Rendering

Sung-eui Yoon
KAIST
September 4, 2016

- This chapter was updated at Sep., 2016.
- Copyright, Sung-eui Yoon, 2016
- This is downloaded from `http://sglab.kaist.ac.kr/~sungeui/render`.

Part II

Physically-based Rendering

In Part I, we discussed rasterization techniques. While the raterization technique provides the efficient performance based on rendering pipeline utilizing modern GPUs, its fundamental approach is not based on the physical interaction between lights and materials. Another large stream of rendering methods are based on such physical interactions and thus are known as physically-based rendering.

In this part, we discuss two different approaches, ray tracing and radiosity, of physically based rendering methods. Ray tracing and radiosity are two main building blocks of many interactive or physically based rendering techniques. We first discuss ray tracing in this chapter, followed by radiosity (Ch. 6. We then study radiometric quantities (Ch. 7) to measure different energy terms to describe the physical interaction, known as the rendering equation (Ch. 8.1).

The rendering equation is a high dimensional integral problem, and thus its analytic solutions in many cases are not available. As an effective solution to solving the equation, we study the Monte Carlo technique, a numerical approach in Ch. 9, and its integration with ray tracing in Ch. 10. In many practical problems, such Monte Carlo approaches are slow to converge to noise-free images. We therefore study importance sampling techniques in Ch. 9.3.

4.1 Available Tools

Physically based rendering has been studied for many decades, and many useful resources are available. Some of them are listed here:

- Physically Based Rendering: From Theory to Implementation [PH10]. This book also known as pbrt comes with concepts with their actual implementations. As a result, readers can get understanding on those concepts and actual implementation that they can play with. Since this book discusses such implementation, we strongly recommend you to play with their source codes, which are available at github.

- Embree [WWB$^+$14] and Optix [PBD$^+$10]. Embree and Optix are interactive ray tracing kernels that run on CPUs and GPUs, respectively. While source codes of Optix are unavailable, Embree comes with their source codes.

- Instant Radiosity. Instant radiosiy is widely used in many games, thanks to its high quality rendering results with reasonably fast performance. Unfortunately due to its importance in recent game industry, mature library or open source projects are not available. One of useful open source projects are from my graphics lab. It is available at: `http://sglab.kaist.ac.kr/~sungeui/ICG/student_presentations.html`.

Chapter 5

Ray Tracing

Ray casting and tracing techniques have been introduced late 70's and early 80's to the computer graphics field as rendering techniques for achieving high-quality images.

Ray casting [App68] shoots a ray from the camera origin to a pixel and compute the first intersection point between the ray and objects in the scene. Ray casting then computes the color from the intersection point and use it as the color of the pixel. It computes a direct illumination that has one bounce from the light to the eye. Its result is same to those of the basic rasterization considering only the direct illumination.

Ray tracing [Whi80] is an recursive version of the ray casting. In other words, once we have the intersection between the initial ray and objects, ray tracing generates another ray or rays to simulate the interaction between and the light and objects. A ray can be considered as a photon traveling in a straight line, and by simulating many rays in a physically correct way, we can achieve physically correct images. While the algorithm is extremely simple, we can support various effects by generating different rays (Fig. 5.1).

5.1 Basic algorithm

Ray tracing simulates how a photon interacts with objects.

The basic ray tracing algorithm is very simple, as shown in Algorithm 1. We first generate a ray from the eye to the scene. While a photon travels from a light source, we typically perform ray tracing in backward from the eye (Fig. 5.2). We then identify the first intersection point between the ray and the scene. This has been studied well, especially around the early stage of developing this technique. At this point, we simply assume that we can compute such intersection points and this is discussed in Sec. 5.2.

Suppose that we identify such an intersection point between the ray and the scene. We can then perform various shading operations based on the Phong illumination (Sec. 4). To see whether the point is under the shadow or not, we simple generate another ray, called shadow ray, to the light source (the bottom image of Fig. 5.2).

