point for the given ray and sphere. The algorithm can be split in three parts: [MHH08]

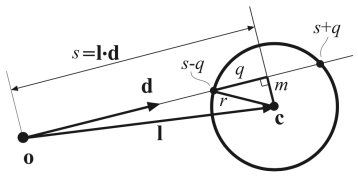


Figure 4: A graphical motivation for the ray-sphere intersection test

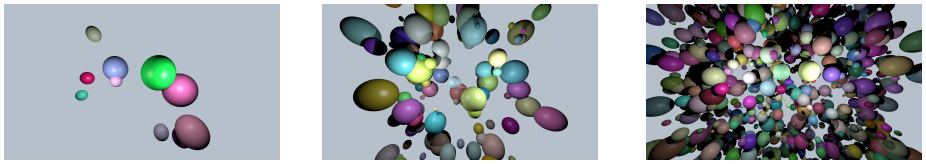
1. This part is concerned with determining if a sphere is behind the ray's origin. If that is the case, one don't need to do further calculations and can exit the procedure for ray-sphere intersection. To do so first the vector $l = c - o$ and its squared length $l^2 = l \bullet l$ are computed. Further one computes the projection of l onto the ray's direction d, $s = l \bullet d$. If $l^2 > r^2$ (the origin of the ray is outside the sphere) and $s < 0$ (ray's direction and the vector l point in opposite directions) one can state, that the sphere is behind the ray's origin $\Rightarrow$ there is no intersection, else one can proceed with the second part of the algorithm.

2. This part is concerned with determining if the ray misses sphere. Similarly as in the first part, if that is the case, one can jump out of the procedure. First a triangle with sides $|l|$, s and m is constructed. The values of l^2 and s are already computed in the last part. One can compute $m^2 = l^2 - s \cdot s$ using the Pythagorean theorem. If $m^2 > r^2$ one can state, that the ray misses the sphere $\Rightarrow$ there is no intersection, else one can proceed with the last part of the algorithm.

3. In this part one computes the actual intersection point. To do so, a triangle with sides r, m and q is constructed. Using again the Pythagorean theorem, one can compute $q = \sqrt{r^2 - m^2}$. If $l^2 > r^2$ the sphere is in front of the ray, so the intersection point is $p(s-q)$, otherwise the ray's origin is inside the sphere and the closest intersection is $p(s+q)$.

Later for shading purposes one would need to determine the surface normal $\hat{sn}$ of a sphere's intersection point p. That is easily done knowing the sphere's center c:

$$\hat{sn} = \frac{p - c}{|p - c|} \tag{2}$$



(a) 10 spheres (b) 100 spheres (c) 1000 spheres

Figure 5: Sphere renderers

Characteristic	10 spheres	100 spheres	1000 spheres
primary rays		14,745,600	
shadow rays	2,844,918	11,097,272	23,438,134
ray-sphere intersection tests	175,905,180	2,584,287,200	38,183,734,000
ray-sphere intersections	1,885,914	9,110,632	28,189,459
render time	4 s	44 s	681 s

Table 1: Rendering information of the images at fig. 5

With the given algorithm one was able to render spheres (see figure 5). The following renderers were done at 1280x720 resolution with 16 samples per pixel without using any acceleration structures. For shading the Phong illumination model was used incorporating shadows as well. There are two light sources

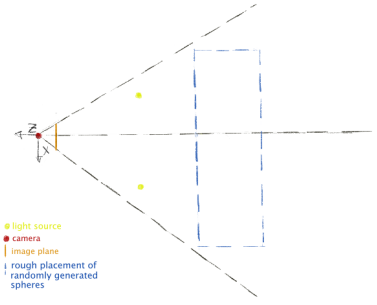


Figure 6: A sketch showing the scene set-up for the renderers at fig. 5

in the scene placed slightly left and slightly right in the front of the mound of spheres (see figure 6). Table 1 provides some rendering information. Using this initial rendering information the author was able to discover a bug, namely in the count of ray-sphere intersection tests, which values for the scenes with 100 and 1000 spheres were too close. The value for ray-sphere intersection tests for the given scenes could be easily calculated analytically:

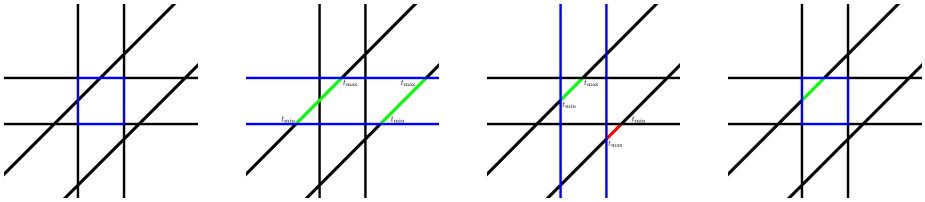
*ray-sphere intersection tests = (primary rays + shadow rays) * # of spheres*

The ray tracer used *uint32_t* ($2^{32} = 4,294,967,295$) from the standard *C++* types to represent this value. Although the author was firstly unaware that this value can be easily exceeded having a big enough scene, seeing the numbers convinced him. Currently the ray tracer uses *uint64_t* to represent rendering information such as the number of ray-sphere intersection tests.
Making the assumption that the whole render time was spent doing ray-sphere intersections, one can give an average duration of the ray-sphere intersection test, which is roughly 20 nanoseconds.

