Lab Project: Computer Graphics
Building a basic ray tracer
SS2017

Haralambi TODOROV

October 13, 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 through the implementation of 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 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 why is aliasing perceived by the viewer and how can one reduce it by using the 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 shading 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.

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 based 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 present-day computer generated movie by Pixar 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.

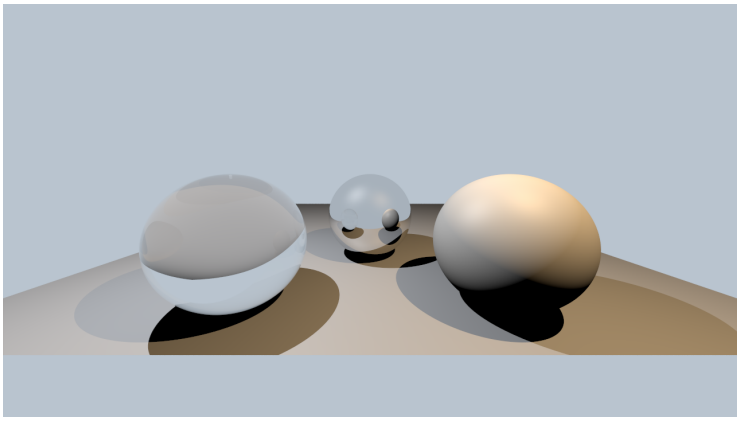


Figure 2: A Whitted-like scene with reflective (back) and refractive (front left) spheres rendered in the accompanying ray tracer

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).

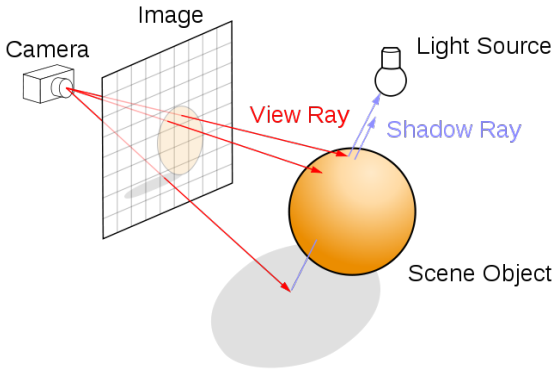


Figure 3: Image depicting the concept behind the ray tracing algorithm. Wikipedia

1.5 Implementation notes

The accompanying ray tracer is developed under *macOS Sierra (version 10.12)* in the *C++* programming language using the "*Apple LLVM version 8.1.0 (clang-802.0.42)*" compiler. The choice of the programming language was based on the adivise of Prof. Dr.-Ing. Matthias Teschner, as well as on the suggestion of many renowned authors, who teach ray tracing, such as Peter Shirley [Shi00] and Kevin G. Suffern [SH14]. Another fact worth mentioning is that many *state of the art* ray tracers are implemented *C++* such as Pixar's *RenderMan* and Solid Angle's *Arnold*.

The author makes use of following external libraries:

- *OpenGL Mathematics (glm)*: a header only C++ mathematics library for graphics software. The library is used to provide robust implementation for manipulations on matrices and vectors.
- *png++*: a C++ wrapper for libpng library. The library is used to save rendered images into **.png* graphics file format.

The actual ray tracer's source code, inside the" *src*" folder, is split into multiple files. One can structure it into seven principal elements, some of which also include derivations. Each of them will be discussed in detail in the following chapters. The following is a high-level overview of the structure:

- *Camera* (see ch. 3): The camera consists of one base class **Camera** and its two derived classes - the supported camera projections **orthographic** and **perspective**. There are described the procedures on how to create a Camera object with a given projection and its desired parameters, how to move it in world space (by applying transformations) and how to render an image using the encoded camera configuration. There is also the procedure on how to apply a view transformation on the objects and light sources in a scene.

- *ImagePlane* (see ch. 3): The image plane class converts a frame buffer object into one of the supported graphics file formats (**.png* or **.ppm*). During the conversion process it encodes a gamma correction on the saved image file.
- *Light* (see ch. 4): The light consists of one base class **Light** and its two derived classes - the supported light types **directional** and **point**. There are described the procedures on how to create a light source object by given type and parameters, how to move it in world space (by applying transformations) and how it illuminates a surface point.
- *Object* (see ch. 2): The object consists of one base class **Object** and its three derived classes - the supported object types **sphere**, **triangle** and **triangle mesh** as well a fourth object type **axis aligned bounding box**, which only serves to accelerate intersection tests and is not being shaded. There are described the procedures on how to create a object by given type and parameters, how to move it in world space (by applying transformations) and whether a given ray intersects this object, and if yes, at which surface point of the object.
- *Ray* (see ch. 2): The ray class describes the procedure of creating a half line object with given parameters in world space and how to trace it within a scene.
- *Utilities*: In the **utilities** header file one stores constants and definitions of structures, which are used on multiple places within the project.
- *main*: The main file serves the purpose to describe a whole scene and to gather statistical informations about a given rendered scene.

2 Ray-object intersection tests

The main objective of the current chapter is to present the concepts along with some noteworthy 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 a point along a half-line (from now on one will use the word *ray* meaning half-line) at 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. In order 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 in the following section on the topic.
The type component is used for distinguishing between different types of rays which are used to give statistic information on a rendered scene.

2.2 Sphere

In the following section one discusses how a sphere is being intersected by a ray. The method used by the author is based on the geometric properties of a sphere.

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 the computation time of a ray-sphere intersection test.
Figure 4 is of graphical assistance for the algorithm's concept. The following notation is the same as on the figure. The goal of the algorithm is to compute the intersection point if the point in front of the ray's origin $p(s - q)$, or if the ray's origin is inside the sphere $p(s + q)$. The algorithm can terminate beforehand, if it is sure that there is no intersection point for the given ray and sphere. The algorithm can be split in three parts: [MHH08]

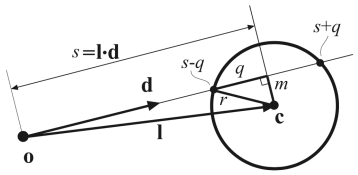


Figure 4: A graphical motivation for the ray-sphere intersection test

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 $\hat{d}$, $s = l \cdot \hat{d}$. If $l^2 > r^2$, (the origin of the ray is outside the sphere) and

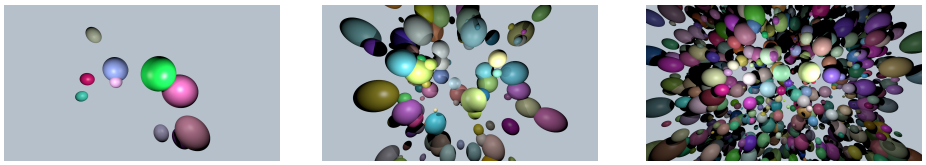
$s < 0$ (ray's direction and the vector l point in opposite directions) one can state that the sphere is behind the ray's origin $\Rightarrow$ there is no intersection, else one can proceed with the second part of the algorithm.