Reflection and refractions are handled in a similar manner by generating another secondary rays (Fig. 5.3). The main question that we need to address here is how we can construct the secondary rays for supporting reflection and refraction. For the mirror-like objects, we can apply the perfect-specular reflection and compute the reflection direction for the reflection ray, where the incoming angle is same to the outgoing angle. In other words, the origin of the reflection ray,

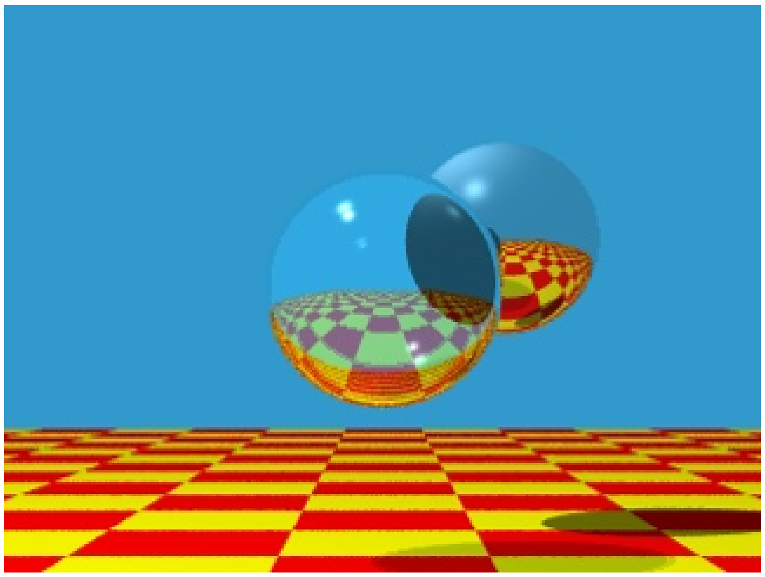

Figure 5.1: One of early images generated by ray tracing, i.e., Whitted style ray tracing [Whi80]. The image has reflection, refraction, and shadow effects. The image is excerpted from [Whi80].

Algorithm 1 Basic ray tracing

```
Trace rays from the eye into the scene (backward ray tracing).
Identify the first intersection point and shade with it.
Generate additional, secondary rays needed for shading.
        Generate ray for reflections.
        Generate ray for refraction and transparency.
        Generate ray for shadows.
```

R, is set to the hit point of the prior ray, and the direction of R is set as the reflection direction. Its exact equation is shown in Sec. 4.

Most objects in practice do not support such perfect reflection. For simple cases such as rays bending in glasses or water, we apply the Snell's law to compute the outgoing angle for refraction. The Snell's law is described as follows:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}, \tag{5.1}$$

where θ_1 and θ_2 are incoming and outgoing angles given rays at the interface between two different objects (Fig. 5.4). n_1 and n_2 are refractive indices of those two objects. The refractive index of a material (e.g., water) is defined as $\frac{c}{v}$, where c is the velocity of the light in vacuum, while v

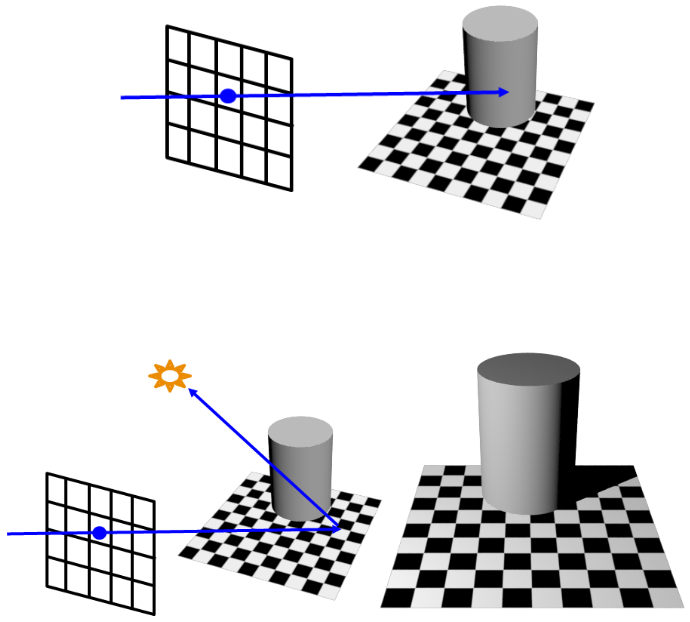

Figure 5.2: We generate a ray, primary ray, from the eye (top). To see whether the intersection point is in the shadow or not, we generate another ray, shadow ray, to the light source (bottom). These images are created by using 3ds Max.

is the speed of the light in that material. As a result, refractive indices of different materials are measured and can be used for simulating such materials within ray tracing.

Many objects used in practice consist of many different materials. As a result, the Snell's law designed for isotropic media may not be appropriate for such cases. For general cases, BRDF and BSSRDF have been proposed and are discussed in Ch. 7.

