FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Sebastian Schimper

Integration of the Embree Raycasting Library into a CSG Renderer

Department of Software and Computer Science Education

Supervisor of the master thesis:	doc. Alexander Wilkie, Dr.
Study programme:	Computer Science
Study branch:	Computer Graphics and Game Development

Prague 2021

I owe a significant depth of gratitude to my supervisor, Prof. Alexander Wilkie, for the possibility to contribute to the development of a groundbreaking rendering system and for his help, support, and trust in me. I would also like to express sincere thankfulness towards Alban Fichet. His help and feedback were of crucial importance to my work. Additionally, I would like to thank various members of the Computer Graphics Group in Prague for numerous reviews and discussions, and the "Computer Graphics Stack Exchange" users Peter and Simon F, two strangers on the internet, for offering help when no help was available, thus preventing me from going completely nuts. I am grateful to my brother for helping me resolving various issues concerning Latex. Another person I wish to thank is Solange Petracchi, my contact person at Charles University, for getting me out of trouble multiple times during my years of study and for her moral support. Finally, I would like to thank my family, friends, and Monika for their love and support during difficult times.

Title: Integration of the Embree Raycasting Library into a CSG Renderer

Author: Bc. Sebastian Schimper

Department: Department of Software and Computer Science Education

Supervisor: doc. Alexander Wilkie, Dr., Department of Software and Computer Science Education

Abstract: Modern High-Performance Ray Casting toolkits, such as the Intel Embree library, which is a de facto industry standard, are a cornerstone of the high-performance levels seen in current CPU rendering. The purpose of Embree is an easy integration into professional image synthesis environments to accelerate rendering scenes with complex geometry, usually composed of many primitives. Unfortunately, Embree does not offer support for rendering constructive solid geometry (CSG), solids composed of a manageable amount of primitive solids by using set operations. This is a significant drawback since CSG modeling is an intuitive and powerful option for describing complex geometry. In this thesis, we describe the integration of Embree into the predictive rendering system ART and propose a method for rendering CSG by combining the traversal of Embree's and ART's internal ray acceleration data structures. The tests we conducted with virtual scenes containing CSG not being constructed from triangle meshes showed that our method is competitive with the original ART renderer and often even faster.

Keywords: Raycasting, CSG, Embree

Contents

Introduction

Ray tracing is a powerful image synthesis technique used in 3D computer graphics, capable of simulating optical effects, such as reflection, refraction, and scattering, with a high degree of visual realism. For this reason, it offers a variety of applications. It is used in the entertainment industry, for product design, visual prototyping, and for creating visualizations in scientific research [Ped19, pp. 91–128]. Reasons for the popularity of ray tracing algorithms are the physical plausibility of its generated images as well as their simplicity and "elegance".

The origins of ray tracing can be traced back as far as 1968. At that time, a method was invented to shade wire-framed solids by shooting random light rays from virtual light sources to the scene geometry and, if an intersection would be found, placing a symbol at the intersection point [App68]. If enough rays were generated, there would be a high concentration of these symbols at regions with high light intensity, which would approximate physical realism. A decade later, an elaboration of this idea to a shading model that takes global information into account for calculating light intensities [Whi79], would revolutionize the computer graphics field.

The calculation of intersection points between cast rays and scene geometry is a crucial part of ray tracing. How these points are calculated depends on the representation of geometric primitives in the scene. Primitives can be represented analytically or approximated by polygon meshes. Another possibility is the representation of a shape as a *Constructive Solid Geometry (CSG)*. CSG is a hierarchical ordering of a set of primitive shapes to which Boolean set operations have been applied to. This form of representation enjoys popularity in e.g. Computer-aided geometric design.

Over the years, novel ray tracing algorithms were developed to increase the physical correctness of rendered images. However, ray tracing remains to this day a computationally demanding task. Much research was (and still is) devoted to accelerating the ray tracing procedure to compensate for its computational expense. Therefore, a wide range of researches focus on accelerating ray tracing algorithms to get better performances. Additionally, ray tracing is by nature is "embarrassingly parallel", meaning it is well suited for parallel processing by

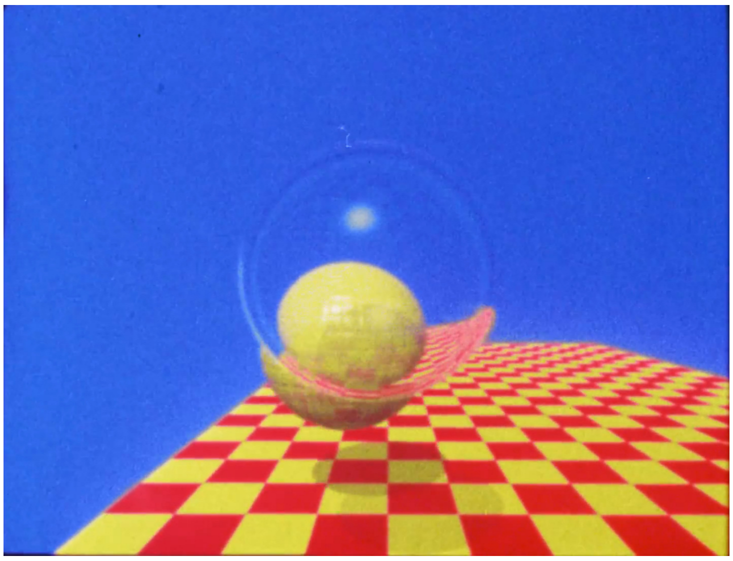


Figure 1 A frame of an animation [NVI18] form [Whi79], demonstrating recursive ray tracing, featuring a glass sphere circulating a second defuse sphere. Both are casting shadows on a diffuse checkerboard-like ground.

automatic vectorization. As Moore's law predicted, the computational power in CPUs gradually increased. Nowadays, modern computers admit an integrated circuit with multiple cores on which the workload of ray tracing algorithms can be distributed. While ray tracing was considered impractical when it was pioneered, it has now become more accessible thanks to the increase of CPU power and specialized hardware.

Despite this, exploiting the computational power of modern processors to their full potential for ray tracing remains challenging. This particular reason served as the motivation for developing the award-winning ([Int21]), open-source framework *Embree* [Wal+14]. Embree offers a set of ray tracing kernels that maximize the compute capabilities of modern x86 CPU architectures.

One of the design goals behind Embree is to provide an easy integration of the framework into existing professional ray tracing environments to achieve high performance when ray tracing virtual scenes with high geometrical complexity.

Thesis subject and motivation

The goal of this thesis is a successful integration of the Embree framework into the predictive rendering framework *The Advanced Rendering Toolkit*, to facilitate the acceleration of ray tracing for constructive solid geometry. The Advanced Rendering Toolkit will be referred to with its abbreviation *ART* throughout this thesis.

ART offers innovative features, such as efficient spectral rendering (by implementing the "Hero Wavelength Spectral Sampling" technique [Wil+14]), proper handling of bi-spectral materials (e.g., fluorescent surfaces) [MFW18] and a physically plausible sky dome lighting model [WH13]. Until the release of Mitsuba 2 [ND+19], ART was, to our best knowledge, the only rendering system that would support rendering polarization effects, and remains the sole open source rendering system to support bi-spectral reflectances. These features make ART an interesting environment for computer graphics researchers interested in the field of Predictive Rendering.

To ensure this features, ART relies on its proprietary internal data structures that diverge to a significant degree from those present in other popular rendering systems (e.g., PBRT [PJH16] or Mitsuba 2). Therefore, Embree's integration into ART is a non-trivial task, although Embree was developed with the intention of it being "used in existing renderers with minimal programmer effort" [Wal+14, p. 1]. Furthermore, Embree unfortunately does not directly support rendering CSG.

If a successful integration of Embree into ART could be achieved, a complex image synthesis system with unique predictive rendering features would be adapted to the industry standard of ray tracing.

Thesis outline

This thesis is structured as follows:

- **Chapter 1** provides fundamental background information, including a brief introduction to the ray tracing technique, explanations of the most common ray acceleration structures, and a brief overview of the functionality of the Embree framework.
- **Chapter 2** provides a description of Embree can be used for ray tracing and a brief introduction to ART, in which Embree will be integrated into.
- **Chapter 3** is dedicated to the description of our approach on the integration of Embree into ART, as well as the implementation of the CSG operations with Embree.
- **Chapter 4** shows the results obtained by testing our implementation on various virtual scenes.
- Lastly, **Appendix B.1** provides a user guide for compiling ART with Embree support.

Chapter 1

Fundamentals

Under the term *ray tracing* we understand the utilization of algorithms that involve the casting of virtual light rays for generating images. The purpose of these light rays is to archive high visual realism by the simulation of real-life behavior.

We as persons can perceive nearby objects or persons due to the following: Light sources, either natural or artificial, emit electromagnetic (EM) waves. This radiation is reflected off objects in the scene (informally, it "bounces" on objects). The human eye or a camera sensor interprets EM waves of different frequencies as colors. The resulting transport of these EM waves in the scene to the human eye or a camera sensor forms an image.

The purpose of ray tracing algorithms is to imitate this behavior, usually by tracing these light rays in reverse order from a point in the scene, a virtual camera or an "eye point", back to the emitting light source.

An advantage of these ray tracing algorithms is its core procedure being straightforward when viewed from a theoretical point of view.

Figure 1.1 shows an example of a virtual scene. It is composed of an orange sphere, a white ground plane, and a light source. Furthermore, an image plane exists in the scene, on which the 2D image of the 3D scene will be projected. A ray is consisting of two components, an origin point and a direction vector. The $EyePoint$ in 1.1 will serve as the origin of the cast rays. In the figure, the image plane is composed of multiple quadratic "cells" that represent the actual pixels of the resulting image. Through each of these cells, a ray is cast into the scene from the eye point.

The subsequent step is to determine, whether that ray intersected a particular geometry by performing intersection tests on all geometries in the scene [1].

In case a geometry is intersected, a secondary ray is generated with its origin

[1]Testing all geometries present in a complex scene for intersection is of course a naive approach and not practical at all. This procedure is solely mentioned here for the sake of illustration. Section 1.3 introduces some common ray acceleration data structures to compensate for this.

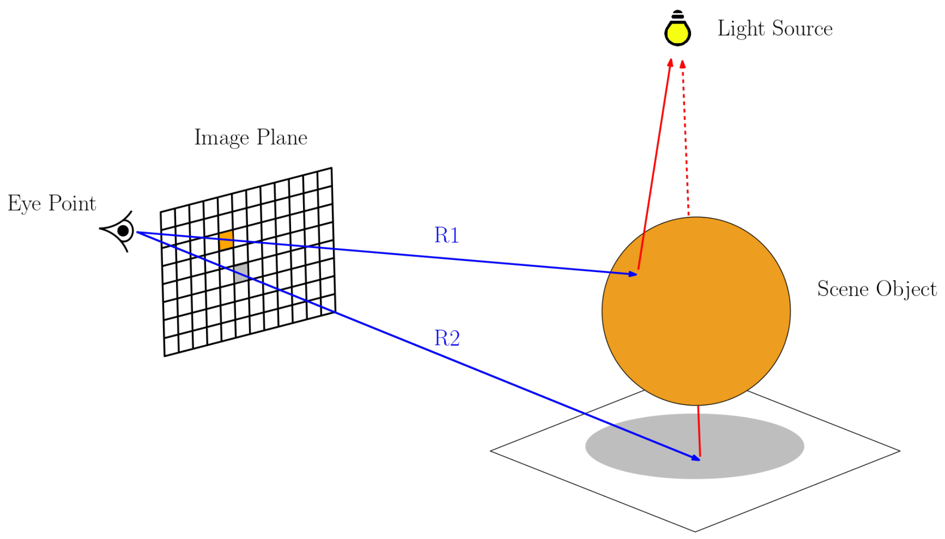


Figure 1.1 Ray tracing procedure for calculating global illumination.

at the intersection point with the closest distance to the $EyePoint$ and its direction toward the light source. In case this secondary light ray does not intersect any other geometry between its origin and the light source, this means that the first intersection point is exposed to light and the material color at that point is used for the corresponding pixel (see R1 in figure 1.1). Otherwise, the intersection point must be in shadow (see R2). This procedure generates an image with local illumination.

The following chapter is dedicated to providing background information on ray tracing, shape representation in rendering systems, and ray acceleration data structures. Furthermore, the Embree framework is introduced.

1.1 Ray tracing algorithms

The following section outlines the development of various ray tracing techniques. The pioneering work of [App68] and [Whi79] will be briefly discussed. Furthermore, the derivation from the definition of radiance to the rendering equation [Kaj86], whose numerical solving via Monte Carlo integration is the purpose of modern rendering environments, will be presented.

1.1.1 Origins of ray tracing

As briefly mentioned in the Introduction, ray tracing was pioneered in 1968 by [App68]. His work aimed to provide basic shading for wire-framed solids to

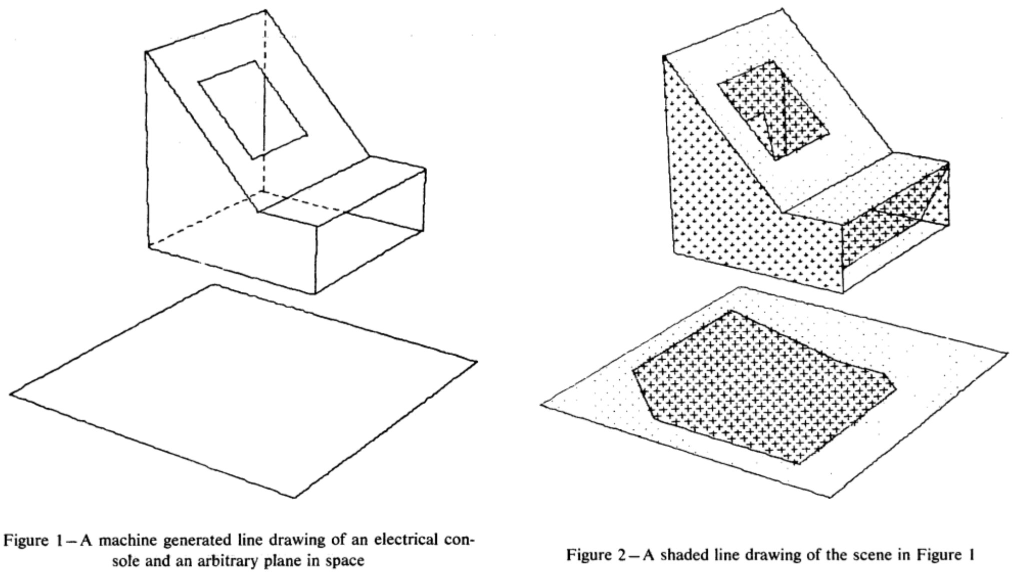


Figure 1 – A machine generated line drawing of an electrical console and an arbitrary plane in space
Figure 2 – A shaded line drawing of the scene in Figure 1

Figure 1.2 Original figures from [App68]. The left figure shows plain solid geometry and the right figure shows the same solid with shading applied to it.

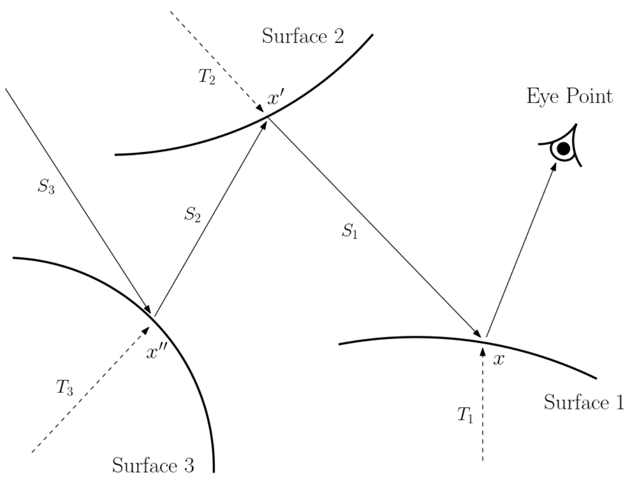
communicate better spatial relations and depth of objects in the rendered image.