2. This part is concerned with determining if the ray misses the 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 the 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 is able to render spheres (see figure 5). The following renderers were done at 1280*x*720 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 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, whose 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:

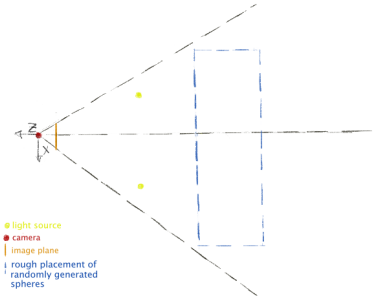


Figure 6: A sketch showing the scene set-up for the renderers at fig. 5

*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.

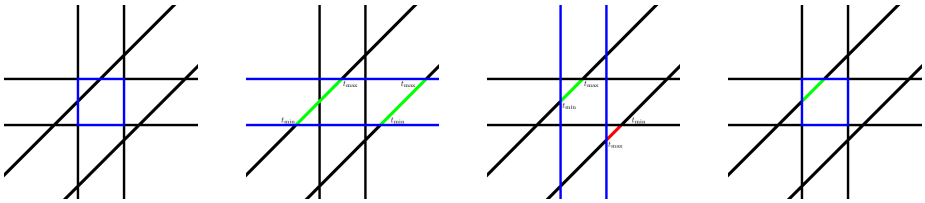
Making the assumption that the whole render time was spent doing ray-sphere intersections, one can give an average duration of the ray-sphere intersection test, which is roughly 20 nanoseconds.

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, called slabs. The concept behind the ray-AABB intersection algorithm is to clip the ray by each pair slabs, and if any portion of the ray remains, it intersects the AABB. The concept in 2D is illustrated by the following figure 7.

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. The optimization relies on the fact that one ray hits multiple AABBs and therefore stores the *ray's inverse direction* ($1/\hat{d}$). One also stores alongside the ray's inverse direction, the sign of the ray's direction. [WBMS05]



(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

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 are 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) v_0, v_1 and v_2 and a surface normal $\hat{n}$. One important aspect of triangles, one have to keep in mind is the order in which the vertices are defined (clockwise or counterclockwise), because of their order depends if the surface normal will point inwards or outwards. In the accompanying ray tracer one computes the surface normal of a triangle during construction, and recomputes it when necessary after a transformation. Surface normals are calculated using counterclockwise order of vertices.

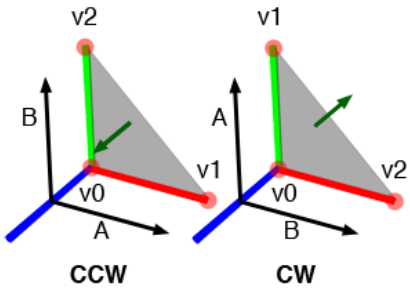


Figure 8: The order of the vertices change the direction of the surface normal. Scratch-a-pixel

One will discuss two methods for intersecting a ray with a triangle.
The first method relies on geometry and is used to intersect a ray with a simple triangle. It only gives information on whether a ray intersects a triangle. This method was implemented initially by the author for the accompanying ray tracer and will be discussed within the current section.
The second method is said to be faster by the computer graphics community [MT97] and prior to finding if a ray intersects a triangle, it also finds the barycentric coordinates

of the intersected point if any. This method was used by the author to find an intersection point for a triangulated mesh and it will be discussed in the next section. Using this method one can easily incorporate surface normal interpolation to give a smooth looking appearance of the mesh.

One way to find if a ray intersects a triangle is to use geometry. An algorithm for solving this problem can be split into two parts:

1. *Finding if a ray intersects the plane formed by the triangle.* One defines the plane formed by the triangle using one of its vertices.

$$\hat{n} \cdot (p - v_0) = 0 \tag{3}$$

   Where p is the point for which we want to find out if it lies on the plane, we substitute p with the parametric equation (see eq. 1) for a point along a ray and get:

$$t = \frac{\hat{n} \cdot (v_0 - o)}{\hat{n} \cdot \hat{d}} \tag{4}$$

   For positive value of t points there is a possible intersection point between the ray and the triangle, else the triangle lies behind the ray's origin. One special case that has to be covered for equation 4 is when the denominator is 0. That is when the dot product of triangle's surface normal $\hat{n}$ and the ray's direction $\hat{d}$ is 0. Geometrically this means that the ray is parallel to the formed plane.

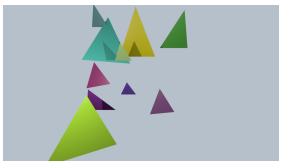
2. *Checking if the point on the plane is within the defined triangle.* To determine if the point $p = o + t\hat{d}$ one has found in the previous step lies on the triangle, one checks if the point p lies within the boundaries created by the edges of the triangle. If this condition is satisfied for all three edges, one can state that the point lies on the triangle.

$$\hat{n} \cdot [(v_1 - v_0) \times (p - v_0)] \geq 0 \tag{5}$$
$$\hat{n} \cdot [(v_2 - v_1) \times (p - v_1)] \geq 0 \tag{6}$$
$$\hat{n} \cdot [(v_0 - v_2) \times (p - v_2)] \geq 0 \tag{7}$$

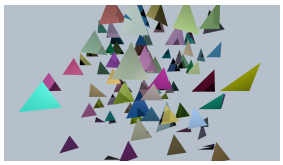
Later for shading purposes one would need to determine the surface normal $\hat{n}$ of a triangle's intersection point $p(u, v)$. The surface normal of a triangle is defined as the cross product of two of its edges:

$$\hat{n} = (v_1 - v_0) \times (v_0 - v_2) \tag{8}$$

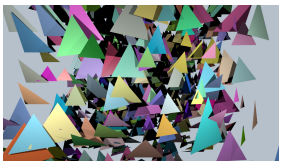
With the given intersection test one is able to render triangles (see figure 9). The following renderers were done with a similar set-up as the one described for the sphere renderers in the above section.
Making the assumption that the whole render time was spent doing ray-triangle intersections, one can give an average duration of the ray-triangle intersection test, which is roughly 19 nanoseconds.



(a) 10 triangles



(b) 100 triangles



(c) 1000 triangles

Figure 9: Triangle renderers

Characteristic	10 triangles	100 triangles	1000 triangles
primary rays		14,745,600	
shadow rays	5,049,906	7,682,896	21,621,572
ray-triangle intersection tests	210,790,689	4,485,699,200	72,734,344,000
ray-triangle intersections	2,715,792	5,509,212	25,111,754
render time	5 s	74 s	1246 s

Table 2: Rendering information of the images at fig. 9

2.5 Triangle mesh

A triangle mesh is a type of polygon mesh. It comprises a set of triangles that are connected by their common edges or corners. Triangle meshes are the preferred type of geometry by many computer graphics applications to represent objects, because of their compactness - the whole object is just a set of triangles. For a ray tracer this fact is very crucial, because one does not have to write an intersection routine for each type of object, but just to convert any given geometry into a triangle mesh, and then intersect it.

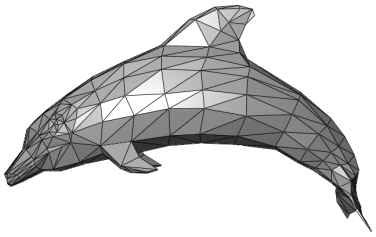


Figure 10: Polygon mesh. Wikipedia

People have developed different types of formats to encode information for triangle meshes over the years, with almost every major 3D software package having its own format. But there also exist open file formats, which are supported by multiple 3D software packages. The file format used to retrieve data for a triangle mesh used in the accompanying ray tracer is such an example - *Wavefront obj*. It is an open file format that has been adopted by many 3D graphics vendors. The file format is a simple data-format that represents 3D geometry alone, namely, the position of each vertex, the corresponding vertex normal and the faces that make each polygon defined as a list of vertices. Vertices are stored in a counter-clockwise order by default. [Wik17d]
The file format can also encode more information for an object (e.g. material information) and even description of whole scenes, but the author has implemented just a

restricted version of it, sufficient to be able to load triangle meshes.