Physically based rendering techniques adopt many physical laws, as exemplified by adopting the Snell's law for computing refraction rays. This is one of main difference between rasterization and physically based rendering methods.

For various effects, ray tracing generate different types of rays, while rasterization adopts different types of texture maps.

Note that in rasterization techniques, to handle shadow, reflection, refraction, and many other rendering effects, we commonly generate some maps (e.g., shadow maps) accommodating such effects. As a result, handling texture mapping efficiently is one of key components for many rasterization techniques running on GPUs. On the other hand, ray tracing generates various rays

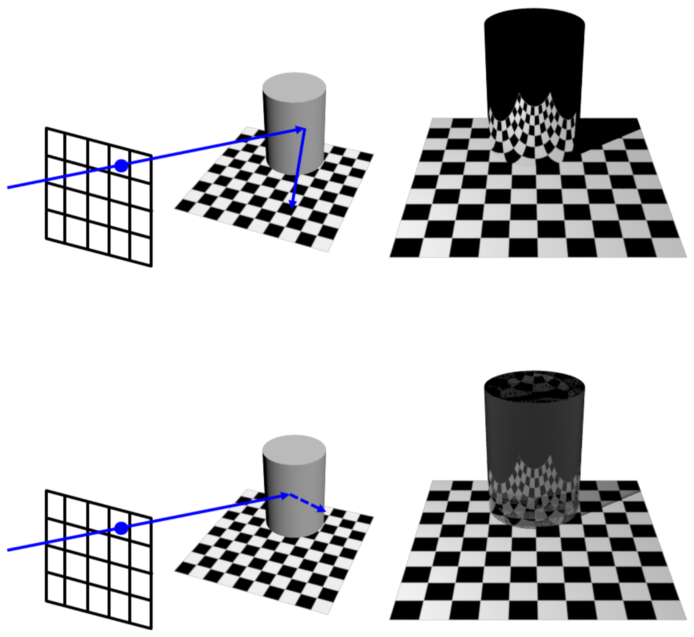

Figure 5.3: Handling reflection and refraction by generating secondary rays.

for such effects, and handling rays efficiently is one of key components of ray tracing.

5.2 Intersection Tests

Performing intersection tests is one of main operations of ray tracing. Furthermore, they tend to become the main bottleneck of ray tracing and thus have been optimized for a few decades. In this section, we discuss basic ways of computing intersection tests between a ray and a few simple representations of a model.

Implicit forms of objects are commonly used for intersection tests.

Any points, $p(t)$, in a ray parameterized by a parameter t can be represented as follows:

$$p(t) = o + t\vec{d}, \tag{5.2}$$

where o and $\vec{d}$ are the origin and direction of the ray, respectively. A common way of approaching this problem is to first define an object in an implicit mathematical form, $f(p) = 0$, where p is any point on the object. We then compute the intersection point, t_i, satisfying $f(p(t_i)) = 0$.

We now look at a specific case of computing an intersection point between a ray and a plane.

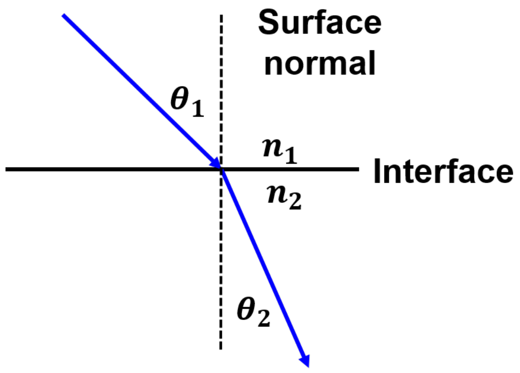

Figure 5.4: How a ray bends at an interface between simple objects, specifically, isotropic media such as water, air, and glass, is described by the Snell's law.

A well known implicit form of a plane is:

$$\vec{n}p - d = 0, \tag{5.3}$$

where $\vec{n}$ is a normalized normal vector of the plane and d is the distance from the origin to the plane. This implicit form of the plane equation is also known as the Hessian normal form [Wei].

By plugging the ray equation into the implicit of the plane equation, we get:

$$\begin{aligned} \vec{n}(o + t\vec{d}) - d &= 0 \\ t &= \frac{d - \vec{n}o}{\vec{n} \cdot \vec{d}}. \end{aligned} \tag{5.4}$$

We now discuss a ray intersection method against triangles, which are one of common representations of objects in computer graphics. There are many different ways of computing the intersection point with triangles. We approach the problem based on barycentric coordinates of points with a triangle.