To achieve this shading, virtual light rays were cast from a scene light source in random directions. Whenever one such ray intersects a geometry, a character or symbol (e.g., a small "plus"-symbol or square) was placed at that intersection point. If enough rays were cast, areas on the solid exposed to light would be shaded by these symbols. The result would then come to be by inverting the shaded and non-shaded areas of the geometry.

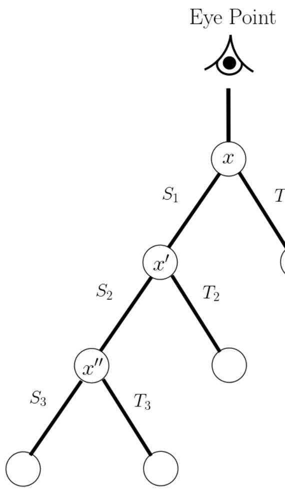
Figure 1.2 shows a result of this approach. Without the additional shading information, it would be difficult for the observer to perceive the position of the upper geometry relative to the plane. To achieve convincing results, a high number of rays had to be generated ("Even for about 1000 light rays, results were splotchy." [App68, p 3]). At the time of publication, the available hardware was hardly powerful enough for this shading method.

The idea of casting rays later became a key utilization for a shading model that aimed for higher realism by taking the "global setting" of geometries into account [Whi79]. At that time, a variety of shading models existed which were able to display optical effects convincingly. However, these models usually worked only in special cases and not well with each other, as noted by Andrew Glassner in the preface of his book *An introduction to ray tracing* [Gla89]. For example, some models existed that were good at calculating reflection effects but could not handle refraction effects well. And vice versa.

[Whi79] introduced a shading model that would truthfully simulate reflec-



(a) Light is being reflected from other surfaces before reaching the Eye Point.



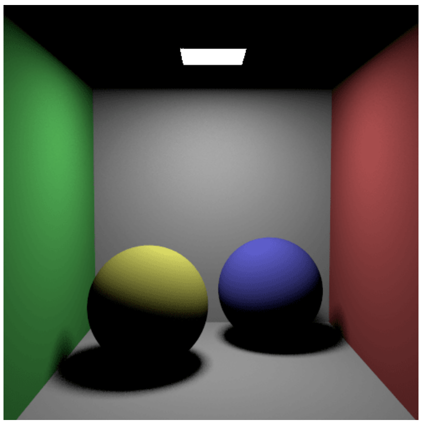
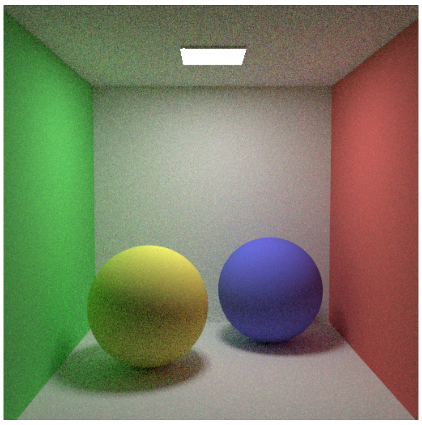
(b) Tree structure storing the individual reflection and transmission components.

Figure 1.3 Figures based on the original figures from the paper "An improved illumination model for shaded display" [Whi79].

tion, shadows and refraction as well as the effects of other conventional shading models at that time. The model is partially derived from an empirical reflection model developed by [Pho75]. This so-called "Phong reflection model" assumes that light, which is reflected from a surface, is composed of ambient reflection, diffuse reflection and specular reflection. In contrast, Whitted's model assumes that the light intensity arriving at the $EyePoint$ from an intersection point is conglomerated by a specular reflection component and a transmission component.

In real-life, light that is propagated towards an observer from a surface most certainly has interacted with other surfaces before. If true realism of computer generated images is desired, these previous interactions have to be taken into consideration, even for virtual scenes with moderately complex geometry. An example of such an event can be seen in Figure 1.3a.

This natural behavior is implemented in the following way. From the $EyePoint$, a ray is cast towards the virtual scene and a possible intersection point x with the scene geometry is calculated. The transmission and specular component rays at that intersection point are then recursively calculated and stored in a tree data structure which is shown in Figure 1.3b. To prevent a branch of the tree from growing infinitely large, it is truncated as soon as an attempt is made to access more storage than was previously made available for it. After such a tree is created, it is traversed recursively in order to calculate the light

(a) **(b)**

Figure 1.4 The Cornell Box scene with direct illumination (Figure 1.4a) and global illumination (Figure 1.4b). Both scenes were rendered by a simple ray tracing program that was implemented by students attending the course *Computer Graphics III - Physically based rendering (NPGR010)*, held at Charles University in Prague [Kř19].

intensity at each node, finally resulting in the calculation of the total light intensity that is reflected towards the $EyePoint$. Between two nodes, the intensity is attenuated according to a distance function between the intersection points, associated with the node and the node's parent node. Such trees are created and traversed for every pixel of the image plane. This procedure allows for the convincing display of a variety of optical effects with the help of a single model.

1.1.2 The rendering equation

Figure 1.4a shows an image of the Cornell Box. The scene's appearance results from casting a ray into the scene, calculating a possible intersection point, and then generating a second ray in the direction of the light source. However, the appearance of the Cornell Box is not physically realistic because the ceiling is not illuminated. This is since, with this approach, no light rays that interact with surface points on the ceiling can directly reach the emissive area of the light source. The degree of realism of the scene shown in Figure 1.4a can be significantly improved by considering global illumination effects.

In 1986, an integral equation was developed by Kajiya [Kaj86], that would describe the total reflected radiance towards an observer as the "sum" (or integral) of all light contributions over a hemisphere at a point on a surface. This mathematical model takes direct illumination (light from light sources) and indirect illumination (light being reflected off other surfaces in the scene) into account.

The so-called *rendering equation* is given by

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_{H(x)} L(x, \omega_i) f_r(\omega_i \to \omega_o) \cos\theta \partial\omega_i \tag{1.1}$$

where

- $L(x, \omega_o)$ is the total reflected light intensity from a surface point x towards the observer along the outgoing direction ω_o,
- $L_e(x, \omega_o)$ is the radiance emitted at the surface point x and propagated along ω_o,
- $H(x)$ is a hemisphere over surface point x,
- $L(x, \omega_i)$ is the light intensity incident to x along direction ω_i,
- $f_r(\omega_i \to \omega_o)$ is the Bidirectional Reflectance Distribution Function (BRDF), and
- $\cos\theta$ is a term, compensating for Lambert's cosine law.

For a better understanding of the individual terms of the rendering equation, a definition of radiance and a brief explanation of the bidirectional reflectance distribution function and the local reflection equation is provided in the following subsections. The information provided by these subsections are collated from the lecture notes of the course *Computer Graphics III - Physically based rendering (NPGR010)* held at Charles University in Prague, Czech Republic [Kř19], from the book *Physically based rendering: From theory to implementation* [PJH16] and from the book *Computer Graphics: Principles and Practice* [Hug+13].

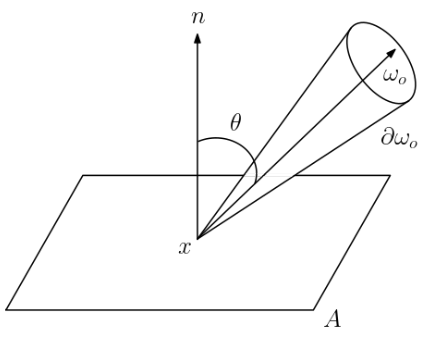
Definition of radiance

We define the *radiance* of a source, sometimes informally referred to as "brightness", as the power per unit area ∂A perpendicular to the ray in the direction ω_o and per unit solid angle that is propagated along with it (see Figure 1.5a). The following equation describes this:

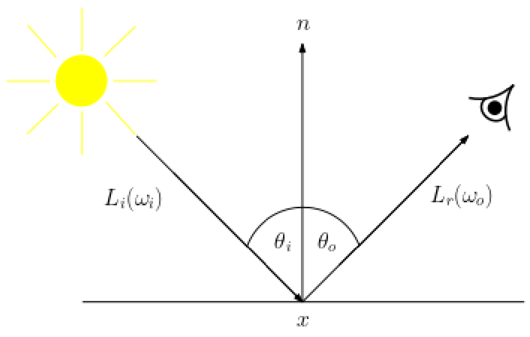
$$L(\omega_o) = \frac{\partial^2 \phi}{\partial\omega_o \partial A \cos\theta} [\mathrm{W} \cdot \mathrm{sr}^{-1} \cdot \mathrm{m}^{-2}] \tag{1.2}$$

where

- ω_o is the outgoing direction
- ϕ is flux or radiance per unit time
- A is the surface area, and



(a) Radiance is defined as the radiant flux emitted, received or reflected per solid angle $\partial\omega_o$ per unit projected area ∂A.



(b) The BRDF is a function defining how much light from the incoming directionω_i is reflected towards the viewer along the outgoing direction ω_o.

Figure 1.5 Diagrams visualizing radiance (Figure 1.5a) and the BRDF (Figure 1.5b).

- $\cos\theta$ a term for compensating Lambert's cosine law

From this definition we can derive the bidirectional reflectance distribution function.

Bidirectional Reflectance Distrubution Function (BRDF)

The *Bidirectional Reflectance Distribution Function (BRDF)* is a mathematical model describing the reflection properties of a given surface. To be precise, it describes the probability of light energy, arriving at a point x on a surface from direction ω_i, being reflected along the reflection ω_o. The BRDF model is defined by:

$$f_r(\omega_i \to \omega_o) = \frac{\partial L_r(\omega_o)}{L_i(\omega_i)\cos\theta\partial\omega_i}[\mathrm{sr}^{-1}] \tag{1.3}$$

where

- $L_r(\omega_o)$ is the reflected light energy from a surface point along the direction ω_o, and
- $L_i(\omega_i)$ is the incident light energy arriving at the surface point from direction ω_i

Properties of BRDFs are the conservation of energy and the Helmholtz reciprocity, which considers the incident and reflected light intensities in Equation 1.3 as interchangeable without affecting the result. There exist different types of BRDFs: empirical BRDFs, physically based BRDFs, and BRDFs being an approximation of measured data. BRDFs are a crucial component when calculating direct illumination with the *local reflection equation.*

Reflection equation

The *Reflection Equation* describes how much total light is reflected from a surface point x towards an observer along a given direction ω_o, taking the light intensities arriving at x from all incident directions into account. It is given by:

$$L_r(x, \omega_o) = \int_{H(x)} L_i(x, \omega_i) f_r(\omega_i \rightarrow \omega_o) \cos\theta \partial\omega_i \tag{1.4}$$

where

- $H(x)$ is a hemisphere over the surface point x.

The total amount of reflected energy is calculated by integrating all contributions of incident radiance over the hemisphere $H(x)$. The BRDF serves as a weight in this equation because only the energy reflected along ω_o is considered.

The rendering equation revisited

The rendering equation, which for reasons of convenience is shown again in Equation 1.5 can be arguably regarded as an extension of the local reflection equation.

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_{H(x)} L(x, \omega_i) f_r(\omega_i \rightarrow \omega_o) \cos\theta \partial\omega_i \tag{1.5}$$

Essentially, it describes the total reflection of energy towards direction ω_o from a surface point x as the "sum" or integral of all the light intensity, incident to all directions over a hemisphere over the point x, together with the intensity emitted from point x, if x is located on an emissive material. The unknown variable L is present on both sides of this equation.

This function is a higher-order integral which is difficult to solve analytically. The most common approach to solving this integral equation is the utilization of Monte Carlo methods. Monte Carlo methods numerically approximate a given integral by drawing random samples. The convergence speed of this procedure is independent of the dimension of the integral. Most of today's image synthesis algorithms implement Monte Carlo methods to approximate the rendering equation's solution.

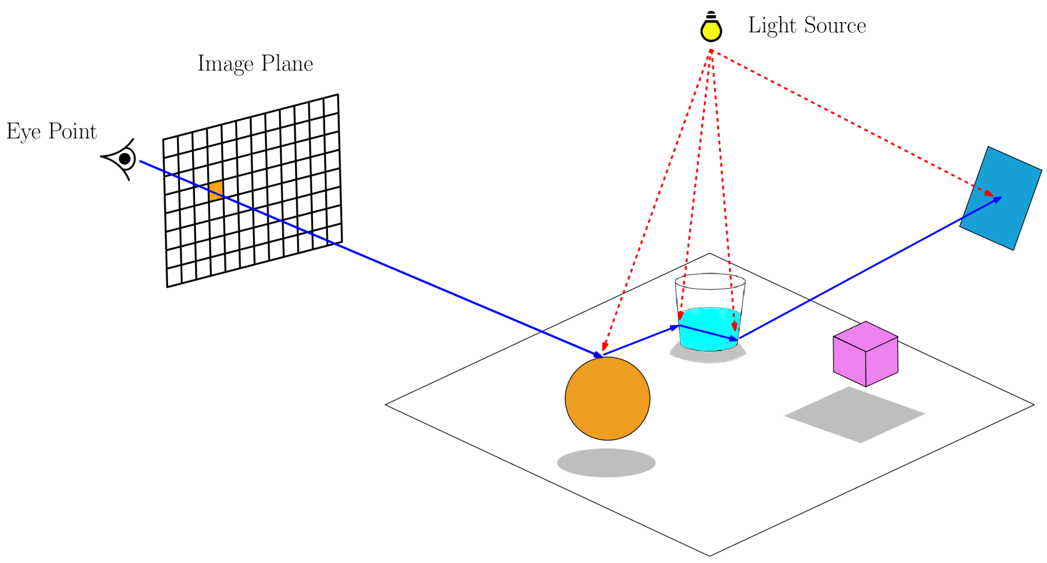


Figure 1.6 Tracing a path for calculating global illumination.

Path tracing

Multiple approaches for solving the rendering equation exist, for example, the *radiosity method* [Gor+84], which aims at solving it via the application of the finite element method. However, the most commonly used techniques for solving the rendering equation are Monte Carlo methods. The Monte Carlo method *path tracing* This method is based on the property that in a vacuum, radiance is constant along straight lines.

When a given ray intersects scene geometry, a secondary ray can be spawned at the intersection point, going along a direction that conforms to the BRDF associated with the intersected geometry. By repetition of this procedure, a "path" of rays is generated.