In the accompanying ray tracer one uses an *index arrays* to represent a triangle mesh. With index arrays, a mesh is represented by multiple pairs of separate arrays, e.g. one array holding vertices, and another holding sets of three indices into that array which define a triangle. In the ray tracer two pairs of separate index arrays are used - one containing vertex information and one containing information about vertex normals. The ray tracer does not support textures by the moment of writing this report.

To find an intersection point of a ray with a triangle mesh, one has to simply iterate through all triangles in the mesh and then find the closest intersection, if any. Because triangle meshes could be fairly large in size - more than 50,000 triangles, one has to have a ray-triangle intersection test that is fast and efficient. By the time triangle meshes were implemented in the ray tracer, one decided to search for a ray-triangle intersection test that is faster. One such routine is found in the book *Real-time rendering*. [MHH08] [MT97] In the following is described the basic concept behind this intersection test.

An intersected point p on a triangle is defined using parametric representation based on barycentric coordinates, where u and v are the barycentric coordinates of the point.

$$p(u, v) = (1 - u - v)v_0 + uv_1 + vv_2 \tag{9}$$

$$u \geq 0, v \geq 0, u + v \leq 1 \tag{10}$$

The actual intersection point is computed using a linear system of equations, where one searches for the 2 barycentric coordinates defining the intersection point $p(u, v)$ and the distance between the intersection point and the ray's origin o: t. In the following equations $e_1 = v_1 - v_0$ is the edge between v_1 and v_0 , $e_2 = v_2 - v_0$ the edge is between v_2 and v_0 and $s = o - v_0$ the vector is spanning from the ray's origin o to triangle's vertex v_0.

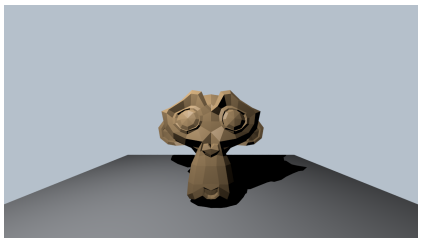
$$o + t\hat{d} = (1 - u - v)v_0 + uv_1 + vv_2 \tag{11}$$

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(d \times e_2) \cdot e_1} \begin{pmatrix} (s \times e_1) \cdot e_2 \\ (d \times e_2) \cdot s \\ (s \times e_1) \cdot d \end{pmatrix} \tag{12}$$

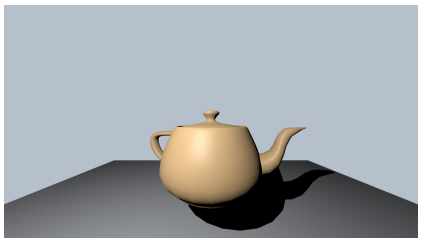
The intuition behind the linear system of equations can also be interpreted geometrically as translating the triangle to the origin and transforming it to a unit triangle in y- and z-axis with the ray direction aligned with the x-axis.

With the given intersection test and the capability to load *obj*-files one is able to render triangle meshes (see figure 12). The following renderers were done with a similar

set-up as the one described for the sphere renderers in the above section. With the difference that AABBs are used to encapsulate the meshes in order to accelerate the render times. Each scene consists of equal number of both types of triangle meshes - *Monkey "Suzanne"* (see figure 11a) and *Utah teapot* (see figure 11b). Information about the mesh's geometry is provided below (see table 3).



(a) Monkey "Suzanne".
Willem-Paul van Overbruggen, Blender



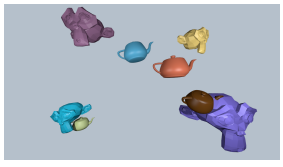
(b) Utah teapot.
Martin Newell, University of Utah

Figure 11: Used triangle meshes

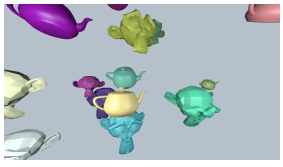
Characteristic	monkey	teapot
# of vertices in the mesh	507	1292
# of triangles in the mesh	968	2464
# of faces in the mesh	500	2464

Table 3: Geometry Information of the used meshes

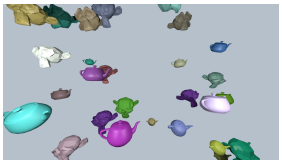
An interesting observation one can make taking a look at the rendering information (see table 4) of the renderers with 16 (figure 12b) and 32 (figure 12c) triangle meshes is that the former has more ray-primitive/triangle intersection tests and due to this has a longer render time. That lies in the fact that the random scale factor used for the render with 16 triangle meshes has a higher range (each object is scaled with a uniformly random factor between 1 and 2 in all directions, compared to the 32 triangle mesh renderer where each object is scaled with a uniformly random factor between 0.5 and 1.5), making the objects take more space in the scene and having more separate rays being tested for intersections with them.



(a) 8 triangle meshes



(b) 16 triangle meshes



(c) 32 triangle meshes

Figure 12: Triangle mesh renderers

Characteristic	8 meshes	16 meshes	32 meshes
primary rays		14,745,600	
shadow rays	4,267,322	7,321,732	6,000,336
ray-prim. intersection tests	16,641,478,672	28,884,298,272	25,675,231,184
ray-prim. intersections	2,576,013	4,822,614	3,747,060
render time	508 s	1006 s	801 s

Table 4: Rendering information of the images at fig. 12

3 Camera and image plane

The main objective of the current chapter is to present the concepts of a camera and an image plane in a ray tracer and how can one discretize a continuous rendered image without loosing too much detail.

3.1 Camera

In a ray tracer one of the primary steps in the process of producing an image is the shooting of (primary/camera) rays from a point (or points) through an image plane into a scene. The point (or points) from which one shoots rays into a scene is called eye of the camera. The image plane plays the role of the camera's film. Both the image plane and the camera's eye model a real camera within a ray tracer.
The way how rays are shot from the camera's eye through the image plane into the scene defines the camera's projection. In the accompanying ray tracer one can use two different camera projections - *perspective* and *orthographic*. One would use the Utah teapot object, introduced in the previous chapter, as a reference to show the differences between renderings of the object using the two projection types.

In the accompanying ray tracer one has a fixed camera position with its eye placed at the origin of the scene's coordinate system, and image plane placed one unit away from the eye in negative z-direction. This is a standard set-up for modelling the camera in simple rendering applications. To move around the scene, one uses view and inverse view transform to place the camera and objects & light sources in the scene respectively.

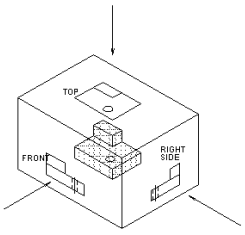


Figure 13: Concept of orthographic projection. Ben Richardson

Orthographic projection

In orthographic projection the primary rays are parallel to each other and orthogonal to the image plane. One of the usages of this projection type is by engineers, who are able to look at a given machine element from different angles without introducing "depth". In orthographic projection distances between points are preserved.