2.3 Axis-aligned bounding box

Axis-aligned bounding boxes (AABB) are used in ray tracing to bound finite objects. Ray-AABB intersections are usually faster to calculate than exact ray-object intersections, and allow the construction of bounding volume hierarchies (BVHs) which reduce the number of objects that need to be considered for each ray. [Bar11]

An AABB is defined in 3D space by two points: a minimum b_{min} and a maximum b_{max} bound. The bounds define a set of three pairs of parallel to the world coordinate axes planes, which encapsulate the box. The concept behind the ray-AABB intersection algorithm is to clip the ray by each pair of parallel planes, and if any portion of the ray remains, it intersects the AABB. The concept in 2D is illustrated by the following figure 7.



(a) The two pairs of horizontal and vertical lines define the box

(b) The two rays are clipped by the pair of horizontal lines

(c) The two rays are clipped by the pair of vertical lines

(d) The remainder of the left ray intersects he AABB, so there is an intersection

Figure 7: Concept of ray-AABB intersection test

The algorithm the author used in the ray tracer is based on this concept and includes

optimizations relying on IEEE numerical properties of the floating point standard that ensure the intersection test is fast and robust. This optimizations make use of the *ray's inverse direction* $(1/\hat{d})$ and *ray's direction sign* data members. [WBMS05]
Information about the speed gain of the render times one can achieve using AABB compared to those without would be discussed in the chapter on acceleration structures (see ch. 6).

2.4 Triangles

Triangles have become a fundamental component in many graphical applications, because of their properties (memory efficiency as triangle strip [EMX02], fast intersections [MT97], triangulation [Wei17]) . In ray tracing they're naturally preferred because of these properties. Most of the *state of the art* ray tracers support triangles and triangulated meshes.

A triangle is defined by three points (called vertices) $v0$, $v1$ and $v2$ and a surface normal n. One important aspect of triangles, one have to keep in mind when dealing with them is the order in which the vertices are defined (clockwise or counter-clockwise), because of their order depends if the surface normal will point inwards or outwards.

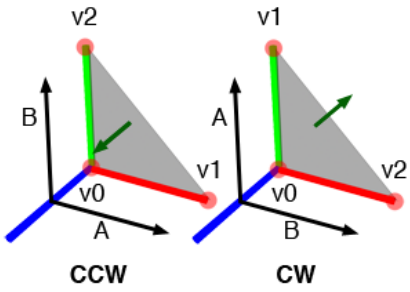


Figure 8: The order of the vertices change the direction of the surface normal. Scratch-a-pixel

3 Cameras and the image plane

4 Shading

5 Transformations

6 Acceleration structures

7 Answers to Definitions

References

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*, pages 37–45, 1968.

[Bar11] Tavian Barnes. Fast, branchless ray/bounding box intersections, 2011.

[CPC84] Robert L. Cook, Thomas K. Porter, and Loren C. Carpenter. Distributed ray tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1984, Minneapolis, Minnesota, USA, July 23-27, 1984*, pages 137–145, 1984.

[EMX02] Regina Estkowski, Joseph S. B. Mitchell, and Xinyu Xiang. Optimal decomposition of polygonal models into triangle strips. In *Proceedings of the Eighteenth Annual Symposium on Computational Geometry*, SCG '02, pages 254–263, New York, NY, USA, 2002. ACM.

[ENSB13] Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. Sorted deferred shading for production path tracing. In *Proceedings of the Eurographics Symposium on Rendering*, EGSR '13, pages 125–132, Aire-la-Ville, Switzerland, Switzerland, 2013. Eurographics Association.

[Goo14] Craig Good. How long does it take to render a Pixar film?, 2014.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1986, Dallas, Texas, USA, August 18-22, 1986*, pages 143–150, 1986.

[MHH08] Tomas Möller, Eric Haines, and Nathaniel Hoffman. *Real-time rendering, 3rd Edition*. Peters, 2008.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graphics, GPU, & Game Tools*, 2(1):21–28, 1997.

[Pix17] Pixar. About RIS, 2017.

[SH14] Kevin Suffern and Helen H. Hu. *Ray Tracing from the Ground Up*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2014.

[WBMS05] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. *J. Graphics Tools*, 10(1):49–54, 2005.

[Wei17] Eric W. Weisstein. Triangulation, 2017.

[Whi79] Turner Whitted. An improved illumination model for shaded display. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1979, Chicago, Illinois, USA, August 8-10, 1979*, page 14, 1979.