To calculate light intensity at each pixel of an image, path tracing generates such paths of rays, starting at the $EyePoint$ and ending at a light source. An example of such a path is visualized in Figure 1.6. At each intersection point along the path, the intersected geometries are tested for occlusion concerning the light sources available in the scene. If another geometry does not occlude the shape, the direct contribution of the light sources is accumulated. To prevent the calculations of unprofitable paths, a path is continued according to a so-called "survival probability." This probability can, for example, be formulated as the reflectivity of the surface, which is intersected by a ray. If this surface, for example, reflects only ten percent of light energy, the path is continued with a probability of then percent. To approximate an integral part of the rendering equation, numerous paths are generated at the intersection point x. To generate the final

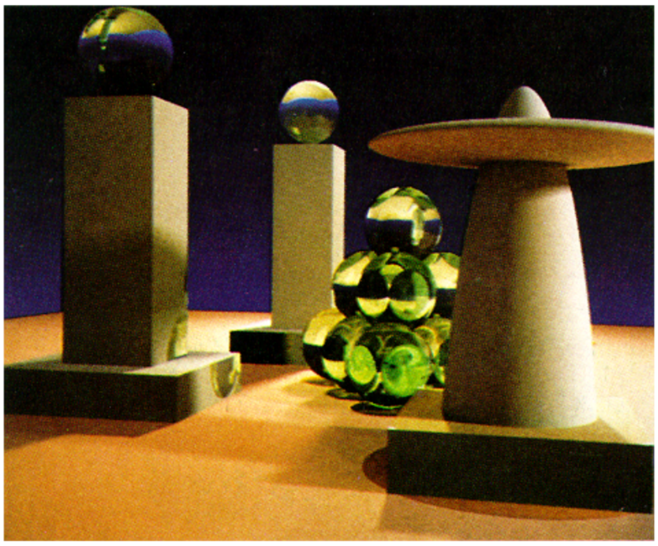


Figure 6. A sample image. All objects are neutral grey. Color on the objects is due to caustics from the green glass balls and color bleeding from the base polygon.

Figure 1.7 Original figure from [Kaj86].

color of the image pixels, the results calculated by the paths are averaged.

Concerning path tracing, we can re-write the rendering equation shown in 1.1 in the following way:

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_{H(x)} L(ray(x, \omega_i), -\omega_i) f_r(\omega_i \to \omega_o) \cos\theta \partial\omega_i \tag{1.6}$$

The term $ray(x, \omega_i)$ is a function for recursively calculating all incoming radiance arriving at the point x.

In comparison to the ray tracing method of [Whi79], path tracing is capable of simulating advanced optical effects such as soft shadows and diffuse interreflection. An example of this can be seen in Figure 1.4b.

1.2 Solid representations in 3D space

A crucial part of ray tracing algorithms is calculating intersection points between rays and the scene geometry. Solid objects can be represented in the Euclidean space in various ways. The intersection testing for these solids will depend on their representation.

The following section illustrates three types of solid representations in image synthesis environments, such as ART: Analytic surfaces, polygon meshes, and

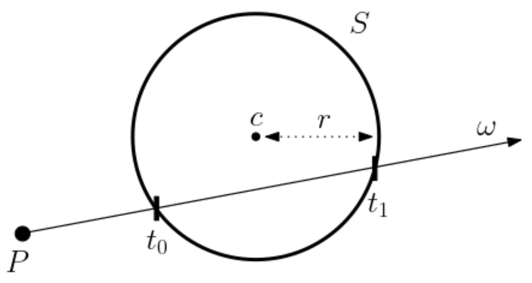


Figure 1.8 Intersection between a sphere S with its center point c and radius r, and a ray with origin P and direction ω. We attempt to solve a quadradic equation in oder to obtain points t_0 and t_1.

constructive solid geometry.

1.2.1 Analytical surfaces

Analytical surfaces are surfaces in three-dimensional Euclidean space, defined by analytic functions. Common analytical surfaces are quadric shapes (or sometimes called quadrics). Examples of this type of surface representation are spheres, cones, cylinders, and paraboloids. The equations describing these shapes can be in implicit form, which has the pleasant property, that testing whether a point p is located at the boundary of such a surface is easy.

Generally speaking, implicit equations are relations of the form $f(x_0, x_1, ..., x_{n-1}) = 0$ where f is a function of multiple variables. These variables can be considered coordinates of a point p in n-dimensional space and evaluated by function f. An evaluation of f will result in two possible outcomes: either $f(p) = 0$, which means p is located on the surface of the shape, or $f(p) \neq 0$, which means the opposite.

Due to this straightforward evaluation, analytical shapes are well suited for intersection testing during ray tracing.

In the following, we will provide an example of an intersection calculation between a given unit sphere S with radius 1 and a given ray R.

The sphere S is implicitly defined by:

$$S(x, y, z) = x^2 + y^2 + z^2 - 1 = 0 \tag{1.7}$$

The ray R in its parametric form is defined by:

$$R(t) = P + t\omega \tag{1.8}$$

where

- P is the origin of ray R, a point in Euclidean space,

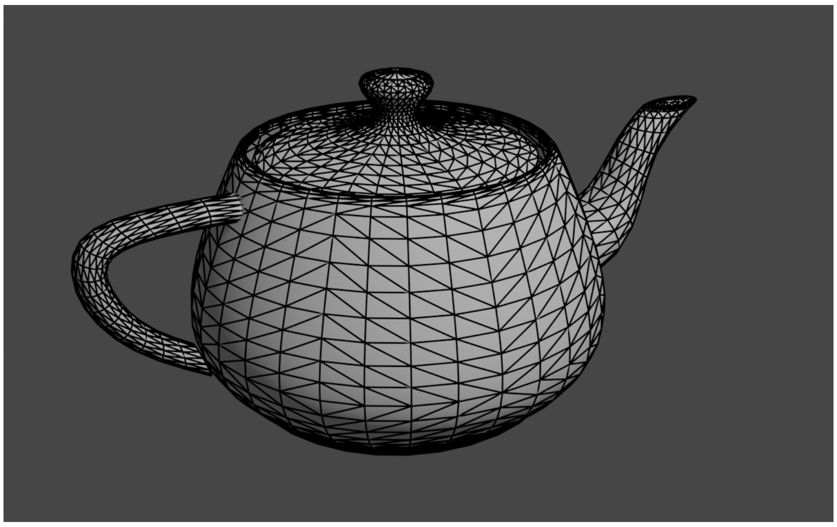


Figure 1.9 Polygon mesh of the famous Utah Teapot, rendered with Blender [Ble18]

- t a scalar, and
- ω is a normalized vector.

To obtain the two intersection points, the parametric equation defining R is substituted into the implicit equation defining S:

$$(P_x + t\omega_x)^2 + (P_y + t\omega_y)^2 + (P_z + t\omega_z)^2 - 1 = 0 \tag{1.9}$$

and then, since all variables with the exception of t are known, solved for t. Once the two resulting values t_0 and t_1 are obtained, the two intersection points I_0 and I_1 can be expressed as $I_0 = P + t_0\omega$, and respectively $I_1 = P + t_1\omega$.

1.2.2 Polygon meshes

Polygon meshes represent shapes as a composition of multiple smaller polygons connected via shared edges. The higher the number of such polygons the mesh exhibits, the higher the accuracy of the approximation of the represented shape. The most common types of polygons used to form a mesh are triangles and quadrangles. An example of a representation of a shape as a triangle mesh can be seen in Figure 1.9. The polygons are usually composed of vertices, edges connecting these vertices, and a face, which is the area bounded by the vertices and edges. This information is sufficient to describe a polygon in 3D space. Triangles are commonly used polygons to form meshes due to their efficient storage in memory and the property that the vertices of a triangle cannot be co-planar.

Various ray primitive intersection algorithms exist, for example, the Möller-Trumbore intersection algorithm [MT97] for intersecting triangles. To test a given ray for intersection with a polygon mesh, in theory, all polygons contained in the mesh in question must be tested for intersection. This is, of course, a naive and costly approach since the number of intersection tests that need to be performed is linear in the number of polygons. As such, polygon meshes are often composed of several thousand polygons, and testing them all for intersection has a decisive influence on the performance of the rendering process. Section 1.3 introduces acceleration data structures, aiming at the minimization of those intersection tests.

1.2.3 Constructive solid geometry (CSG)

The idea behind *constructive solid geometry (CSG)* is the representation of more complex geometry by the application of Boolean set operations to a manageable amount of geometric primitives. The Boolean set operators are union (logical `OR`), intersection (logical `AND`) and difference (denoted as `SUB`-operator). The geometric primitives of which CSG are composed are usually analytically described surfaces such as spheres, cones, and cylinders. However, polygon meshes can be treated as such primitives, too.

The representation of solids as constructive solid geometry offers one significant advantage: Complex geometry can be expressed via the composition of a manageable amount of primitive shapes instead of large numbers of polygons in a polygon mesh. Because testing whether a point lies inside or outside a primitive is easy, as noted in Section 1.2.1. Since the number of primitives of a composed CSG is usually lower than the number of polygons in a polygon mesh, the number of intersection tests in each render pass is comparatively low and computationally cheap.

Figure 1.10 shows the results of the application of each Boolean set operator to two sphere primitives. The resulting shapes come about by evaluating the intersections between a ray and the spheres, that have been calculated during the ray tracing procedure, according to the respective set operation. Figure 1.11 provides a visual depiction of this.

Multiple such applications of set operations with geometric primitives can be hierarchically ordered in a binary tree, which we will call the *CSG tree.* One example of such CSG tree is provided in Figure 1.12. Its leaves are associated with the geometry primitives, its interior nodes with set operations. Due to the possibility that two primitives can be translated, rotated, or scaled before a Boolean operator is applied to them, the corresponding edges of the tree can be associated with transformation information (this is indicated in Figure 1.12 with the matrix icon).

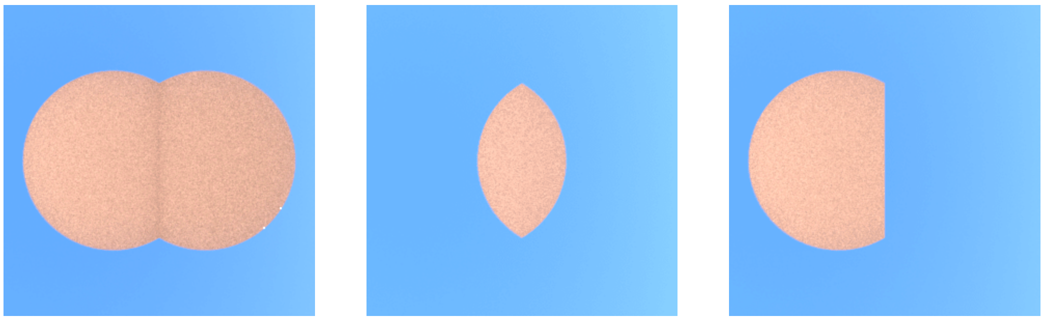

(a) Union of two spheres (`OR` operator).
(b) Intersection of two spheres (`AND` operator).
(c) Left sphere with the right sphere "subtracted" from it (`SUB` operator).

Figure 1.10 Boolean operators union (1.10a), intersection (1.10b) and difference (1.10c) applied to two sphere objects.

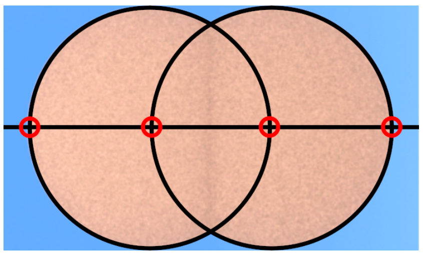
The CSG tree is utilized during ray casting for calculating the intersections between a ray and the CSG. The tree is traversed recursively. When visiting a node's child node, the ray is transformed according to the transformation information associated with the edge between node and child node. If the visited node is a leaf node, we calculate the intersection points between the ray and the solid represented by that node. If the visited node is an internal node representing a Boolean set operation, we determine two sets of intersections by recursively applying the procedure to the node's two child nodes. The individual intersection points are then inversely transformed to the current coordinate system of the node and evaluated according to the set operation represented by it.

The first algorithm for ray tracing CSG was developed by [Rot82].

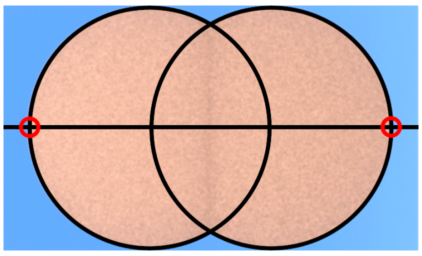
1.3 Acceleration data structures

The computational cost associated with ray tracing and path tracing algorithms has always been regarded as a "necessary evil" one faces when desiring highly realistic images. An often-cited fact is Whitted's observation that, for complex scenes, 95 percent of the time used by his algorithm is spent on intersection calculations [Whi79, p 349].

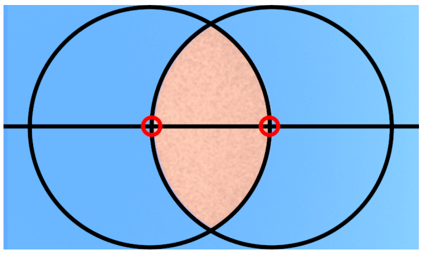
It was only a logical consequence that new ideas were introduced over time, aiming at accelerating the ray tracing process. Be it through the minimization of the number of rays cast into the scene, the development of faster intersection testing algorithms, or the minimization of ray-primitive intersection tests. *Acceleration data structures* are intended to address the latter. By utilizing these structures, one essentially trades a a performance increase for a larger mem-



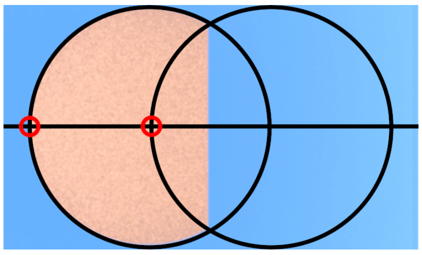
(a) In total, the ray intersects both spheres at four points.



(b) When applying a union set operation to the two spheres, the two intersection points that lie in the interior or boundary of both spheres are disregarded.



(c) Complementary to the union, the intersection set operation disregards the two intersection points that do not lie in the interior or boundary of both spheres.



(d) When a difference set operation is applied to "subtract" the right sphere from the left sphere, only the intersection points that lie in the interior or boundary of the left sphere are considered.

Figure 1.11 Selection of calculated intersection points between a ray and two spheres according to the Boolean set operations. The horizontal line illustrates the ray, and the red circles depict the intersection points. The outlines of the two spheres depict a cross-section of the 3D objects.

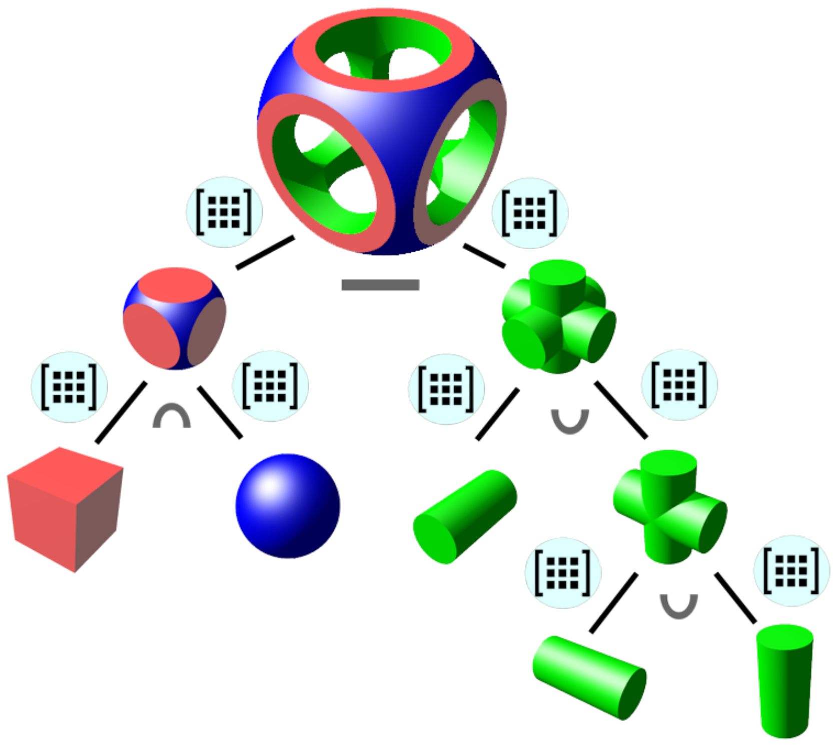


Figure 1.12 Visualization of a CSG tree hierarchy, originally created by [Zot21] and updated with a matrix icon symbolizing affine transformations of geometries.