On figure 13 one can see how a machine element is projected using orthographic projection on three image planes - front, right and top.

The way orthographic projection is implemented in the ray tracer consist of two steps:

1. Calculate the position where a primary ray passes through the image plane (cp). This position is dependent on the dimensions set for the rendered image (raster), their aspect ratio and a *zoom factor*.

$$x = (2 * \frac{p_x + 0.5}{r_x}) * \frac{r_x}{r_y} * zf \tag{13}$$

$$y = (1 - 2 * \frac{p_y + 0.5}{r_y}) * zf \tag{14}$$

   Equations 13 and 14 give how is the transformation from the x,y-positions of the raster (p_x, p_y) to the x,y-positions of a primary ray in world space calculated, given the dimensions of the raster (r_x, r_y), as well as the zoom factor (zf).

2. Translate the calculated position one unit backwards along the z-axis to be aligned with the origin of the scene's world coordinate space. Set this position as a origin of the primary ray. The direction of all primary rays point into negative z-axis. $\hat{d} = \begin{pmatrix} 0 & 0 & -1 \end{pmatrix}^\top$

To be able to zoom in and out of a scene using orthographic projection, one has introduced a parameter in the derived class for this camera projection called *zoom factor*. One uses the facts that in world space the image plane occupies a certain amount of space and the camera rays which go through this image plane are orthogonal to it. The concept behind *zoom factor* parameter is that one can space the distance between the origins of the camera rays shot in the scene. Shrinking or expanding the distance one alters the space the image plane occupies in the world space, but the dimensions of the raster do not change and in this way the rendered image contains more or less what is in the scene. Placing the ray's origin points closer to each other restricts the amount of the scene visible through the image plane (perceived as zooming in on the rendered image). Placing them further away from each other increases the amount visible through the image plane (perceived as zooming out on the rendered image).

Perspective projection

Perspective projection simulates the way human eye views a given scene, objects in the distance appear smaller than objects close by - this phenomenon is known as perspective. [Wik17a]
Perspective projection provides a more realistic look into a scene and is used as the default perspective for virtual cameras inside many software packages capable of rendering images (Blender, Autodesk's Maya & 3Ds Max, etc.). On figure 14 one can see how an object is projected using perspective projection.

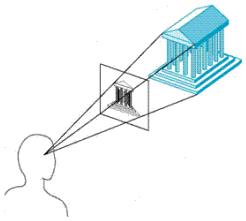


Figure 14: Concept of perspective projection.
Loren K. Rhodes

The way perspective projection is implemented in the ray tracer consist of two steps as well:

1. Calculate the position where a primary ray passes through the image plane (cp). This position is also dependent of the dimensions set for the raster, their aspect ratio and a *scale factor*. The equations used to calculate this position are the same as for the orthographic projection (see eq. 13, 14) except the *zoom factor* is replaced with *scale factor*.

2. All primary rays in perspective projection have the same origin $o = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^\top$, but different directions. The direction for a primary ray in perspective projection is calculated by substituting the the position where the ray passes through the image plane ($cp = \begin{pmatrix} x & y & -1 \end{pmatrix}^\top$) and the ray's origin, and because one uses normalized direction vectors, this vector also has to be normalized: $\hat{d} = \frac{cp-o}{|cp-o|}$.

When talking about perspective camera projection, one have to mention field of view (fov). The field of view is the extend of the scene that is seen by the camera. One distinguishes between horizontal and vertical field of view (see figure 15). Because of the fixed camera set-up that one is using the accompanying ray tracer (image plane is always one unit away from the camera's eye, we have a fixed focal length, distance between eye of the camera and image plane), one can incorporate the field of view to "zoom" in and out of a scene's detail. Using the trigonometry one can use the field of view to scale up or down the image plane. This alternation of the place which the image plane occupy in world space have the same effect as described above for orthographic projection. This is achieved in the ray tracer using a *scale factor* (sf).

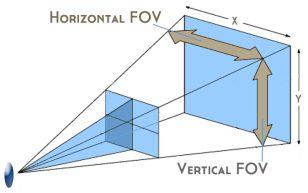
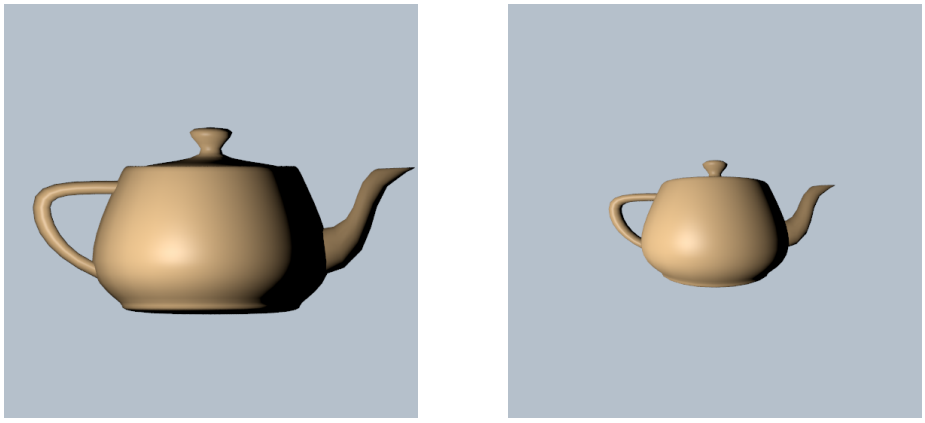


Figure 15: Field of view.
Chris Dawson

$$sf = \tan(\frac{fov}{2}) \tag{15}$$

In the implemented ray tracer one does has just one parameter for the field of view, which is both applied in horizontal and vertical directions of the image plane.



(a) orthographic (b) perspective

Figure 16: Renders of the Utah teapot using orthographic and perspective camera projection

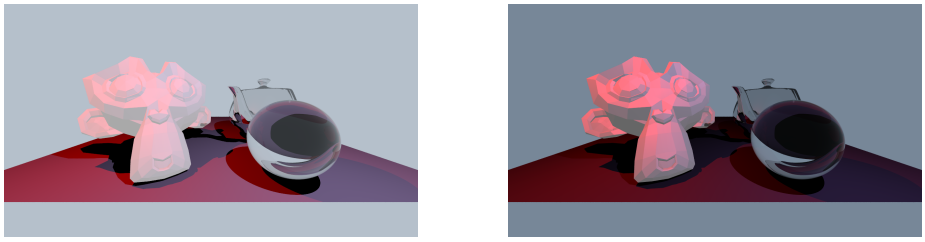
The renders on figure 16 show the Utah teapot rendered using orthographic and perspective projections with the same scene set-up. For the orthographic camera one has used a *zoom factor* 1, and for the perspective camera one has used a field of view $90°$ with a raster of size $550x550$.

3.2 Image plane

The image plane in the ray tracer takes the role of a camera film in a virtual camera set-up. It defines the dimensions, which a rendered image would take in graphics file format using the pixel unit (height and width). Within the implemented ray tracer one has also included the procedures to save a rendered image from a raw framebuffer object (containing just 3 normalized values for each of the RGB components for every pixel) into a graphical file format - *png* or *ppm*.