Barycentric coordinates are computed based on non-orthogonal bases.

Barycentric coordinates are computed based on non-orthogonal bases unlike the Cartesian coordinate system, which uses orthogonal bases such as X, Y, and Z-axis. Suppose that p is an intersection point between a ray and a triangle consisting of three vertices, v_0, v_1, v_2 (Fig. 5.5).

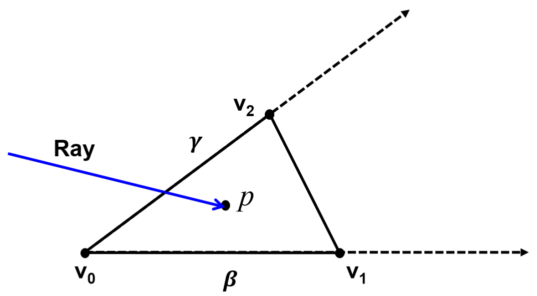

Figure 5.5: In the barycentric coordinate system, we represent the point p with β and γ coordinates with two non-orthogonal basis vectors, $v_1 - v_0$ and $v_2 - v_0$.

We can represent the point p as the following:

$$\begin{aligned} p =& v_0 + \beta(v_1 - v_0) + \gamma(v_2 - v_0) \\ =& (1 - \beta - \gamma)v_0 + \beta v_1 + \gamma v_2 \\ =& \alpha v_0 + \beta v_1 + \gamma v_2, \end{aligned} \tag{5.5}$$

where we use α to denote $1 - \beta - \gamma$. We can then see a constraint that $\alpha + \beta + \gamma = 1$, indicating that we have two degrees-of-freedom, while there are three parameters.

Let's see in what ranges of these parameters the point p is inside the triangle. Consider edges along two vectors $v_0 - v_1$ and $v_2 - v_0$ (Fig. 5.5). Along those edges, β and γ should be in $[0, 1]$, when the point is inside the triangle. Additionally, when we consider the other edge along the vector of $v_1 - v_2$, points on the edge satisfy $\gamma = 1 - \beta$[1]. When we plug the equation into the definition of α, we see α to be zero. On the other hand, on the point of v_0, β and γ should be zero, and thus α to be one. As a result, we have the following property:

$$0 \leq \alpha, \beta, \gamma \leq 1, \tag{5.6}$$

Barycentric coordinates are also known as area coordinates, since they map to areas of sub-triangles associated with vertices.

where these three coordinates are barycentric coordinates and $\alpha = 1 - \beta - \gamma$.

There are many different ways of computing barycentric coordinates given points defined in the Cartesian coordinate system. An intuitive way is to associate barycentric coordinates with areas of sub-triangles of the triangle; as a result, barycentric coordinates are also known as area coordinates. For example, β associated with v_1 is equal to the ratio of the area of $\triangle pv_0v_2$ to that of $\triangle v_0v_1v_2$.

Once we represent the intersection point p within the triangle with the barycentric coordi-

[1]When we consider a 2 D space whose basis vectors map to canonical vectors (e.g., X and Y axises) with β and γ coordinates, one can easily show that the relationship $\gamma = 1 - \beta$ is satisfied on the edge of $v_2 - v_1$.

nates, our goal is to find t of the ray that intersects with the triangle, denoted as the following:

$$o + t\vec{d} = (1 - \beta - \gamma)v_0 + \beta v_1 + \gamma v_2, \tag{5.7}$$

where unknown variables are t, β, γ. Since we have three different equations with X, Y, and Z coordinates of vertices and the ray, we can compute those three unknowns.

5.3 Bounding Volume Hierarchy

We have discussed how to perform intersection tests between a ray and implicit equations representing planes and triangles. Common models used in games and movies have thousands of or millions of triangles. A naive approach of computing the first intersection point between a ray and those triangles is to linearly scan those triangles and test the ray-triangle intersection tests. It, however, has a linear time complexity as a function of the number of triangles, and thus can take an excessive amount of computation time.

Many acceleration techniques have been proposed to reduce the time spent on ray intersection tests. Some of important techniques include optimized ray-triangle intersection tests using Barycentric coordinates [**?**]. In this section, we discuss an hierarchical acceleration technique that can improve the linear time complexity of the naive linear scan method.