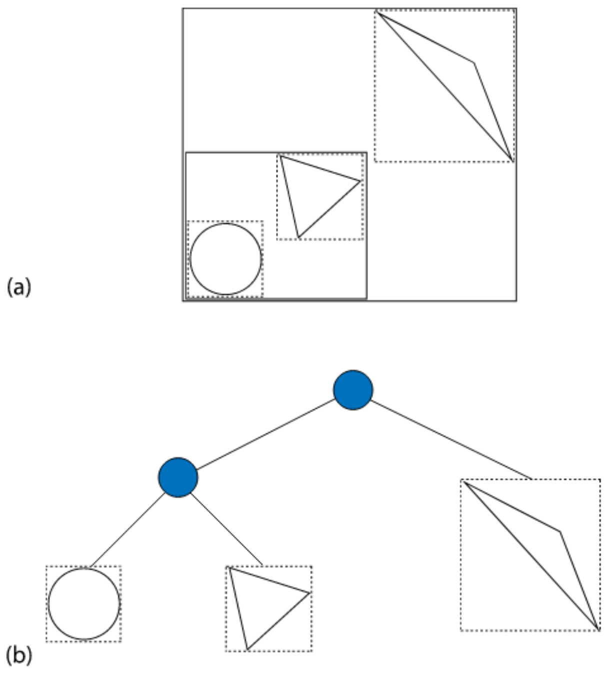


Figure 1.13 Example of a bounding volume hierarchy, published in [PJH16]. (a) represents a 2D scene, consisting of shapes enclosed with axis-aligned bounding boxes (indicated by the rectangles with dashed edges). Further bounding boxes enclose the bounding boxes of the two shapes in the bottom left and the whole scene. (b) shows the associated BVH tree structure.

ory footprint. The following subsections focus on two popular acceleration data structures commonly used: Bounding volume hierarchies and KD trees.

1.3.1 Bounding volume hierarchies

Bounding volume hierarchies (BVHs) are hierarchical structures of *bounding volumes*, aiming at the reduction of unnecessary intersection tests. Bounding volumes enclose geometries with an analytical shape, such as a sphere, a cylinder, or a box. As previously discussed in Section 1.2.1, testing for an intersection between a given ray and an analytical shape is easy. If no intersection with such volume is found, it is logical that an intersection between the ray and the enclosed geometry is not possible. Therefore, the intersection calculation between the ray and the enclosed geometry can be safely disregarded.

Bounding volumes should ideally be chosen to fit the enclosing geometry as tight as possible to reduce the number of unnecessary intersection tests further. However, a most commonly used volume is an *axis aligned bounding box*, a hyperrectangle whose sides are parallel to the axis of the coordinate system. Such boxes might not enclose the geometry as tight as possible. On the other side, testing for intersections becomes computationally cheap, and not much memory is needed to store them.

These bounding volumes can be again enclosed in a bounding volume, and thus, complex hierarchies of bounding volumes can be formed. A BVH is a hierarchy of multiple of such bounding volumes stored in a tree structure. The leaves of the tree represent the bounding boxes of the geometry contained in the scene, and the interior nodes are associated with bounding volumes enclosing the node's children. The bounding box associated with the tree root encloses the entire scene. An example of such a tree structure is given by Figure 1.13.

A BVH that is build over n primitives obtains $2n - 1$ total nodes, n leave nodes and $n - 1$ interior nodes, as noted in [PJH16].

During ray tracing, the BVH is traversed, and a cast ray is tested for intersections between the bounding box associated with the current tree node. If an intersection is found, the traversal continues in the node's children, otherwise traversing the subtree rooted at that node is disregarded.

A variety of different variants of BVHs exists. Their individual strengths and weaknesses depend on the target application (e.g., on whether it supports dynamic or static scenes) and the system architecture (e.g., on the CPU instruction set) [2].

1.3.2 KD trees

K-dimensional trees (or *KD trees*), which were introduced by [Ben75], belong to the group of *binary space partitioning trees (BSP trees).* The crucial difference between this tree structure and the BVH is that the BVH uses bounding boxes to group geometric shapes, while BSP trees recursively subdivide k dimensional space with $k - 1$ dimensional hyperplanes to group shapes.

One subdivision of space occurs when the number of geometric primitives contained in it is greater than a specified threshold. When one such subdivision takes place, space is divided into two smaller half-spaces. This procedure is repeated for each of the two half-spaces until the number of primitives in a subdivided area is smaller or equal to the threshold. The hyperplanes that partition space are stored in a tree structure. The root of the tree is associated with the hyperplane splitting the bounding volume of the entire scene. Its children to the left are the hyper-planes and geometric objects in one half-space, and the

[2]For more information on these various types of BVHs, we would like to re-direct the interested reader to the research article "A Survey on Bounding Volume Hierarchies for Ray Tracing" [Mei+21].

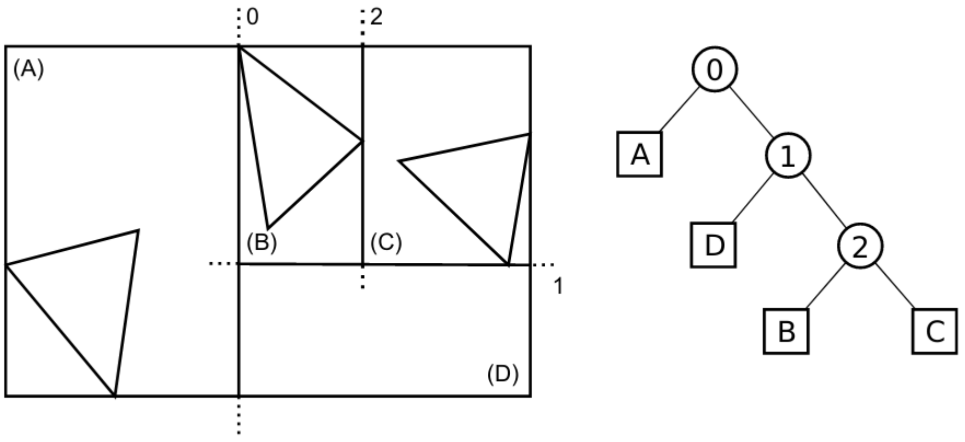


Figure 1.14 Partition of a 2D scene in subspaces A, B, C, and D with one-dimensional lines 0, 1, and 2, subdividing the scene. This figure was originally published in [HH11].

children to the right are the planes and objects located in the other half-space. The leaf nodes correspond to a list of geometries located in the area bounded by the hyper-planes associated with their parent nodes in the tree.

KD trees are a specialized case where the split-planes are perpendicular to one of the coordinate axes. In this subsection, we will only consider the subdivision of three-dimensional space with KD trees. During the recursive splitting, the axis to which the current plane is perpendicular is alternated. The tree's root represents the plane subdividing the bounding box enclosing the entire scene.

The traversal of the KD tree for calculating intersections between a ray and the scene geometry is almost similar to the traversal of bounding volume hierarchies. The crucial difference lies in the calculation of volumes bounded by split planes, as opposed to axis-aligned bounding boxes, before testing the contained geometry for intersections.

When a given ray does not intersect the box containing the entire scene geometry, there is no need to traverse the KD tree. Otherwise, we test for intersections between the ray and the two volumes bounding the half-spaces, separated by the split plane, represented by the tree root. These two half-spaces are associated with the children to the left and the right of the root node. If the ray does not intersect one such half-space, bounded by the scene bounding box and the split plane, the corresponding subtree is not further traversed. Otherwise, the same procedure is recursively applied to the child nodes until a leaf node is reached. In this case, the geometries in the list corresponding to the leaf node are tested for intersection.

A variety of construction and traversal algorithms exists for KD trees, for example the algorithm described in [WH06].

1.4 Intel's Embree framework

Despite the effect of various ray acceleration data structures and hardware optimizations, exploiting the full capabilities of modern CPUs, which exhibit different architectures and instruction sets and support different algorithms and data structures, remains challenging for ray tracing applications.

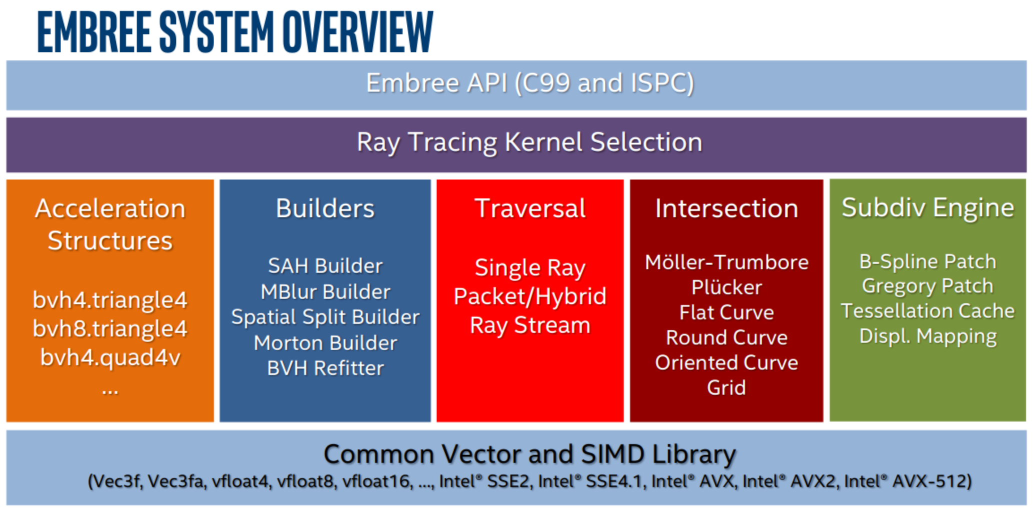
Embree is a high performance rendering framework written in C99 which addresses this problem. It offers a collection of vectorized kernels optimized for communication with CPUs that support SSE, AVX, AVX2, AVX-512, and instruction sets. The provided kernels offer a variety of components, e.g., different types of BVHs, traversal algorithms, and intersection algorithms, Figure 1.15 provides an overview. For rendering, Embree will decide on which are the best-suited components to use for building the acceleration structure. It selects these components based on the information provided by the user and on the target CPU architecture. The latter is performed by the "Ray Tracing Kernel Selection" layer of the hierarchy shown in Figure 1.15.

Interesting features of Embree are:

- Finding of the closest hit point, or alternatively any hit point,
- support for the cast of single rays, ray packets containing 4, 8 or 16 rays, and so-called ray streams of any desired number of rays,
- high-quality BVH builders,
- support for the Intel SPMD Program Compiler (ISPC) and the Intel Threading Building Blocks (TBB), and
- independence from any other graphics API such as OpenGL or DirectX

An API for the integration into existing rendering systems is provided and described in a detailed documentation [Int21]. This documentation furthermore offers a tutorial for the familiarization of the framework to new users. The name of the functions belonging to this API are preceded by the abbreviation `rtc` ("ray tracing kernels"), data types have the abbreviation `RTC` predeceasing their name. For example the variable `RTCScene` stores the virtual scene for Embree, and the function `rtcIntersect1` performs the intersection testing with a single ray.

Embree supports various geometric shapes such as triangles, quads, and certain types of curves, such as Bézier curves, B-Splines and Catmull-Rom-Splines. Another notable feature of Embree is the support of custom geometries of the rendering system it is integrated into. These will be referred to as *user-defined geometries.*


EMBREE SYSTEM OVERVIEW

Embree API (C99 and ISPC)

Ray Tracing Kernel Selection

Acceleration Structures	Builders	Traversal	Intersection	Subdiv Engine
bvh4.triangle4 bvh8.triangle4 bvh4.quad4v …	SAH Builder MBlur Builder Spatial Split Builder Morton Builder BVH Refitter	Single Ray Packet/Hybrid Ray Stream	Möller-Trumbore Plücker Flat Curve Round Curve Oriented Curve Grid	B-Spline Patch Gregory Patch Tessellation Cache Displ. Mapping

Common Vector and SIMD Library
(Vec3f, Vec3fa, vfloat4, vfloat8, vfloat16, …, Intel® SSE2, Intel® SSE4.1, Intel® AVX, Intel® AVX2, Intel® AVX-512)

Figure 1.15 System overview of Embree. This figure is a slide from the SIGGRAPH presentation *Embree Ray Tracing Kernels 3.X: Overview and Features* [Ben18].

Embree is open source and therefore publicly available. Supported platforms are (32-bit and 64-bit), Linux (64-bit), and macOS (64-bit). The latest version of Embree at the time of writing this thesis is 3.13.0.

Chapter 2

Technical details of Embree and ART

The following chapter is dedicated to the familiarization of the Embree library and the target application ART, in which Embree will be integrated. However, in this chapter, we will only consider the aspects that are relevant for our implementation. For a comprehensive overview of Embree, we would like to re-direct the reader to the Embree user documentation [Int21], further information about ART can be found in the ART Handbook [Wil+ndb] and the ART Scene File Reference Manual [Wil+nda] [3].

2.1 Ray Tracing with Embree

Embree follows a device concept, allowing for using the Embree API by different components of the image synthesis application without interfering with each other. An `RTCDevice` is created with the function `rtcNewDevice` and released via the function `rtcReleaseDevice`. Embree uses such device types to create further components, such as virtual scenes, which serve as the container for various scene geometry.

Scene Devices

A scene in Embree, represented by the data type `RTCScene`, is created via the function `rtcNewScene`, to which the `RTCDevice` is passed to as an argument. It is released via the function `rtcReleaseScene`. Different geometries can be

[3] At the time of writing this thesis, both the ART Handbook and the ART Scene File Reference Manual are still work in progress, and therefore incomplete. However, these documents are about to be completed in the near future.

attached to the scene device by the function rtcAttachGeometry, which will furthermore assign a unique ID to the geometry, and detached from the scene device by the function rtcDetachGeometry. Once an RTCGeometry is attached to the RTCScene, it can be released via the call of the function rtcReleaseGeometry.

After the attachment of the complete scene geometry, the scene is committed by calling the function rtcCommitScene to trigger the creation of Embree's internal acceleration structures. After the invocation of this function, the individual geometries cannot be edited or manipulated.

Geometries

Geometries in Embree are represented by a RTCGeometry data type. These types can be created with the function rtcNewGeometry which takes the RTCDevice and an enum specifying the geometry type (e.g., triangle, quad, user-defined geometry) as input parameters. In the case of the geometry is a triangle, quadrangle, or a type of curve, corresponding *geometry buffers* can be created and linked to the RTCGeometry by invoking the function rtcSetNewGeometryBuffer. These buffers will store information such as vertices, indices, and surface normals of the geometry.

To initialize a user-defined geometry, one has to provide a function for calculating the bounding box for the geometry, a function for intersection testing, and another function for occlusion testings. These functions are passed to Embree as callback functions. Furthermore, the number of geometric primitives, which together compose the geometry, has to be set, and a so-called *user data pointer* associated with the geometry has to be initialized. A user data pointer points to the representation of the geometry by the rendering application in memory. The user data pointer can also be associated with geometry types other than user-defined geometry. In the interior of the callback function for calculating the intersection with the user-defined geometry, the original representation of the geometry can be easily retrieved via this pointer.

User data pointers are initialized with the function rtcSetGeometryUserData. The passing of the various callback functions to Embree is done via invocation of the functions rtcSetGeometryBoundsFunction, rtcSetGeometryIntersectFunction, and rtcSetGeometryOccludedFunction.

Ray Casting

Once the RTCDevice and the RTCScene are set up, the scene geometry is attached to the RTCScene and the internal acceleration data structures have been built, nothing stands in the way of performing the actual ray tracing with Embree.