During the conversion process from a framebuffer object into one of the graphical file formats one encodes gamma to the final rendered image. Gamma is a non-linear operation used to encode and decode luminance or tristimulus values in video or still image systems. [Poy03] In simple words, the human eye can distinguish many more colours in the darker spectrum than in the lighter one and gamma correction make the shown image on display looks more appealing to the human eye. The following function is used to transform the intensity per pixel for each of the RGB-channels. The function is not the gamma function, but the one used to transform from linear space to sRGB, where I is the radiance at a pixel [Kem15]:

$$gamma(I) = \begin{cases} 12.92I & \text{if } I \leq 0.0031308 \\ 1.055I^{\frac{1}{2.4}} - 0.055 & \text{otherwise} \end{cases}$$



(a) with gamma correction

(b) without gamma correction

Figure 17: Renders of a scene with and without gamma correction

3.3 Aliasing

Radiance is a continuous function over the image plane and the ray tracer sample this function at discrete positions. In doing so it only gives an approximation of the rendered scene by trying to reconstruct the original function.
Sampling this continuous function can introduce artefacts, called aliasing. To avoid this one have to choose appropriate samples to represent the original function as good as possible. The sampling technique one uses in the accompanying ray tracer is *half-jittered sampling*. The concept of this sampling technique is to subdivide a pixel's area into $n * n$ strata and to have one sample per stratum, with the sample being closer to the centre of the stratum. On figure 18 one can see how is a pixel subdivided into 5 x 5 strata using half-jittered sampling. All of the rendered images in this report have between 9 to 25 samples per pixel depending on the scene's complexity.

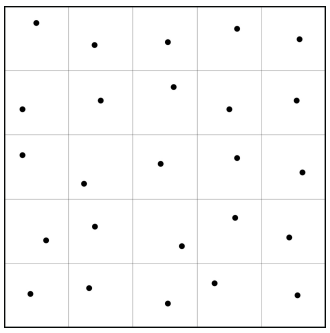


Figure 18: How is a pixel subdivided into strata using stratified sampling.
Kevin Suffern

4 Shading

The main objective of the current chapter is to present how are objects being shaded within the accompanying ray tracer.

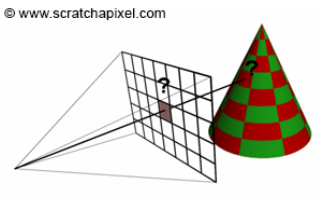


Figure 19: The concept of shading is to compute the radiance value at a pixel position of the image plane Scratch-a-pixel

Shading is one of the key components of a ray tracing application. It is the process of computing or simulating the color of objects as seen from a given viewpoint. In photorealistic rendering one tries to achieve this by modelling object material's surface reflection properties and lightning physically accurate. In the accompanying ray tracer one does not strive for photorealism and uses an empirical reflectance model, the Phong illumination model, which is efficient to evaluate and to some extent also physically accurate. One is also able to render reflective and transparent surfaces using the basic physical properties of these phenomena and the idea of recursive ray tracing.

4.1 Lights

Before presenting the Phong illumination model, one will first introduce the two light types, which are supported in the ray tracer - *directional* and *point* light sources. As well as how one computes shadows cast by those light sources.

In a ray tracer light is a entity whose only function in to indicate where is light emitted from in the scene. If there is no light source in the scene, the scene is rendered black. Light sources have an intensity (which is just a scalar value) and a color (vector containing red, green and blue components of the color). In the ray tracer one have a base *Light* class, which encapsulates this two parameters - intensity and color and two derived classes for the two light types: *Directional light* and *Point light*.

Directional light

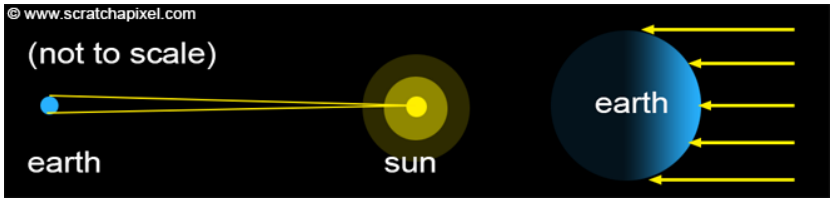


Figure 20: Example of directional light source, the Sun.
Scracth-a-pixel

Directional or distant lights are lights which are considered to be so far away from us

that the light they emit is only reaching us in the form of light rays parallel to each other. With such light sources, all one cares is the direction (this is also the only component of the derived *Directional light* class in the ray tracer) of these light rays. An example of directional light source is the Sun (see figure 20). [Sap17]

Point light

Point light is a type of a spherical light source. Spherical light sources are the most common type of light sources found in nature. To some extent even lights which are not spherical can be somehow approximated as a collection of spherical light sources. To the difference of distant light sources, the position of spherical lights matters. In fact, for spherical light sources, this is the only thing that matters (also the only parameter of the derived *Point light* class).

Figure 21: Example of point light source, a desk lamp.
TurboSquid

If we want to illuminate a point on a object's surface p, then we have to find the direction (ld) of that light ray, where lp is the position of the point light source:

$$ld = p - lp \tag{16}$$

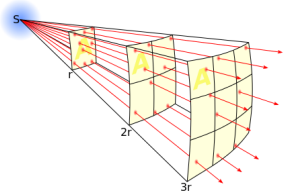


Figure 22: Inverse square law.
Wikipedia

Point lights emit light from a single point in space. Objects which are closer to the source are brighter. This effect can easily be observed with any real point light sources such as a light bulb. The contribution of the light decrease as the distance from the light source to the objects increases. That's due to the Inverse square law which states that a specified physical quantity or intensity is inversely proportional to the square of the distance from the source of that physical quantity. The fundamental cause for this can be understood as geometric dilution corresponding to point-source radiation into three-dimensional space. [Wik17b]
The intensity of the light arriving at point p is inversely proportional to the spherical area (point light sources could be thought as infinitely small spheres), where li is the light's intensity and lc is the light's color and E is the arrived intensity at point p:

$$E = \frac{li * lc}{4\pi |ld|^2} \tag{17}$$

4.2 Shadows

Simulating shadows in a ray tracing application is pretty straightforward (one talks here about hard shadows). All one needs to do is cast a ray from the object visible through a particular pixel of the frame, from the point of intersection to the light source. If this ray which we call a shadow ray intersects an object on its way to the light, then the point that we are shading is in the shadow of that object.
One problem that occurs with this technique of calculating shadows is is self-intersection. Because of some small numerical errors introduced by the fact that numbers can only be represented within a certain precision, sometimes, the intersection point is not exactly directly above the surface, but slightly below. When this happens and that a shadow ray is cast in the light direction, then rather than intersecting no object at all or some other object above the object surface, the shadow intersects the surface from which it is cast. A simple solution to this problem is to systematically displace the origin of the shadow ray in the direction of the surface normal, to move it with above the object's surface. This value is often referred in ray tracing as shadow bias. The value the accompanying ray tracer is using is 10^{-4}, but it could be tweaked depending on the scene. Increasing the bias causes shifts in the shadow position.