Bounding volume hierarchies are simple to use and have been widly adopted in related applications including collision detection.

Two hierarchical techniques have been widely used for accelerating the performance of ray tracing. They are kd-trees and bounding volume hierarchies (BVHs). kd-trees are constructed by partitioning the space of a scene and thus are classified as spatial partitioning trees. On the other hand, BVHs are constructed by partitioning underlying primitives (e.g., triangles) and thus known as object partitioning trees. They have been demonstrated to work well in most cases [WWB+14]. We focus on explaining BVHs in this chapter thanks to its simplicity and wide acceptance in related fields such as collision detection.

5.3.1 Bounding Volumes

We first discuss bounding volumes (BVs). A BV is an object that encloses triangles. Also, the BV should be efficient for performing an intersection test between a ray and the BV. Given this constraint, simple geometric objects have been proposed. BVs commonly used in practice are sphere, Axis-Aligned Bounding Box (AABB), Oriented Bounding Box (OBB), k-DOPs (Discrete Oriented Polytopes), etc. (Fig. 5.6).

Spheres and AABBs are fast for checking intersection tests against a ray. Furthermore, constructing these BVs can be done quite quickly. For example, to compute a AABB from a soup of triangles, we just need to traverse those triangles and compute mim and max values of x, y, and z coordinates of triangles. We then compute the AABB out of those computed min and max values. Since many man made artifacts have box-like shapes, AABB works well for those types.

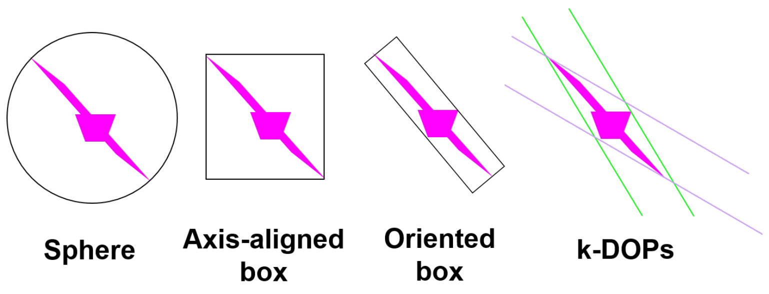

Figure 5.6: This figure shows different types of Bounding Volumes (BVs).

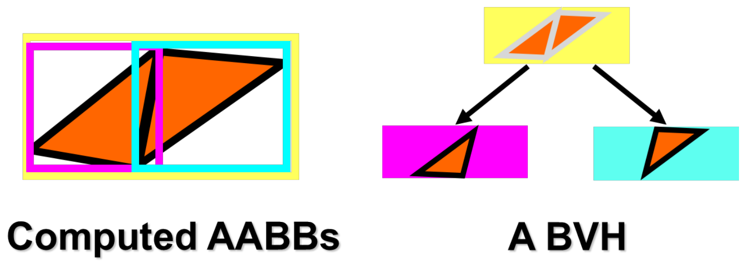

Figure 5.7: This figure shows a BVH with its nodes and AABBs given a model consisting of three triangles. Note that two child AABBs have a spatial overlap, while their nodes have different triangles. As a result, BVHs are classified into an object partitioning tree.

A single BV type is not always better than others, but AABBs work reasonably well and are easy to use.

Nonetheless, spheres and AABBs may be too lose BVs, especially when the underlying object is not aligned into such canonical directions or is elongated along a non-canonical direction (Fig. 5.6).

On the other hand, OBBs and k-DOPs tend to provide tighter bounding, but to require more complex and thus slow intersection tests. Given these trade-offs, an overhead of computing a BV, tightness of bounding, and time spent on intersection tests between a ray and a BV, it is hard to say which BV shows the best performance among all those BVs. Nonetheless, AABBs work reasonably well in models used for games and CAD industry, thanks to its simplicity and reasonable bounding power on those models.

5.3.2 Construction

Let's think about how we can construct a bounding volume hierarchy out of triangles. A simple approach is a top-down construction method, where we partition the input triangles into two sets in a recursive way, resulting in a binary tree. For simplicity, we use AABBs as BVs.

We first construct a root node with its AABB containing all the input triangles. We then partition those triangles into left and right child nodes. To partition those triangles associated with a current node, a simple method is to use a 2 D plane that partitions the longest edge of the current AABB of the node. Once we compute triangle sets for two child nodes, we recursively perform the process until each node has a fixed number of triangles (e.g., 1 or 2).