Listing 2.1 The RTCRayHit struct. The segment of code displayed here is part of the original Embree source code.

```
/* Combined ray/hit structure for a single ray */
struct RTCRayHit
{
  struct RTCRay ray;
  struct RTCHit hit;
};
```

Listing 2.2 The RTCRay struct. The segment of code displayed here is part of the original Embree source code.

```
/* Ray structure for a single ray */
struct RTC_ALIGN(16) RTCRay
{
  float org_x;          // x coordinate of ray origin
  float org_y;          // y coordinate of ray origin
  float org_z;          // z coordinate of ray origin
  float tnear;          // start of ray segment

  float dir_x;          // x coordinate of ray direction
  float dir_y;          // y coordinate of ray direction
  float dir_z;          // z coordinate of ray direction
  float time;           // time of this ray for motion blur

  float tfar;           // end of ray segment (set to hit
        distance)
  unsigned int mask;  // ray mask
  unsigned int id;    // ray ID
  unsigned int flags; // ray flags
};
```

Listing 2.3 The `RTCHit` struct.The segment of code displayed here is part of the original Embree source code.

```
/* Hit structure for a single ray */
struct RTC_ALIGN(16) RTCHit
{
  float Ng_x;          // x coordinate of geometry normal
  float Ng_y;          // y coordinate of geometry normal
  float Ng_z;          // z coordinate of geometry normal

  float u;             // barycentric u coordinate of hit
  float v;             // barycentric v coordinate of hit

  unsigned int primID; // primitive ID
  unsigned int geomID; // geometry ID
  unsigned int instID[RTC_MAX_INSTANCE_LEVEL_COUNT]; //
        instance ID
};
```

To cast rays into a virtual scene with Embree, a per ray query intersection context, `RTCIntersectContext`, has to be set up via the function `rtcInitIntersectContext`. This structure is used for the configuration of intersection flags, among other things. Subsequently, a `RTCRayHit` struct is declared. This struct is composed of an `RTCRay` struct, abstracting the ray that Embree uses to perform the intersection testing, and an `RTCHit` struct, in which information concerning the intersection point is stored. This information contains the surface normal, the barycentric UV coordinates of the point, and the geometry ID associated with the intersected geometry are stored. The `RTCRay` struct stores the ray orientation and direction, so-called `tnear` and `tfar` values, indicating the boundaries of a range of possible hit distances and other information.

The `RTCRayHit`, `RTCRay` and `RTCRay` are shown in Listings 2.1, 2.2, and 2.3.

When the target ray tracing application is generating a ray, the values of the `RTCRay` struct are updated with the ray orientation and direction. The `tnear` value is to a very small value (usually close to zero) and the `tfar` value is set to a large value. The geometry ID of the `RTCHit` struct is initialized with the macro `RTC_INVALID_GEOMETRY_ID`.

After the `RTCIntersectContext` and the `RTCRayHit` structs have been successfully initialized and updated, the ray-primitive intersection testing is performed via invocation of the function `rtcIntersect1` to which the `RTCScene`, a reference to both the `RTCIntersectContext` and the `RTCRayHit` are passed as arguments. In case of a found intersection, this function will update the `tfar` value of the `RTCRay` with the closest hit distance and the geometry ID of the

`RTCHit` with the geometry ID associated to the intersected geometry.

In the case of performing intersection testing between a user-defined geometry and a ray, these values must be updated manually inside the intersection function that was passed to Embree as a callback function.

When the intersecting testing procedure has finished, the geometry ID of the `RTCHit` will give insight on whether an intersection was found or not. If the value of this variable remains `RTC_INVALID_GEOMETRY_ID`, one can conclude that no intersection was found. Otherwise, an intersection was found, and the `RTCRay` and `RTCHit` components of the `RTCRayHit` will provide the hit distance, the coordinates of the surface normal at the intersection point, and the UV coordinates of the intersection point.

2.2 A brief introduction to ART

As indicated in the introductory chapter, ART considers its target audience computer graphics researchers interested in predictive rendering. Predictive rendering is a branch of the computer graphics field that is dedicated to the accurate prediction of object appearances under different viewing conditions (compare [Wil+09].) While the purpose of "conventional rendering" (to which we count the ray tracing techniques encountered in Chapter 1) lies in the creation of "believable" imagery to create a certain impression to an observer, predictive rendering is concerned with the synthesis of radiometrically correct images.

Its unique predictive rendering features mentioned in the introductory chapter make ART stand apart from other rendering systems. The latest version of ART at the time of writing this thesis is 2.0.3.

2.2.1 Overview of ART

ART is composed of several UNIX-like-command line applications (hence the name Advanced Rendering *Toolkit*). These applications are written in C and Objective-C.

Scenes about to be rendered by ART are described in proprietary scene files. These files, with the file extension `.arm`, contain valid Objective-C code that is compiled by ART at the beginning of a rendering job.

In the following, we provide an overview of the individual applications contained in the toolkit:

artist

This is the actual command-line application renderer, taking an ARM scene file as input and storing the raw information gathered by path tracing process in an intermediate file described in a proprietary file format.

tonemap
: This tool can be used for tone-mapping the (possibly spectral) information stored in the file being created by `artist` in order to obtain viewable results.

bugblatter
: This application creates difference images of two provided, same-sized images. These difference images can be useful when debugging computer graphics applications or comparing different rendering techniques.

polvis
: Since ART supports rendering polarization effects by storing the amount of polarized light per pixel in each pixel of a spectral image, these polarization effects can be visualized by using this tool.

For our integration of Embree, the only relevant application is `artist`. To successfully render an image with `artist` in the ARM scene file, one has to provide at least a virtual camera, the scene geometry, and so-called *action sequence*. An action sequence is a user-defined procedure that can be thought of as a pipeline. Other applications of the toolkit, like `tonemap` or `polvis`, can be invoked in these action sequences, too.

To execute the individual steps defined by an action sequence, which are referred to as *actions*, ART makes use of a single stack data structure. During the execution of one such action, one or multiple data objects are taken off the stack, manipulated according to the action in question, and placed back on the stack. One example of such an action would be creating axis-aligned bounding boxes for each object present in the scene. Here, the scene graph object is popped from the stack, bounding boxes are calculated for each geometry, then these boxes are inserted into the scene graph. Further bounding boxes enclosing these for the geometries are calculated and inserted into the scene graph. Finally, the manipulated scene graph is pushed back to the stack.

Another noteworthy detail to mention is the ability of ART to utilize multiple available processor cores to perform a rendering job. By default, ART determines the number of available cores before the path tracing. However, a desired amount of cores to perform the rendering job can be provided by invoking

```
$ artist foo.arm -j<n>
```

To achieve lock-free parallelism between the individual threads, the scene graph is copied, and one such copy is assigned to each thread.

2.2.2 Scene graph infrastructure

Some popular rendering systems (among them Mitsuba 2 and PBRT) describe a virtual scene in the following way: A scene, abstracted by a `Scene` class object, obtains a collection of geometric shapes. These shapes are represented by a `Shape` base class, from which various geometry types are derived.

This `Shape` base class offers functions for calculating bounding boxes for the described geometry and performing intersection tests between them and a ray. This is an ideal design for Embree since these shapes can be initialized as user-defined geometries. Once Embree finds an intersection with a bounding box of a particular shape, its representation in memory can be retrieved via the user data pointer and an instance function of the `Shape` class for ray-intersection testing can be called. The values of the `RTCRayHit` struct can be updated in the intersection callback function with the values.

The internal acyclic scene graph of ART diverges significantly from this design. Here, we provide an overview of different nodes in the scene graph and their functionality. The scene graph, which was assembled by ART to render the image displayed in Figure 1.10a, is shown in Figure 2.1. Table 2.1 displays its nodes and a brief description of their functionality and relation to other nodes. However, for reasons of brevity, we will only list nodes for which the comprehension of their functionality is important to follow the implementation described in the next chapter. On particularity of the scene graph is that it can be utilized by every application mentioned in the previous section.

Throughout this thesis, we will refer to the subgraph rooted at the *Bounding Box node*, which is a direct child of the *BSP Tree node* and resembling the scene geometry as *Original scene graph*. The BSP Tree node is associated with another data structure, namely the KD tree, built over the scene geometry described by the original scene graph.

Furthermore, in the scene graph shown in Figure 2.1, from one particular node of type *Combined Attributes*, arrows with dashed lines are pointing to other nodes associated, e.g., with the BRDF of the shape represented by the child node, or nodes associated with transformation information. All of the Combined Attributes nodes have these references. For reasons of clarity, these references are shown for only one Combined Attributes node in the figure.

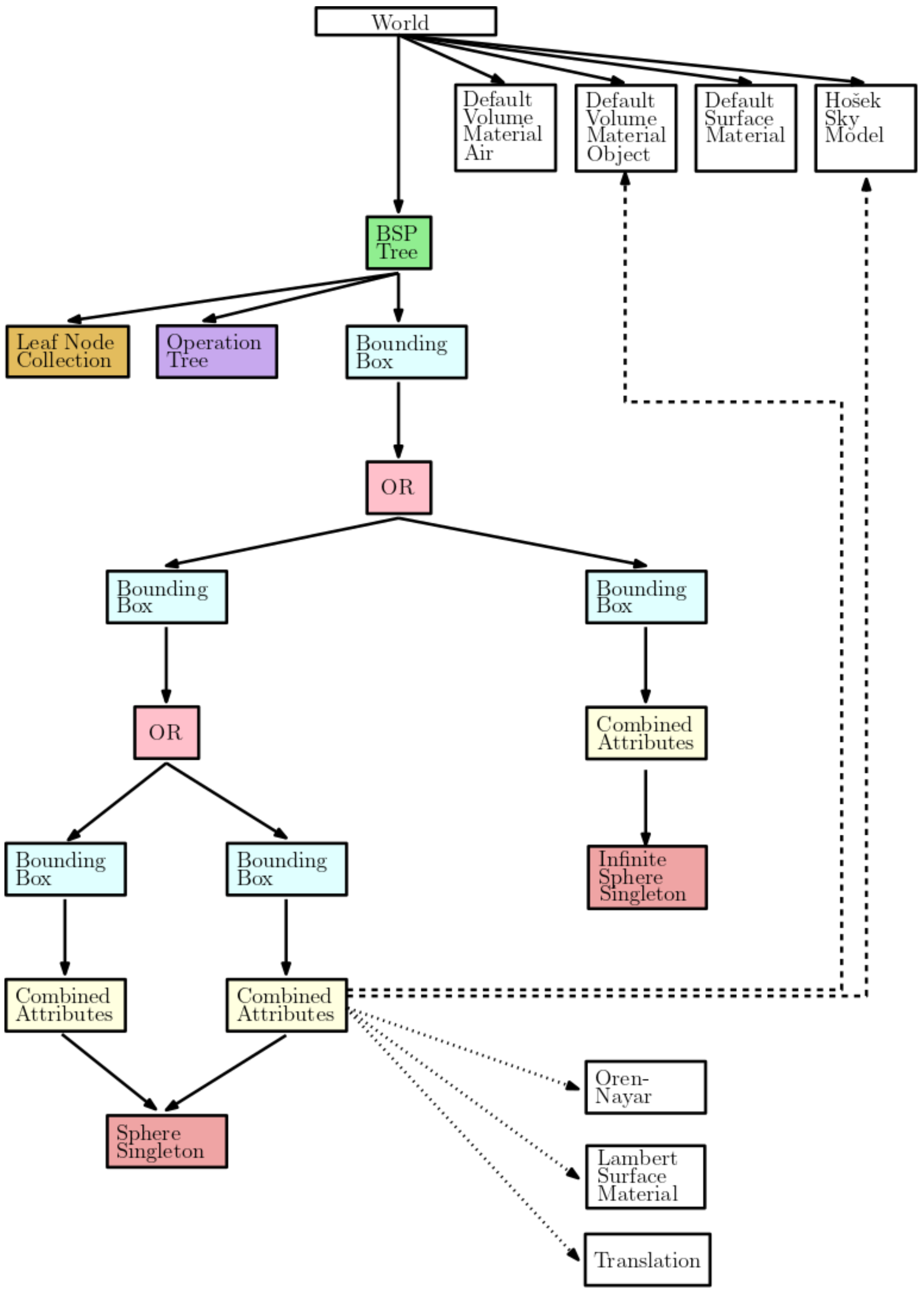


Figure 2.1 Scene graph used to render Figure 1.10a. In this graph, bounding boxes have already been inserted, and a KD tree was built over the scene. This KD tree is associated with the *BSP Tree* node in the graph.

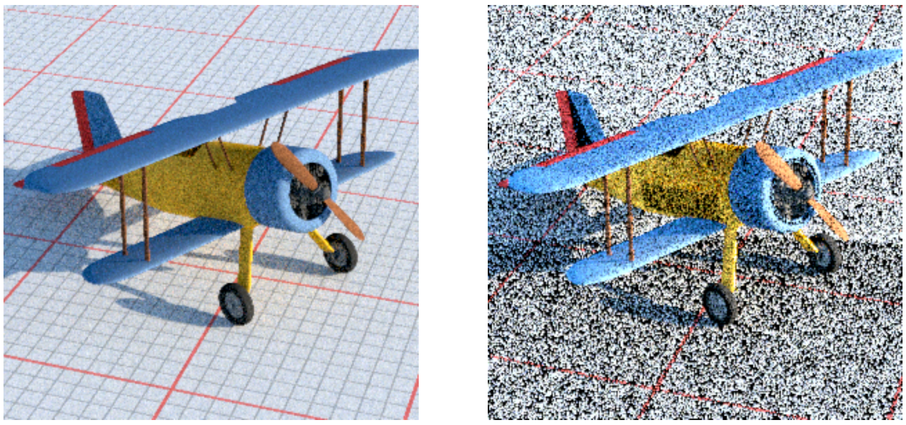
Scene graph node	Function
Sphere Singleton	*Shape nodes* are nodes that are associated with a geometric shape, such as a sphere or a triangle. Nodes of this type are always leaf nodes. In Figure 2.1, two such nodes are present: One representing a sphere, which is instantiated twice, and a so-called *infinite sphere*, used as a skydome model.
Combined Attributes	*Combined Attributes nodes* are superordinate to Shape nodes. They contain references to, e.g., the BRDF associated with the underlying shape and to a *Transformation* node describing the transformation of the shape. In Figure 2.1, one single Shape node is the child of two different Combined Attributes nodes. This means two instances of shape are created with different attributes.
Bounding Box	*Bounding Box nodes* represent bounding boxes, enclosing the components associated with the node's children.
OR	*CSG nodes* are nodes that are associated with the Boolean set operations union (OR), difference (SUB), and intersection (AND). They have two direct child nodes. The subgraphs rooted at those child nodes represent geometries on which the set operation associated with the CSG node is applied.
BSP Tree	The *BSP Tree node* is associated with the KD tree that is built over the scene. The subgraph rooted at this node corresponds to the entire scene geometry, over which the KD tree is built.

Table 2.1 Nodes in the scene graph displayed in Figure 2.1 and their functionality.

2.2.3 Intersection calculation in ART

A crucial part of rendering a scene in ART is the calculation of intersections between a cast ray and scene geometry and their subsequent evaluation. These intersections are calculated by traversing ART's internal KD tree. Functionality for this traversal is abstracted in an Objective-C class called `ArnRayCaster`. Instance variables of this class store, among other information, the ray that is cast into the scene, a reference to an intersected shape, and a so-called *traversal state*, which is a C struct containing references to, e.g., the surface and volume material of the intersected shape and its transformation information.