4.3 Phong illumination model

Phong reflection is an empirical model of local illumination. It describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces. It is based on Phong's informal observation that shiny surfaces have small intense specular highlights, while dull surfaces have large highlights that fall off more gradually. The model also includes an ambient term to account for the small amount of light that is scattered about the entire scene. [Wik17c]

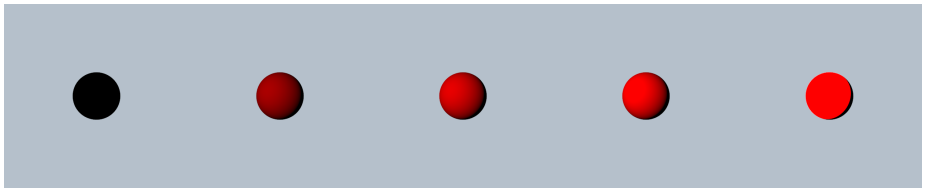
Phong's diffuse component

Objects which have a diffuse surface are often made of materials that exhibit a complex internal structure. Because of this light rays get "trapped" for a while in the object's matter bouncing back many times against the surfaces of that structure before eventually leaving the object. Light rays bounce so many times against these internal structures that their direction when they leave the surface is totally uncorrelated with their incident direction. For this particular reason people like to see diffuse objects as reflecting incident light equally in every possible in the a hemisphere of directions centered around the lit surface point p and oriented about the surface normal $\hat{sn}$ at the point p. Examples of objects which have diffuse material properties are paper and marble. Because diffuse materials reflect light equally in all directions, their brightness does not change with the view direction, which is not the case of objects with specular material surfaces.
The computation of diffuse reflection is governed by the Lambert's cosine law that states the radiant intensity observed from an ideal diffusely reflecting surface is directly proportional to the cosine of the angle θ between the direction of the incident light and the surface normal $\hat{sn}$. One uses following equation to compute the radiance

received at the image plane from a visible object (see equation 18), where $color$ is the color of the object and l is the direction vector from the point on the surface toward a light source. The radiance measured at the image plane is independent from the angle θ, and that's the reason why diffuse reflection is also independent from the viewer direction and the distance.

$$L_{diffuse} = color * E * max(0, \hat{sn} \cdot l) * k_d \tag{18}$$

k_d is material's diffuse reflection coefficient and is a constant between 0 and 1, which varies from material to material. On figure 23 one have rendered a sphere with perfect diffuse material. A point light source is positioned in front of the sphere, slightly above on the left. As one can observe the higher the diffuse reflection coefficient is, the brighter the sphere's surface appear. One can observe on figure 23d that the brightest spot on the sphere is where the sphere's surface normal points to light source. Figure 23e is an example what happens if one uses a value way higher than the given range between 0 and 1. All of the sphere's surface which is lit by the light source appears equally illuminated.



(a) $k_d = 0$ (b) $k_d = 0.3$ (c) $k_d = 0.6$ (d) $k_d = 0.9$ (e) $k_d = 100$

Figure 23: Renders of a sphere with different k_d values

Phong's specular component

Specular reflection emerges when direct light rays reflect from the surface in direction to the camera. Specular reflection creates highlights on the surface. Highlight is a bright spot, which is visible only when light from light source reflects directly to the camera. Specular reflection on an object's surface is visible only if the direction to the camera/viewer v and the direction of the reflected ray r are similar enough (see figure 24).

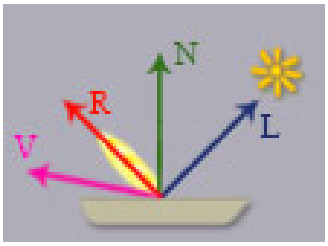


Figure 24: Principle of specular reflection.
Igor Dykhta

To compute the reflection vector r one uses the surface normal $\hat{sn}$ and the direction towards the light source l (on the figure one uses capital letters to denote the vectors, but the author prefers using smaller one to avoid inconveniences, e.g. radiance is denoted also with capital L).

$$r = \frac{2(\hat{sn} \cdot l)\hat{sn} - l}{|2(\hat{sn} \cdot l)\hat{sn} - l|} \tag{19}$$

The radiance received at the image plane due to specular reflection is computed in the ray tracer using following equation, where $color_{specular}$ is the color of the specular highlight (the author decided not to use custom color for every specular highlight, but to hardcode a white highlight, so $color_{specular}$ in the ray tracer corresponds the the RGB value of white) and v is the normalized vector from the surface point towards the camera's eye (viewer direction):

$$L_{specular} = color_{specular} * E * (max(0, v \cdot r))^{se} * k_s \tag{20}$$

se is specular exponent coefficient (or in some places referred as shininess constant). It is an empirical coefficient and the higher its value, the the tighter and smaller the specular highlight appears (see figure 25). k_s is material's specular reflection coefficient and is also an empirical coefficient. On figures 26, 27, 28 one can observe how the specular highlight varies with different values for both coefficients se and k_s.

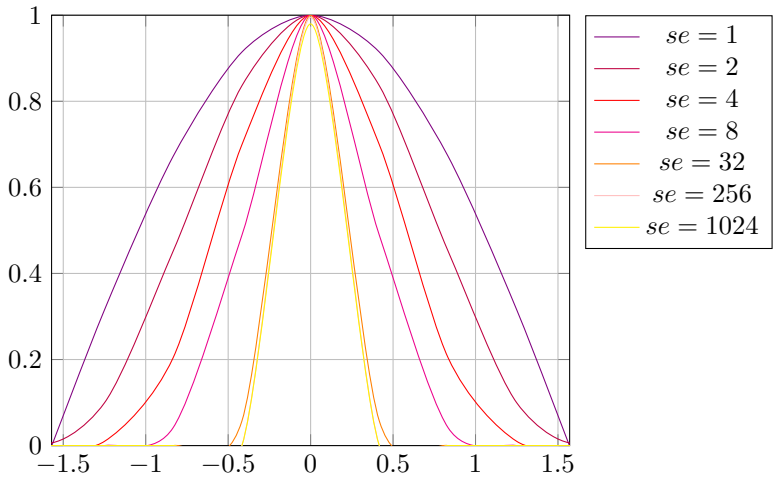
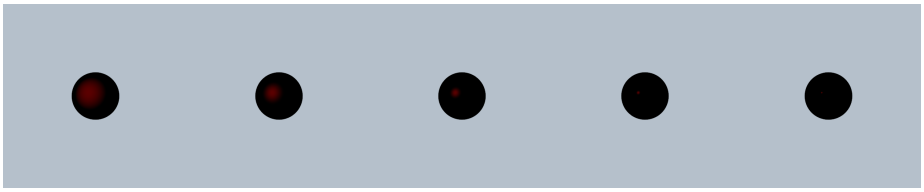
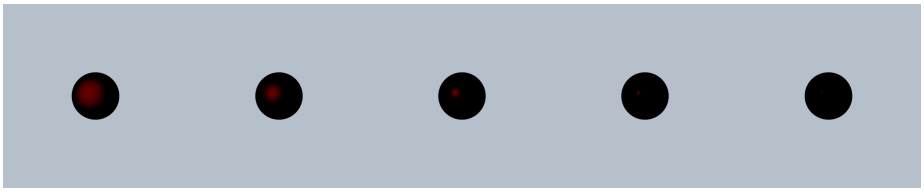