In the aforementioned method, we explained a simple partitioning method. More advanced techniques have been proposed including optimization techniques with Surface Area Heuristic (SAH) [LYTM06, WBS07]. The SAH method estimates the probability that a BV intersects with random rays, and we can estimate the quality of a computed BVH. It has been demonstrated that this kind of optimizations can be slower than the simple method, but can show shorter traversal time spent on performing ray-BVH intersection tests.

BVHs suits well for dynamic models, since it can be refitted or re-computed from scratch efficiently.

Dynamic models. Many applications (e.g., games) use dynamic or animated models. As a result, it is important to build or update BVHs of models as they are changing. This is one of main benefits of using BVHs for ray tracing, since it is easy to update the BVH of a model, as the model changes its positions or is animated.

One of the most simple methods is to refit the existing BVH in a bottom-up manner, as the model is changing. Each leaf node is associated with a few triangles. As they change their positions, we re-compute the min and max values of the node and update the AABB of the node. We then merge those re-computed AABBs of two child nodes for their parent node by traversing the BVH in a bottom-up manner. This process has the linear time complexity in terms of the number of triangle. Nonetheless, this refitting approach can result in a poor quality, when the underlying objects deform significantly.

To address those problems, many techniques have been proposed. Some of them is to build BVHs from scratch every frame by using many cores [LGS$^+$09] and to selectively identify a sub-BVH with poor quality and rebuild only those regions, known as selective restructuring [YCM07]. At an extreme case, the topology of models can change due to fracturing of models. BVH construction methods even for fracturing cases have been proposed [HSK$^+$10].

5.3.3 Traversing a BVH

Once we build a BVH, we now traverse the BVH for ray-triangle intersection tests. Since an AABB BVH provides AABBs, bounding boxes, on the scene in a hierarchical manner, we traverse the BVH in the hierarchical manner.

Given a ray, we first perform an intersection test between the ray and the AABB of the root node. If there is no intersection, it guarantees that there are no intersections between the ray and triangles contained in the AABB. As a result, we skip traversing its sub-tree. If there is an intersection, we traverse its sub-trees by accessing its two child nodes. Among two nodes, it is more desirable to access a node which is located closer to the ray origin, since we aim to identify the first intersection point along the ray starting from the ray origin.

Suppose that we decide to access the left node first. We then store the right node in a stack to process it later. We continue this process until we reach a leaf node containing primitives (e.g., triangles). Once we reach a leaf node, we perform ray-triangle intersection tests for identifying an intersection point. If it is guaranteed that the intersection point is the closest to the ray origin, we terminate the process. Otherwise, we contribute to traverse the tree, by fetching and accessing nodes in the stack.

Many types of BVHs do not provide a strict ordering between two child nodes given a ray. This characteristic can result in traversing many parts of BVHs, leading to lower performance. Fortunately, this issue has been studied, and improvements such as identifying near and far child nodes have been proposed [LYTM06, WBS07].

5.4 Visibility Algorithms

In this chapter, we discussed different aspects of ray tracing. At a higher level, ray casting, a module of ray tracing, is one type of visibility algorithms, since it essentially tells us whether we can see a triangle or not given a ray. In this section, we would like to briefly discuss other visibility algorithms.

While the Z-buffer method was considered as a brute-force method, it is the de-factor standard in the rasterization method thanks to its adoption in modern GPU architectures.

The Z-buffer method, an fundamental technique for rasterization (Part I), is another visibility algorithm. The Z-buffer method is an image-space method, which identifies a visible triangle at each pixel of an image buffer by considering the depth value, i.e., Z values of fragments of triangles. Many different visibility or hidden-surface removal techniques have been proposed. Old, but well-known techniques have been discussed in a famous survey [SSS74]. Interestingly, the Z-buffer method was mentioned as a brute-force method in the survey, because of its high memory requirement. Nonetheless, it has been widely adopted and used for many graphics applications, thanks to its simple method, resulting in an easy adoption in GPUs.

Compared with the Z-buffer, ray casting and ray tracing is much slower, since it uses a hierarchical data structure, and has many incoherent memory access. Ray casting based approaches, however, become more widely accepted in movies and games, because modern GPUs allow to support such complicated operations, and many algorithmic advances such as ray beams utilizing coherence have been prosed. It is hard to predict future exactly, but ray casting based approaches will be supported more and can be adopted as an interactive solution at some point in future.