The intersections calculated by the KD tree traversal are stored in *intersection lists*. An intersection list, abstracted within ART with the C struct `ArIntersectionList`. This structure is essentially a linked list, storing the individ-


(a) Image rendered by traversing the KD Tree.

(b) Image renderd by traversing the original scene graph.

Figure 2.2 Comparison of rendered images by traversing ART's internal KD tree and traversing the original scene graph.

ual intersections. Such an individual intersection is represented in ART by the `ArcIntersection` class, whose instance variables are a double storing the distance from the ray origin to the intersection point and references, e.g., to the intersected shape.

There exists an alternative to the KD tree traversal in ART for calculating intersections: Namely through the traversal of the original scene graph. When considering Figure 2.1, the original scene graph is rooted at the Bounding Box node that is a child of the BSP Tree node. The classes corresponding to the Bounding Boxes node, the CSG node, the Combined Attributes node, and the Shape node provide a function called `getIntersectionList`. When this function is called from the topmost Bounding Box node in Figure 2.1, the function recursively calls the `getIntersectionList` functions of its children until the Shape nodes are reached, where the actual intersections are calculated. The `getIntersection-List` function of the CSG node first calls the `getIntersectionList` function of its left and right child, and then evaluates the found intersections according to the set operation this node represents.

Although the calculation of the intersections between a ray and the scene geometry is significantly faster with this alternative procedure, since no actual acceleration data structure is traversed, traversing the original scene graph leads to visible artifacts in the resulting image. Figure 2.2 shows a comparison between an image rendered by traversing ART's internal KD tree (Figure 2.2a), and the original scene graph (Figure 2.2b). The image rendered by the original scene graph traversal exhibits noticeable noise, and a black triangle is visible on the vertical stabilizer of the biplane in the image.

Chapter 3

Integration of Embree into ART

The following section outlines our approach to the integration of Embree into ART. The language in which the individual procedures are formulated is Objective-C since the higher-level functionality of ART has been written in this language. The Embree library itself is written in C and intended for the integration into image synthesis environments written in C/C++, which is the industry standard. However, due to Objective-C being a strict superset of C, these two languages can be intermixed seamlessly. Therefore, no issues concerning the cross-linking of C and Objective-C were discovered during the development.

3.1 Design choices

One important design choice was to abstract functionality regarding Embree in a single class, which we gave the name `ArnEmbree`, conform to the naming convention of Objective-C classes in ART. Classes of type "`Arn<name>`" belong to the category of so-called "Scene graph classes", to which, according to our own opinion, the `ArnEmbree` class belongs the closest to.

The main tasks of this class are the creation and deletion of an `RTCDevice` and an `RTCScene`, the adding of different scene geometry to the `RTCScene`, and performing the intersection calculations with Embree. This class will act as a singleton object. To quote from the ART handbook: "Apart from this struct, [the `art_gv` global variable][4] there are no genuine global variables in ART, only global constants" [Wil+ndb, Chapter 4.1.2]. The singleton object obviously contradicts this statement. The main reason for this design is to keep the functionality regarding Embree separate from the functionality of what we will from now on referring to as *Native ART*, the original Advanced Rendering Toolkit without

[4]The `art_gv` is a struct containing information that is globally accessible to the different class objects in ART. For more information on this, we refer to the ART Handbook.

Listing 3.1 Retrieval the `ArnEmbree` singleton object.

```
ArnEmbree * embree = [ArnEmbree embreeManager];
```

Listing 3.2 Verifying if Embree support was enables by the user.

```
 if( [ArnEmbree embreeEnabled] )
 {
    // ...
 }
```

Embree integration.

At an early stage of the development of our approach, it was not obvious whether Embree could be integrated into ART at all. If our work on the integration were unsuccessful, this separation would ease restoring ART to its original form. In spite of this separation, an inclusion of the `ArnEmbree` singleton class to the `art_gv` variable is a solvable problem.

Another design decision was to enable support from Embree only when the user provided the parameter flag `-e` or `-embree` when invoking `artist`, like this:

```
$ artist foo_scene.arm -e
```

Initially, the provision of a parameter flag was intended for easily switching Embree support on and off to draw comparisons between the performances of ART with and without the help of Embree. However, we kept this functionality because there is one case (revolving around the rendering of CSG composed of triangle meshes) where ray tracing with Embree is inefficient. We will turn to this circumstance in more depth when discussing the results in Chapter 4.

Once the command line arguments of `artist` are evaluated, the `ArnEmbree` singleton class object, which we gave the name `"embreeManager"`, is initialized and set up, if the parameter flag was set. Otherwise, `embreeManager` is set to `NULL`. From this point on, the ArnEmbree singleton can be retireved from anywhere in the code and at any point of the action sequence by the instruction, shown in Listing 3.1.

Furthermore, if the singleton was initialized and set up, another global Boolean variable, indicating whether Embree is enabled or not, is set to `true`. The value of this Boolean can be retrieved by calling the class method called `embreeEnabled`. An example of this is given in Listing 3.2.

3.2 The `ArnEmbree` class and extension of the `ArnRayCaster` class

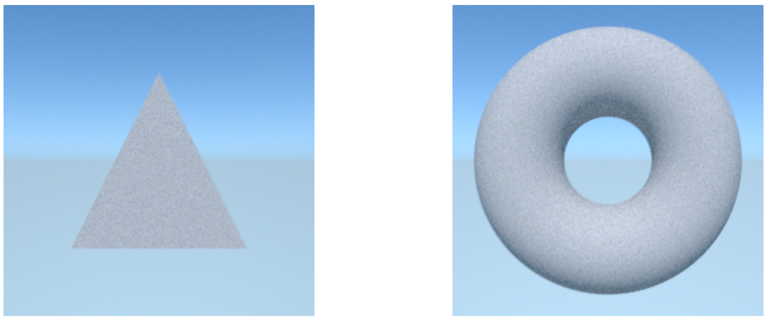
The `ArnEmbree` class is constructed the following way: Instance variables are provided to store a single `RTCDevice` and a single `RTCScene` for Embree. Although the instantiation of multiple scenes on a single device is possible with Embree, we consider only single scenes containing all scene geometry since ART does not support the instantiation of multiple scenes.

Furthermore, class methods are defined for the initialization of different geometry types for Embree, their attachment to the `RTCScene`, and finally for committing the `RTCScene`, which will trigger the build of Embree's internal spatial acceleration data structures. Due to these data structures, the creation of ART's internal KD tree is not necessary and can be discarded.

In order to perform the actual ray-primitive intersection testing with Embree, an instance method was added to the `ArnRayCaster` class, which we gave the name `getIntersectionListWithEmbree`. If Embree support is enabled, instead of traversing ART's KD tree, this method is called. In this function, a given ray is converted to an `RTCRay` and Embree's internal function for intersection testing, `rtcIntersect1`, is invoked. Since ART only supports the cast of single rays, as opposed to ray packets, functionality for casting ray packets with Embree is not considered. After the intersection testing is performed, the `tfar` value is updated with the hit distance (which will remain being set to `INFINITY` if no intersection was found). The UV coordinates, the surface normal at the hit point, and the shape associated with the hit geometry are stored in an intersection list. From this point, the ray tracing process continues as usual until the next ray is cast into the scene.

3.3 Initializing shapes for Embree

ART supports a variety of different geometrical shapes. An overview of these shapes can be found in the ART Scene File Reference Manual. The supported shapes are divided by ART into two categories: *Analytic shapes* and *Simple indexed shapes*. Analytic shapes are represented by ART as outlined in Section 1.2.1. Simple indexed shapes, on the other hand, are described by an array of vertices and indices associated with the shape in question (similar to the triangle primitives in *OpenGL*). In ART, two shapes, namely triangles and quadrangles, are considered simple indexed shapes. Figure 3.1 shows example shapes belonging to the two categories. This section describes the initialization of these shape types for Embree.



(a) An example of a simple indexed shape: a triangle.

(b) An example of an alytical shape: a torus aligned on Y-axis.

Figure 3.1 Shape types in ART.

Although it would be convenient to initialize all shapes in ART as user defined geometries for Embree, we make the distinction between *user-defined geometry* and *non-user-defined geometry.*

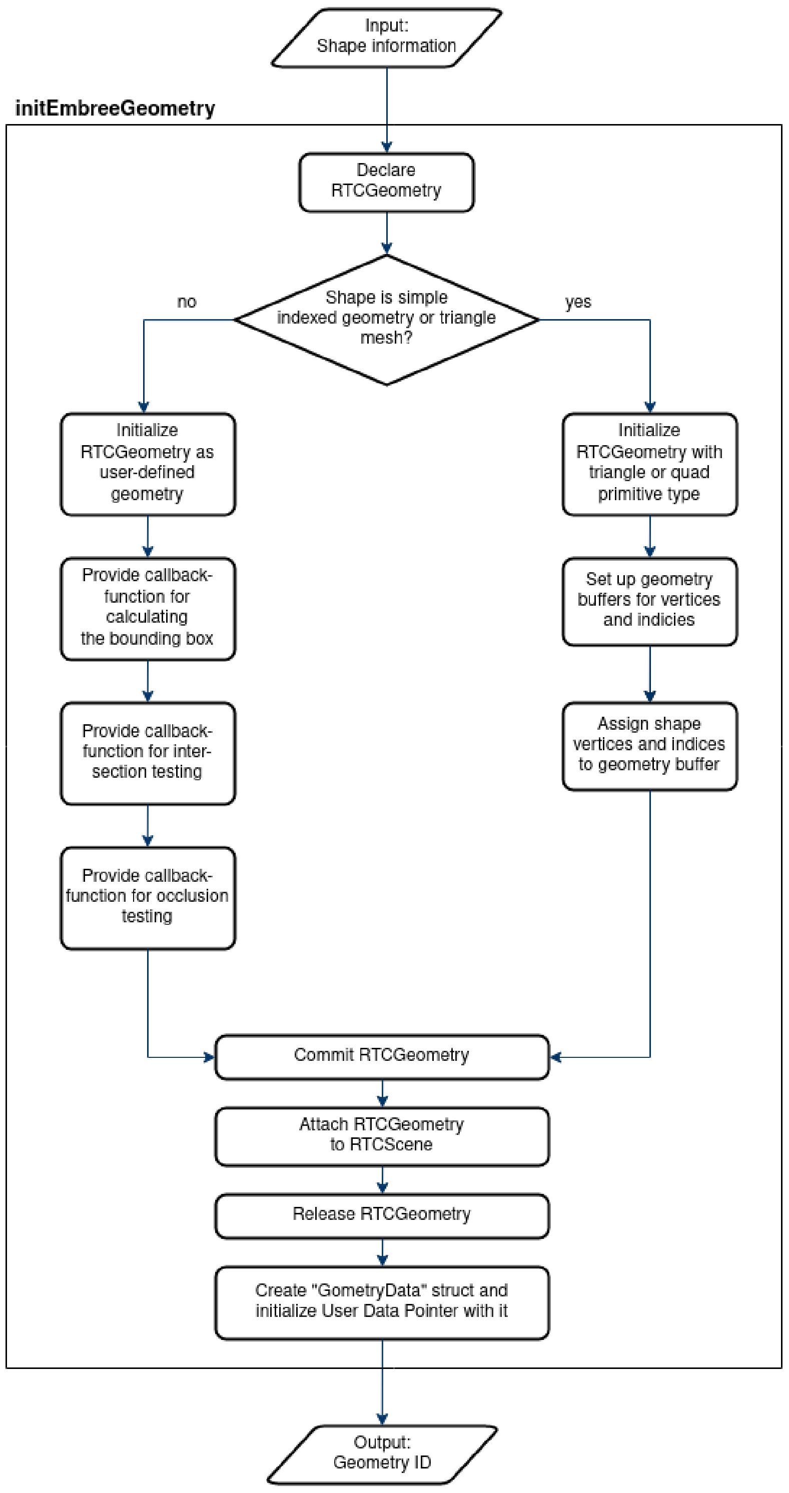
Into the first category will fall the analytically described shapes. Triangles and quadrangles will be regarded as non user defined geometry and represented by Embree's own primitive types `RTC_GEOMETRY_TYPE_TRIANGLE` and `RTC_GEOMETRY_TYPE_QUAD`. This division makes sense since the rendering of these primitive types with Embree is more efficient than the rendering user defined geometries. Furthermore, all the information needed to set up these primitive types is the vertices and indices associated with the shape. They can be easily transferred from ART to Embree.

The initialization of geometries for Embree takes place during the assembly of the scene graph from the information that has been parsed from the ARM scene file. To be precise, a particular shape is initialized for Embree when the Combined Attributes nodes and Shape nodes associated with it are created and inserted into the scene graph. After this insertion, an instance method of the `ArnEmbree` class, `initEmbreeGeometry`, is called. The shape object itself, together with the `Combined Attributes` object, and the transformation matrix, are passed as function arguments.

The `initEmbreeGeometry` function does the following:

- **Initialization of an `RTCGeometry`**

  To initialize a geometry for Embree, an `RTCGeometry` needs to be created. After initializing a geometry for Embree, it is committed via the function `rtcCommitGeometry`, attached to the `RTCScene` by invocation



Input:
Shape information
initEmbreeGeometry
Declare RTCGeometry
Shape is simple indexed geometry or triangle mesh?
no
yes
Initialize RTCGeometry as user-defined geometry
Provide callback-function for calculating the bounding box
Provide callback-function for inter-section testing
Provide callback-function for occlusion testing
Initialize RTCGeometry with triangle or quad primitive type
Set up geometry buffers for vertices and indicies
Assign shape vertices and indices to geometry buffer
Commit RTCGeometry
Attach RTCGeometry to RTCScene
Release RTCGeometry
Create "GometryData" struct and initialize User Data Pointer with it
Output:
Geometry ID

Figure 3.2 Flowchart describing the workflow of the `initEmbreeGeometry` function.

of the function `rtcAttachGeometry`, and released by calling the function `rtcReleaseGeometry`. When a particular geometry is successfully attached to the `RTCScene`, a unique identifier is assigned to it. This identifier, an unsigned integer value, is referred to as the *geometry ID*. The geometry ID is stored in an instance variable of the `Shape` or `Simple Indexed Shape` class. It allows for the retrieval of the `RTCGeometry` from Embree for alteration. However, the retrieving of `RTCGeometry` variables is only possible before committing the `RTCScene`.

- **Initialization of `GeometryData` struct and setting up the user data pointer for Embree**

  Once a shape is successfully attached to the `RTCScene`, a C struct associated with the shape is dynamically allocated and initialized. Subsequently, a user data pointer pointing to this struct is set up for Embree via the function `rtcSetGeometryUserData`. This struct, which we call `GeometryData`, stores information that is needed for calculating the intersections between a ray and a user-defined geometry in ART. It is shown in Listing 3.3.

`GeometryData` structs store the geometry ID associated with the shape and issued by Embree, ART's representation of the shape in memory, a struct called `ArTraversalState`, storing information such as the surface material of the shape, a `Combined Attributes` object, and a Boolean variable indicating if the shape is user-defined or not.

As mentioned before, user data pointers are intended to retrieve shapes in a specified callback function for ray tracing user-defined geometry with Embree. We, on the other hand, associate every shape present in a scene with such a `GeometryData` struct, even for non-user-defined geometries, solely to differentiate between these two types of shapes.

Subsequently, this struct is stored in a linked list, whose head is an instance variable of the `ArnEmbree` class. `GeometryData` structs can be extracted from this list by linear search.