Figure 25: Plot of the function $(v \cdot r)^{se}$



(a) $se = 2$ (b) $se = 8$ (c) $se = 32$ (d) $se = 256$ (e) $se = 1024$

Figure 26: Renders of a sphere with k_s fixed at 0.08 and different se values

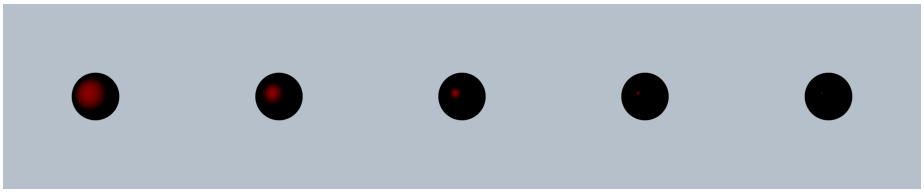


(a) $se = 2$ (b) $se = 8$ (c) $se = 32$ (d) $se = 256$ (e) $se = 1024$

Figure 27: Renders of a sphere with k_s fixed at 0.1 and different se values

Phong's ambient component

The ambient component of the Phong illumination model is motivated by the existence of indirect lightning in the real world. Ambient lighting occurs due to multiple reflections of light rays from different objects. Constant value k_a for ambient lighting gives a good performance to shading the image, but it's very unrealistic. The main disadvantage of such ambient lighting is that all objects will be lightened with same ambient



(a) $se = 2$ (b) $se = 8$ (c) $se = 32$ (d) $se = 256$ (e) $se = 1024$

Figure 28: Renders of a sphere with k_s fixed at 0.2 and different se values

lighting intensity even if objects aren't accessible for direct or reflected light rays. One computes the received radiance at the image plane due to the ambient component using following equation (see equation 21), where $color$ is the defined object's color:

$$L_{ambient} = color * k_a \tag{21}$$

Combining Phong's components

Phong illumination model combines all three components: *ambient*, *diffuse* and *specular* considering an environment with multiple light sources. The equation used in the ray tracer is a bit alternated version of the Phong's original equation. One have just one color for an object, and not as in the original equation, where each of the three components could have its own color. Also as mentioned above, the author has decided for the specular highlight to be only white (which in RGB is the vector $\begin{pmatrix}1 & 1 & 1\end{pmatrix}^T$, thus could be skipped in the equation).

$$L_{Phong} = L_{ambient} + \sum_{m \in lights} E_m(color * max(0, \hat{sn} \cdot l)k_d + (max(0, v \cdot r))^{se} k_s) \tag{22}$$

4.4 Reflection

Reflection is a simple light-matter interaction, where an incident light beam is reflected at the object's surface into a reflection direction $\hat{d}_{reflection}$. An example of a reflective surface is the mirror. Reflection is a view dependent phenomenon. If one looks at the reflection of a static object in the mirror and then changes position, the image of that object would change as well.

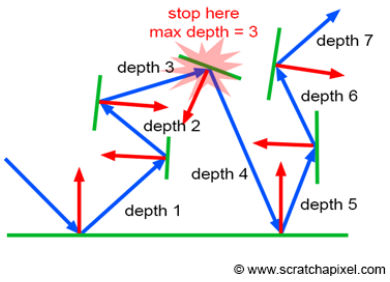


Figure 29: Concept of recursive reflection algorithm.
Scratch-a-pixel

To compute reflections in the accompa-

nying ray tracer one uses the concept of *recursive ray tracing*. [Whi79]
If the object that an incident ray i hits is a mirror like (reflection) surface, one computes the reflection direction $\hat{d}_{reflection}$ of the incident primary ray.

$$\hat{d}_{reflection} = \frac{\hat{d}_{incident} - 2(\hat{sn} \cdot \hat{d}_{incident})\hat{sn}}{|\hat{d}_{incident} - 2(hatsn \cdot \hat{d}_{incident})\hat{sn}|} \tag{23}$$

Then one recursively cast *reflection rays* with origin at the surface normal $\hat{sn}$ in that reflected direction. To avoid self-intersections, similarly as for shadow rays, one slightly displace the origin of the *reflection ray*. And assign to the radiance measured at the primary ray the color gained by the reflection rays. Reflection rays are shot in the scene recursively until a certain *maximum recursion depth* is reached. Because there does not exist a perfect reflection materials in real world which reflect 100% of the incident light, one reduces the amount of reflected radiance at each bounce by 20%. This value is chosen empirically and could be changed in the application to gain more appealing renders. Figure 29 shows the recursion process visually.

4.5 Transparency

When light rays pass from one "transparent" medium to another, they change direction. The new direction of the ray depends on two factors: the angle of incidence and the new medium *refractive index*. This phenomenon is described by what is called the "Snell's law", which states that for a given pair of media, the ratio of the sines of the angle of incidence θ_1 and angle of refraction θ_2 is equivalent to the opposite ratio of the indices of refraction.

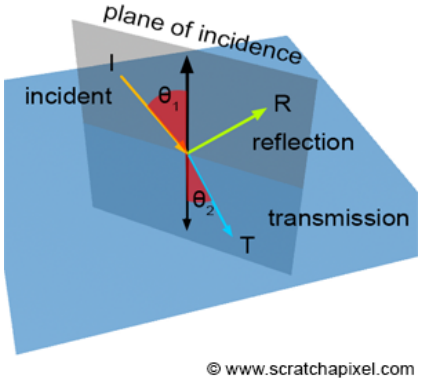


Figure 30: Concept of refraction. Scratch-a-pixel

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{\eta_2}{\eta_1} \tag{24}$$

To compute the direction of refraction/transmission t (see figure 30) one can use the information gained by "Snell's law" law, where $\hat{i}$ is the direction of the incident light ray:

$$t = \frac{\eta_1}{\eta_2}\hat{i} + \hat{sn}(\frac{\eta_1}{\eta_2}\cos\theta_1 - \sqrt{1 - \sin^2\theta_2}) \tag{25}$$

One condition that comes from the above equation for t is that $\sin^2\theta_2 \leq 1$. If this condition is not fulfilled, there's no transmission and one have the phenomenon of

total internal reflection. The incoming angle at which this phenomenon happens is called critical angle θ_c and is given by:

$$\theta_c = \arcsin \frac{\eta_1}{\eta_2} \iff \eta_1 > \eta_2 \tag{26}$$

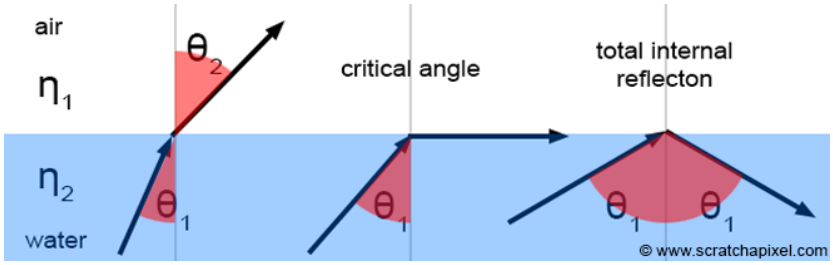


Figure 31: Total internal reflection. Scratch-a-pixel

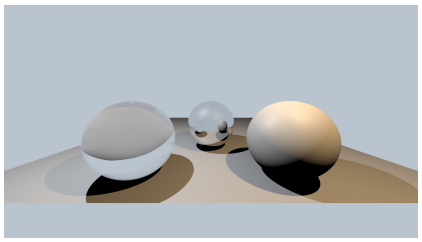
Transparent objects not just only transmit light in direction t, but they also reflect light (as seen on figure 30). That could be explained by the physics of light. When a photon arrives at a transparent surface it has two options: it can either go through the medium by following direction t or it can reflect by following direction $\hat{d}_{reflection}$. Of all photons that arrive at the transparent medium, one part is reflected and the other is transmitted. This parts are calculated by the *transmittance* T and *reflectance* R:

$$T + R = 1$$

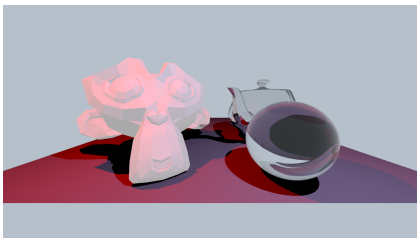
Their amount depend on the refractive indices η_1 and η_2, but also on the angle of incidence θ_1. How exactly their parts are split is governed by the *Fresnel equations*, where $\eta = \frac{\eta_1}{\eta_2}$: [DG04]

$$R = \frac{1}{2}((\frac{\eta \cos\theta_1 - \cos\theta_2}{\eta \cos\theta_1 + \cos\theta_2})^2 + (\frac{\cos\theta_1 - \eta \cos\theta_2}{\cos\theta_1 + \eta \cos\theta_2})^2) \tag{27}$$

$$T = 1 - R \tag{28}$$



(a) Sphere in the front left is transparent and in the back is reflective



(b) Sphere is transparent and teapot in the back is reflective

Figure 32: Renders showing transparent and reflective objects

5 Transformations

The current chapter deals with the different kind of transformations supported by the accompanying ray tracer.

Transformations are an important feature included in almost all of graphic applications. They give one the ability to position, reshape and animate objects, light sources and virtual cameras.
During the development of computer graphics as a field, one has developed the preference of using *homogeneous coordinates*. This coordinate system gives one the ability to represent all types of affine transformations (scale, translation, rotation) with a matrix and points and vectors to be transformed in an unified way with matrix-vector multiplications. Another useful feature of this coordinates is that complex transformations can be encoded in a single matrix. By the chain rule, any sequence of transformations can be multiplied out into a single matrix. A 3D point in homogeneous coordinates is represented: $(x, y, z, w)^T$ with $w \neq 0$, where it corresponds to the 3D point $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})^T$. A vector is represented: $(x, y, z, w)^T$ with $w = 0$.

The way affine transformations work on objects and light sources is by transforming their control points or normals (e.g. *directional light* has just a normal representing its direction) using a transformation matrix. For example a triangle is rotated by rotating each of its vertices individually. A transformation matrix looks like following, where elements m_{ij} are responsible for encoding rotation and scale, t_i are responsible to encode translation, p_i encode projection and w is analogue to the fourth component for points and vectors:

$$\begin{bmatrix} m_{00} & m_{01} & m_{02} & t_0 \\ m_{10} & m_{11} & m_{12} & t_1 \\ m_{20} & m_{21} & m_{22} & t_2 \\ p_0 & p_1 & p_2 & w \end{bmatrix} \tag{29}$$

A useful use of *homogeneous coordinates* is the view transform. In the sense of *homo-*

geneous coordinates a view transform could be thought of a basis transform. Objects and light sources are placed with respect to a (global) coordinate system. The camera is placed in its own camera coordinate space. After a view transform all objects and light sources are represented in the camera coordinate space. Placing and orienting the camera corresponds to the application of inverse transform to objects and light sources.

Normals in *homogeneous coordinates* are transformed by the transpose of the inverse of the transformation matrix we want to apply.

To implement the different type of transformations one uses the *glm*-library, which provides an efficient and easy to use API to deal with transforming objects, light sources and cameras.

6 Intersection acceleration

This chapter deals with the use of acceleration structure to reduce the computation time for a rendered image.

As shown on the chapter on ray-object intersections, ray tracing scenes with multiple complex objects can take a long time to render. There are multiple ways to accelerate these render times, but one has chosen two simple, but rather efficient ones: encapsulating the whole scene geometry in what is called *scene bounding box* and encapsulating each triangle mesh in a bounding box as well. For the bounding boxes one uses AABBs. Depending on the scene one can have different amount of decrease of render time and ray-object intersections.

To show the usefulness of this comparatively small improvement, one has the following simple scene consisting of three triangle meshes and a floor made of two triangles (see figure 33).

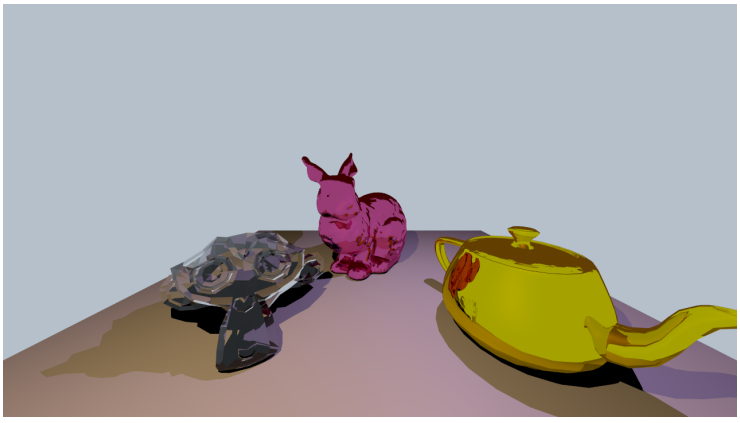


Figure 33: Intersection acceleration scene

The scene is rendered twice: once with the intersection acceleration structures mentioned above, and once without. Table 5 provides render statistics. The two noteworthy of them are the number of *ray-primitive intersection tests*. As one can see in the renderer where no intersection acceleration is used their amount is 7 larger. Also its render time is larger by roughly the same amount of time. This is due to the fact, that the triangle meshes used in the scene (and especially the so called *Stanford bunny*, with $69,630$ triangles) are quite expensive to compute. And having each individual ray to intersect $73,064$ triangles, made the scene take so long to compute, more than 20 hours in comparison to the a slightly more than 3 hours needed to compute the same scene using intersection acceleration.

Characteristic	with intersection acceleration	without intersection acceleration
ray-prim. intersection tests	270,039,112,893	2,059,088,146,519
ray-object intersections	12,934,567	12,935,679
render time	10,982 s	73,528 s

Table 5: Rendering information of the images at fig. 12

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.

[DG04] Bram De Greve. Reflections and refractions in ray tracing. 2004.

[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.

[Kem15] Mitchell Kember. How to write a ray tracer, 2015.

[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.

[Poy03] Charles A. Poynton. *Digital Video and HDTV: Algorithms and Interfaces*. Morgan Kaufmann, 2003.

[Sap17] Scract-a-pixel. Introduction to shading, 2017.

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

[Shi00] Peter Shirley. *Realistic ray tracing*. A K Peters, 2000.

[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.

[Wik17a] Wikipedia. 3d projection, 2017.

[Wik17b] Wikipedia. Inverse-square law, 2017.

[Wik17c] Wikipedia. Phong reflection model, 2017.

[Wik17d] Wikipedia. Wavefront .obj file, 2017.