3.3.1 Initialization of non-user-defined geometry

Simple index geometries and triangle meshes are initialized the following way: Inside the `initEmbreeGeometry` function, another class method with the name `initEmbreeSimpleIndexedGeometry` is called, with the shape object, the vertex set containing the vertices that describe the shape, and the transformation

Listing 3.3 C struct associated with each initialized geometry.

```
// each geometry in the scene is associated with
// this stuct, it is needed for embree to
// perform user defined geometry intersection
// calculations
typedef struct GeometryData
{
  unsigned int _embreeGeomID; // geometry id of the shape
  ArNode * _shape; // ART's shape representation in memory
  ArTraversalState _traversalState; // C struct storing, e.
        g., surface material

  ArNode<ArpRayCasting> * combinedAttributes; // node used
        for ray casting
  BOOL _isUserGeometry; // determines if geometry is User-
        defined or Non-user-defined
}
GeometryData;
```

matrix being passed as arguments.

In the interior of this function, the following steps are executed:

- **Initialization of the `RTCGeometry`**

  Depending on whether the shape passed to it is a triangle or quadrangle, the new variable is initialized either with the geometry type `RTC_GEOMETRY_TYPE_TRIANGLE` or `RTC_GEOMETRY_TYPE_QUAD`.

- **Creation of *geometry buffers***

  After this initialization is the creation and assignment of two so-called geometry buffers, one for storing the vertices and one for storing its indices that are both associated with the shape. Listing 3.4 shows how the function `rtcSetNewGeometryBuffer` is used to achieve that for a triangle shape.

  As input parameters, this function takes the `RTCGeometry` to which the geometry buffer will be linked, the buffer type, a buffer slot number, the specified format for the buffer (`RTC_FORMAT_FLOAT3` and `RTC_FORMAT_UINT3` in Listing 3.4), a byte stride argument and the number of items that are about to be stored in the buffer.

  The setup for the geometry buffers for quadrangles is almost identical.

Listing 3.4 Setting up geometry buffers for the vertices and indices of a triangle shape.

```
RTCGeometry newGeometry = NULL;
float * vertices;
unsigned * indices;

// if the shape is a triangle,
// create a new geometry buffer with type
// RTC_GEOMETRY_TYPE_TRIANGLE
if([shape isKindOfClass: [ArnTriangle class]])
{

  newGeometry = rtcNewGeometry(device,
         RTC_GEOMETRY_TYPE_TRIANGLE);

  vertices = (float *) rtcSetNewGeometryBuffer(
                        newGeometry,
                        RTC_BUFFER_TYPE_VERTEX,
                        0,
                        RTC_FORMAT_FLOAT3,
                        3*sizeof(float),
                        3
                        );

  indices = (unsigned *) rtcSetNewGeometryBuffer(
                          newGeometry,
                          RTC_BUFFER_TYPE_INDEX,
                          0,
                          RTC_FORMAT_UINT3,
                          3*sizeof(unsigned),
                          1
                          );

}
```

The only exception is that the `RTCGeometry` is initialized with the geometry type `RTC_GEOMETRY_TYPE_QUAD` and that the byte stride of the vertex buffer is four instead of three.

Once the geometry buffers are initialized, the vertices stored in the vertex set and indices associated with the shape are transferred to the vertex and index geometry buffers.

- **Transformation of the vertices according to the transformation matrix**

  In case the transformation matrix that was passed to the function is not `NULL`, the vertices, one by one, are transformed according to it before being transferred to the vertex buffer. Embree allows instancing of geometry, meaning that geometry in Embree can be translated, scaled, and rotated by referring to an instance stored in memory and applying this transformation to it. However, we decided to perform the transformation calculation for each vertex before transferring to the vertex geometry buffer because this is more intuitive and easier to perform.

The initialization of a triangle mesh, being parsed from a PLY file with the help of the "RPly" library [KS16], follows the same outline described for triangles and quadrangles, although triangle meshes do not fall into the category of simple indexed shapes. The only difference is that the size of Embree's geometry buffers is set according to the number of total triangles in the mesh. ART originally creates internal KD trees for triangle meshes. Their creation can be omitted when initializing a triangle mesh for Embree. For large triangle meshes, this omission will drastically reduce the time needed to prepare the ray tracing procedure.

After the setup of the geometry buffers, the newly created RTCGeometry is returned from this function and assigned to the RTCGeometry, created in the `initEmbreeGeometry` function, followed by the allocation of a `GeometryData` struct and the setup of its variables. The `isUserGeometry` boolean variable is set to `false`.

3.3.2 Initialization of user-defined geometry

Under this category fall any shape of ART other than triangles, quadrangles, and triangle meshes. One particular geometry that is supported by ART is a cube, which one can create by the `CUBE` macro in the ARM scene file. Although this cube geometry can be described by six quadrangles or twelve triangles, for simplicity, we treat such a cube as a user-defined geometry as well.

For initializing this kind of geometry type for Embree, we do the following:

- **Initialization of the `RTCGeometry`**

  For user-defined geometries, the corresponding `RTCGeometry` will be initialized with Embree's primitive type `RTC_GEOMETRY_TYPE_USER`.

- **Provision of a callback function for calculating the bounding box of the shape**

  We created a function called `embree_bbox`, which Embree will call before building its internal BVHs over the scene. In the function, we calculate the bounding box of the user-defined geometry and pass it to Embree. This callback function is passed to Embree via invocation of the function `rtcSetGeometryBoundsFunction`.

- **Provision of a callback function for performing the intersection testing between a user-defined geometry and an `RTCRay`**

  For this purpose we created a function called `embree_intersect`. We will describe this function in more detail in Section 3.4. This callback function is passed to Embree via the function `rtcSetGeometryIntersectFunction`.

- **Provision of a callback function for performing the occlusion testing for a user-defined geometry**

  For ray tracing purposes, only one function for intersection calculation and occlusion testing would be necessary since both operations are performed by ray casting. However, Embree strictly expects two separate functions, each with predetermined arguments. To compensate for this, we use a strategy which was inspired by the source code of Mitsuba 2: We refractor the ray tracing functionality into a fourth function called `embree_intersect_geometry`, which is called from both the `embree_intersect` and `embree_occluded` function.

Once an intersection with a bounding box enclosing a user-defined geometry is found during ray tracing, Embree will call the `embree_intersect` function, which performs the intersection testing between the ray and the shape that is associated with that bounding box.

3.4 Ray tracing with Embree in ART

Once the geometries contained in a virtual scene are initialized for Embree and Embree's internal BVH has been created, the scene can be ray cast with the help of Embree. If Embree support is enabled by provision of the `-e` flag, an instance function of the `ArnRayCaster` class with the name `getIntersection-ListWithEmbree` is called, which takes an empty `ArIntersectionList` struct as an argument.

In the body of this function, an `RTCIntersectContext` is set up and a `RTCRayHit` struct is declared and updated according to the state of the `ArnRay-Caster` object: the information with which the `RTCRayHit` stuct is being updated contains the orientation and direction of the ray that is stored as an instance variable of the `ArnRayCaster` object, and the ID associated to the ray. The `tfar` value of is initialized with Objective-C's `INFINITY` macro and the `geomID` field of the `RTCHit` struct is initialized with the macro `RTC_INVALID_GEOMETRY_ID`.

Embree utilizes single-precision floating-point numbers for its internal calculations, whereas ART uses double-precision floating-point numbers. To compensate for visual artifacts in the final image, we do not initialize the `tnear` value of the `RTCRay` with zero. Instead, we give it a little offset to prevent calculating an intersection between a secondary ray and the same shape that was already hit. We found the value $1 \cdot 10^{-3}$ to be reliable. A comparison of an image rendered with and without this offset is given by Figure 3.4.

After the update of the `RTCRayHit` struct, the actual ray tracing is performed by the invocation of the `rtcIntersect1` function. Depending on whether a bounding box of a user-defined geometry or non-user-defined geometry in Embree's internal BVH was intersected by the `RTCRay`, either our custom callback function `embree_intersect_geometry` is called by Embree, or Embree performs the intersection testing with its built-in functionality.

Once the overall rendering job successfully completed, a "clean-up" is performed. The `GeometryData` structs associated with the scene are released, followed by the release of the `RTCScene` via the function `rtcReleaseScene` and the release of the `RTCDevice` via `rtcReleaseDevice`. As a final step, the `ArnEmbree` object `embreeManager` itself is released.

3.4.1 Intersecting user-defined geometry

In case a bounding box of a user-defined geometry is intersected with an `RTCRay`, the callback function `embree_intersect_geometry` is invoked by Embree. In this function, the `Geometry Data` struct associated with the geometry in question is retrieved via the geometry user data pointer.

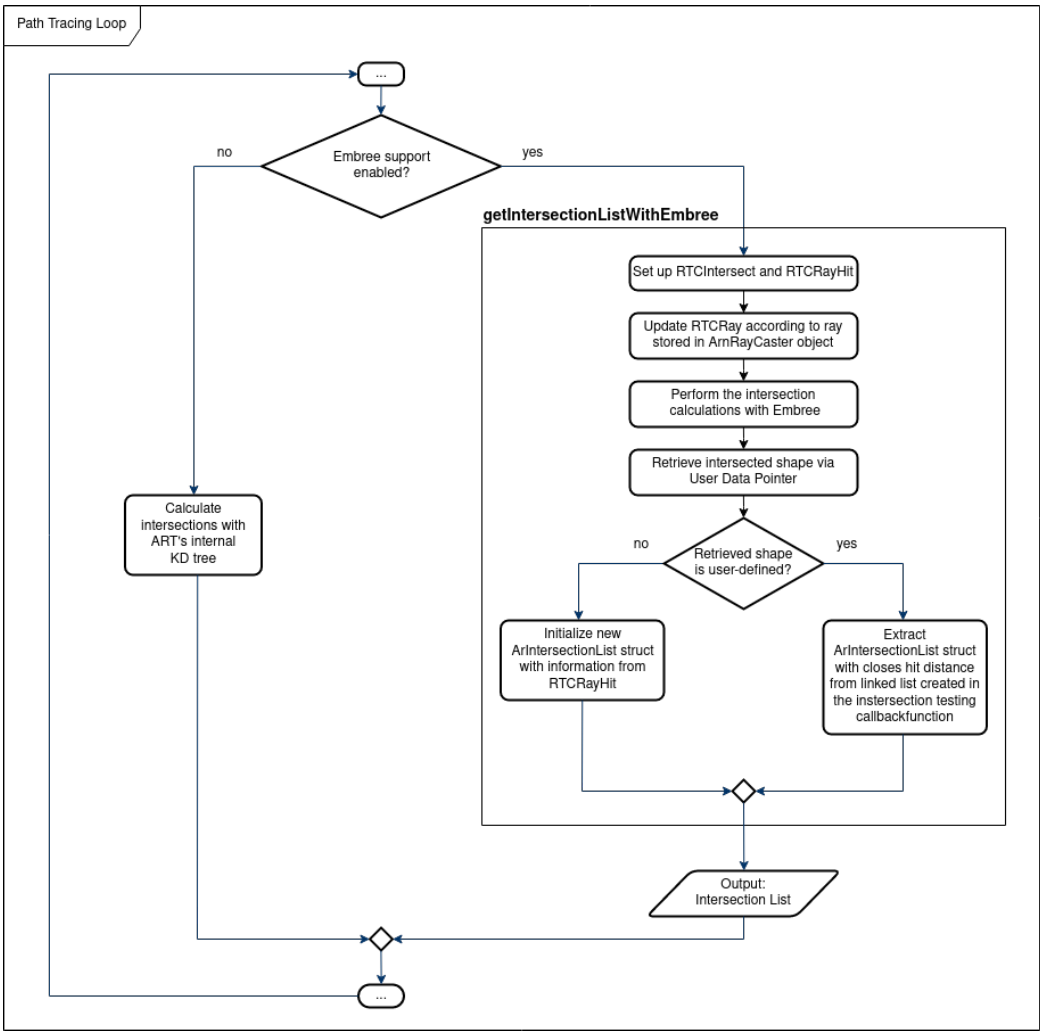
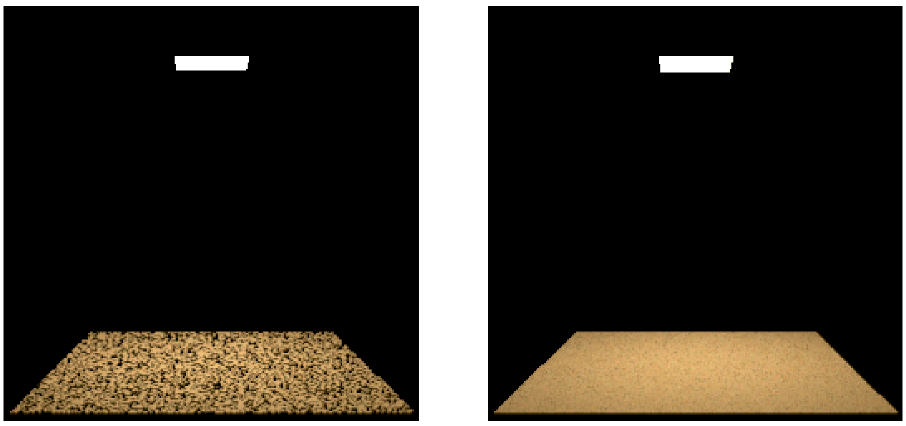

Path Tracing Loop
...
Embree support enabled?
no
yes
getIntersectionListWithEmbree
Set up RTCIntersect and RTCRayHit
Update RTCRay according to ray stored in ArnRayCaster object
Perform the intersection calculations with Embree
Retrieve intersected shape via User Data Pointer
Retrieved shape is user-defined?
no
yes
Initialize new ArIntersectionList struct with information from RTCRayHit
Extract ArIntersectionList struct with closes hit distance from linked list created in the instersection testing callbackfunction
Calculate intersections with ART's internal KD tree
Output: Intersection List
...

Figure 3.3 Flowchart the workflow of ray tracing with Embree in ART. The nodes with the three dots depict the internal procedures of ART.


(a) Result of rendering a simple scene when the `tnear` value of the `RTCRay` is set to zero.

(b) Result of rendering a simple scene when the `tnear` of the `RTCRay` is given an offset of $1 \cdot 10^{-3}$.

Figure 3.4 Artifact caused by the conversion of the hit distance of the ray from a double-precision floating-point number to a single-precision floating-point number. Due to an imprecise calculation of the intersection point, a secondary ray might intersect the same shape again.

Subsequently, an empty `ArIntersectionList` struct is declared and the intersection between ART's representation of the ray and the shape is calculated via calling the `getIntersectionList` function of the `Combined Attributes` object, which takes a reference to the empty `ArIntersectionList`, as well a reference to the `ArnRayCaster` object as input. This function will calculate the intersection points on the shape and update the `ArIntersectionList` struct accordingly. The reason why the `getIntersectionList` instance method of the `Combined Attributes` object is called, rather than the `getIntersectionList` instance method of the shape object, is that by doing so, the transformation information of the shape will be taken into account.

If no intersection was found, we return immediately from the `embree_intersect_geometry` function. Otherwise, we will update the `tfar` value of the `RTCRay` with the hit distance and the geometry ID of the `RTCHit` struct with the geometry ID associated with the intersected shape. The resulting `ArIntersectionList` is then stored in a linked list.

After the invocation of the `rtcIntersect1` function in the body of the `getIntersectionListWithEmbree` function, the geometry ID of the `RTCHit` is evaluated. If the value of this variable remains `RTC_INVALID_GEOMETRY_ID`, we conclude that no intersection was found and we return an empty `ArIntersectionList`. Otherwise, we retrieve the `RTCGeometry` that has been intersected

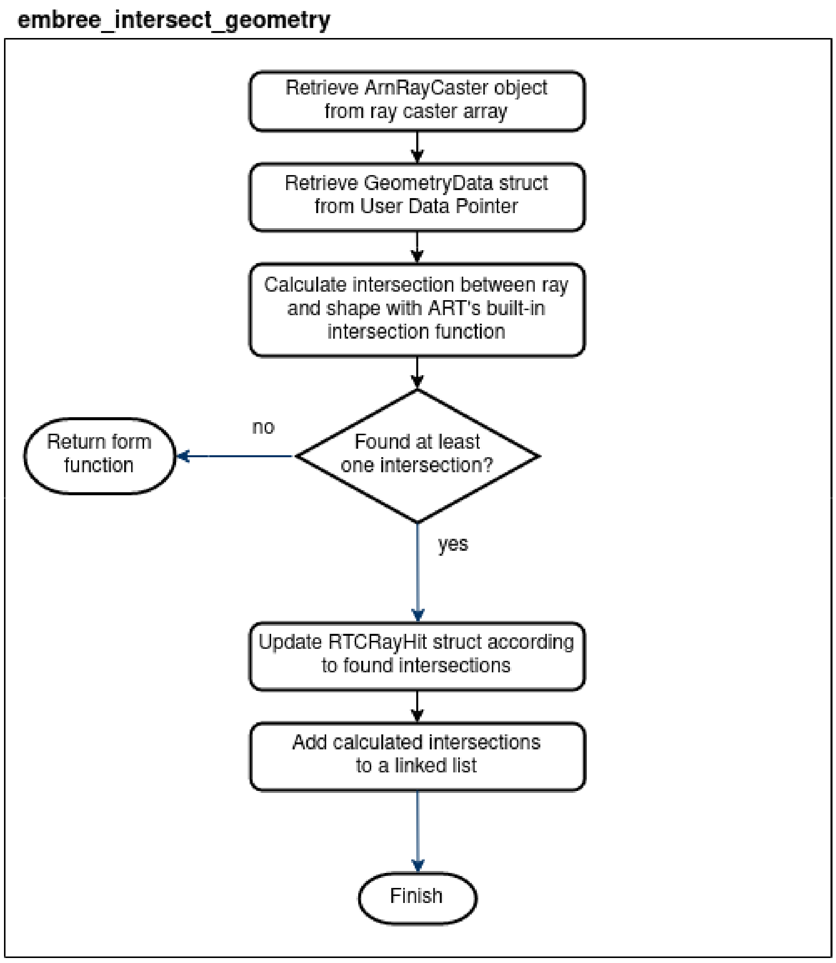

embree_intersect_geometry
Retrieve ArnRayCaster object from ray caster array
Retrieve GeometryData struct from User Data Pointer
Calculate intersection between ray and shape with ART's built-in intersection function
Found at least one intersection?
no
Return form function
yes
Update RTCRayHit struct according to found intersections
Add calculated intersections to a linked list
Finish

Figure 3.5 Workflow of the custom `embree_intersect_geometry` callback function.

with the geometry ID.

This RTCGeometry is used for the retrieval of the associated GeometryData struct via the user data pointer. Based on the Boolean variable _isUserGeometry, we check whether the intersected shape is a user-defined geometry or a simple indexed geometry.

For the latter, we initialize the empty ArIntersectionList, that was passed to the getIntersectionListWithEmbree function "from scratch" with the updated tfar value of the RTCRay, the shape object of the GeometryData struct, and the ArnRayCaster object itself. This newly initialized ArIntersectionList is then returned from the getIntersectionListWithEmbree.

In case the intersected geometry is a user-defined geometry, the linked list, in which the calculated ArIntersectionLists where placed during the intersection testing, must contain at least one ArIntersectionList struct. We locate the ArIntersectionList, whose head has the minimal hit distance, by linear search. When this ArIntersectionList is located, we extract it from the linked list and release all other ArIntersectionLists stored in it. The extracted list is then assigned to the initially empty ArIntersectionList that was passed to the getIntersectionListWithEmbree function, and ART proceeds as usual until the next ray is cast.

3.4.2 Resolving of encountered issues

The following subsection outlines two major issues encountered with the approach described in the last sections. We furthermore describe how these issues can be resolved.

Multi-threaded intersection testing for user-defined geometry

As briefly mentioned in Chapter 2.2, ART supports ray tracing with multiple threads. Before the ray tracing procedure is initiated by ART, copies of the ArnRayCaster object are created for each thread. A copy of the scene graph and the KD tree is assigned to each copy of the ArnRayCaster object to ensure lock-free parallelism.

However, the implementation of the embree_intersect_geometry callback function for intersecting user-defined geometry is not thread safe. This is due to the ArnRayCaster object that needs to be passed as an argument to the getIntersectionList function of the Combined Attributes object inside the embree_intersect_geometry function.

For a simple retrieval of the ArnRayCaster inside our custom intersection callback function, a static reference to it was originally initialized. This works fine when performing rendering jobs with only a single thread. If multiple threads are involved in the intersection computations, the procedure outlined in Subsection 3.4.1 is prone to errors since multiple ArnRayCaster objects are traversing and altering a single copy of the scene graph.

To archive lock-free parallelism, we need to retrieve the "right" ArnRayCaster copy associated with the current thread in the interior of the intersection callback function. However, we cannot utilize the user data pointer for this since the user data pointer is associated with the scene geometry, which can be intersected by multiple rays belonging to different ArnRayCaster objects on different threads at the same time.

For the placement of an ArnRayCaster into and for the retrieval of an ArnRayCaster from the ray caster array, we use an identifier of the thread associated with it. We obtain the thread ID via invocation of the gettid function, provided with the unistd header file, for accessing the POSIX operating system API. The reason we chose the function gettid over the function pthread_self, is that for n cores involved for rendering, gettid, called from n different threads, will return n strictly consecutive integer values. With the help of these values we can write a fairly simple "hash" function for placing ArnRayCaster pointers in the ray caster array and for retrieving them from it in our intersection callback function: The index of the particular ArnRayCaster pointer will be the thread ID received by the gettid function taken modulo with the counter variable.

During the beginning of the ray tracing procedure, a reference to the ArnRayCaster object associated with the current thread is added to the ray caster array, if not already been done, retrieved in the intersection callback function in constant time, and passed to the getIntersectionList function of the Combined Attributes object.

The head of the linked list, in which the collected intersection lists are stored, is an instance variable of the ArnRayCaster class, which allows for a simple retrieval of these intersection lists outside the intersect callback function.

Intersecting infinite spheres

A specific type of geometry supported by ART is a sphere with a huge radius. These *infinite spheres* are used in a virtual scene for environment lighting. Due to its radius, the length of the bounding box edges enclosing the infinite sphere is twice the infinite sphere's radius. Generally speaking, axis-aligned bounding boxes can be described by two vertices in Euclidean space, connected via the bounding box's body diagonal. In this subsection, we will refer to these vertices as *upper point* and *lower point*.

In ART, all three coordinates of the upper point are set to a huge value, represented by the double value MATH_HUGE_DOUBLE, and respectively, all three coordinates of the lower point are set to the negative of that value, - MATH_HUGE_-

Listing 3.5 Casting of a double precision floating point number to a single precision floating point number by explicit conversion.

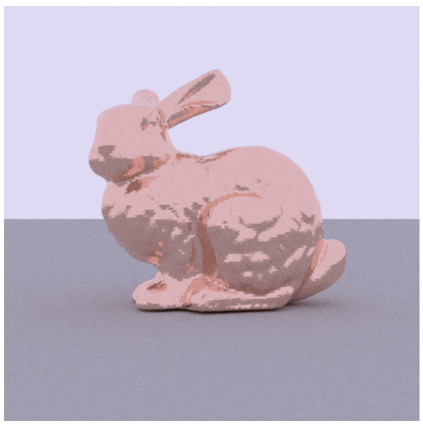
```
struct RTCBounds * bounds_o = args->bounds_o;

bounds_o->lower_x = (float) boundingBox.min.c.x[0];
bounds_o->lower_y = (float) boundingBox.min.c.x[1];
bounds_o->lower_z = (float) boundingBox.min.c.x[2];

// ...
```

DOUBLE.

As mentioned in Subsection 2.1, Embree uses single-precision floating-point numbers for its internal calculations. ART, on the other hand, uses double-precision floating-point numbers. Therefore, after calculating the double values describing the bounding box, we cast them to float values via the explicit conversion operator in C/C++ before passing them to Embree, as shown in Listing 3.5. When, during ray tracing, the tfar value of the RTCRay is set to the single-precision representation of infinity by Objective-C, the bounding box is never intersected. This is due to intersection testing being only performed in the interval $[1 \cdot 10^{-3}, \infty]$ and Objective-C's representation of infinity is "smaller" than the value MATH_HUGE_DOUBLE. The bounding box enclosing the infinite sphere is never intersected by an RTCRay.

Fortunately, ART provides a representation for infinity as a single-precision number as well, MATH_HUGE_FLOAT. Therefore, we can resolve this issue by checking in the embree_bbox callback function whether the two vertices of the calculated bounding box have the coordinates MATH_HUGE_DOUBLE (and resp. -MATH_HUGE_DOUBLE) and updating them with the value MATH_HUGE_FLOAT (and resp. -MATH_HUGE_FLOAT). By doing so, Embree can detect intersections between an RTCRay and the bounding box of the infinite sphere, and the intersection with the sphere itself can be calculated.

However, we decided to exclude Embree functionality from intersection testing with this type of shape. The reason for this lies in the further reduction of the number of unnecessary intersection calculations. If an intersection point on the infinite sphere is occluded by other scene geometry, we can ignore it. Therefore we only calculate the intersection between a ray and the infinite sphere if the ray did not intersect other scene geometry.


(a) Scene rendered with Native ART.

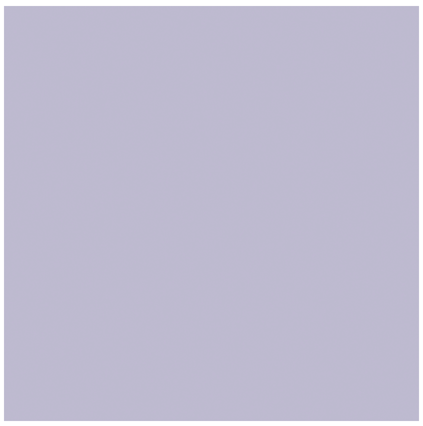

(b) Scene rendered with the approach outlined in Section 3.4

Figure 3.6 Rendered images of a scene containing a non-user-defined triangle mesh and quadrangle, and a user-defined user-defined infinite sphere, illuminating the rest of the sphere.

Consecutive intersection of user-defined and non-user-defined geometry

With the approach described in Section 3.4.1, an issue arises when ray tracing virtual scenes which contain both user-defined and non-user-defined geometry. In such scenes, a cast ray could consecutively intersect a non-user-defined geometry and a user-defined geometry. To give an example, this is the case for the virtual scene displayed in 3.6. This scene is composed of a quadrangle serving as the ground plane, the Stanford Bunny triangle mesh, which was provided by the Stanford 3D Scanning Repository [Ply], and an infinite sphere for environment lighting.

Given a ray that first intersects the PLY mesh and subsequently the infinite sphere, we noticed that through the invocation of the `rtcIntersect1` function, Embree internally calculates the intersections between the ray and the triangle mesh first and then calls the `embree_intersect` callback function to calculate the intersection point with the infinite sphere.

The problem, which arises here, is that the values of the `RTCRayHit` struct, e.g., the `tfar` value of the `RTCRay`, are updated with information corresponding to the intersection with the triangle mesh first. Afterward, the user-defined infinite sphere is intersected, and the intersection calculation is performed by our custom `embree_intersect_geometry` function. In this function, we overwrite the values of the `RTCRayHit` struct that has been previously calculated while the triangle mesh was intersected. Therefore, the information regarding the intersections between the `RTCRay` and the triangle mesh is lost, and only the `ArIn-`

tersectionList storing the intersections with the infinite sphere is present in the linked list. A result of this behavior can be seen in Figure 3.6b.

To resolve this issue, we make use of a slight modification: In the embree_-intersect_geometry callback function, before performing the actual intersection calculation between a ray and a user-defined geometry, we check whether the value of the geometry ID of the RTCHit struct remains being set to RTC_-INVALID_GEOMETRY_ID. If this is not the case, and if the head of the linked list storing the collected intersections is not NULL, we conclude that an intersection with a non-user-defined geometry must have already been calculated. We assume that is intersection already contains the closest distance to the ray origin. With the help of the geometry ID, we retrieve the associated GeometryData struct for the geometry in question from the linked list storing all the GeometryData structs linked to the scene geometry. With the information stored in this GeometryData struct, we initialize a ArIntersectionList "from scratch" and add it to the linked list storing the collected intersections.

With this approach, all the scenes on which our implementation was tested (which will be introduced in Chapter 4) could be rendered without further problems.

3.5 Rendering CSG with Embree

Unfortunately, Embree does not support rendering of constructive solid geometry direct. However, this does not mean that ray tracing CSG with Embree is completely impossible. The following section outlines three different approaches for ray tracing virtual scenes containing constructive solid geometry in ART with the help of Embree.

Since Embree is an open-source framework, one could consider rigging Embree itself for suitable CSG rendering. Nevertheless, we refrained from such an undertaking because of two reasons: On one hand, it is possible that the altering of the Embree framework would exceed the scope of this thesis. On the other hand, we want our integration to be compatible with the original Embree framework in its current and future versions.

We, therefore, consider Embree as a "black box" for which we provide information such as a ray origin and direction and receive in turn information concerning intersections with the scene geometry such as the hit distance and the surface normal at the hit point.

3.5.1 Evaluation of collected intersections according to the scene graph

In the past, an attempt for the implementation of CSG rendering with Embree was conducted by Markéta Karaffová and described in her master thesis [Kar16]. Her approach consists of the collection of intersections between the scene geometry and a ray that Embree calculates and their subsequent evaluation according to a provided CSG tree as described in Subsection 1.2.3. With the approach outlined in this subsection, we adapt the approach described in [Kar16] for ART. We were not intimidated by the increased rendering times that result from the implementation described in [Kar16] since we believed we would be able to improve it for ART.

For our approach outlined in this subsection, we will only consider CSG that are composed of user-defined geometry.

The intersections between a ray and user-defined scene geometry can be collected via storing the corresponding `ArIntersectionLists` in a linked list as described in subsection 3.4.1. One advantage of maintaining such a "list of intersection lists" as opposed to the merging the individual `ArIntersectionLists` into a single larger `ArIntersectionList`, is having the `ArIntersectionList` separated according to the shape associated with them. This is convenient since ART provides functions for evaluating two given `ArIntersectionList` structs according to the binary operators `OR`, `AND` and `SUB` as described in Subsection 1.2.3. To avoid confusion, we will refer to the `ArIntersectionList` struct as *intersection list* and to the linked list, which stores individual `ArIntersection-List` structs as *intersection linked list*.

The ray tracing of the geometric primitives proceeds as outlined in Subsection 3.4.1 Once the intersection calculation with Embree has finished, we retrieve the associated `GeometryData` struct of the intersected geometry. We then evaluate the collected intersections stored in the intersection linked list according to the original scene graph.

During the evaluation, the subgraph rooted at the Bounding Box node that is a direct child of the BSP Tree node (compare Figure 2.1) is traversed until the leaves representing the shape are reached. Due to the reason that intersections between the ray and some shapes have already been calculated previously by Embree, we first check whether an intersection list associated with the shape in question is already present in the intersection linked list. If this is the case, the corresponding intersection list is located in the intersection linked list by linear search, extracted from it, later evaluated according to the CSG tree, which is, in our case, the original scene graph.

Since ART supports instancing of geometry, meaning that multiple instances of the shape with different associated Combined Attributes nodes exist in the scene, we use the Combined Attributes node as a unique identifier to retrieve the right intersection list from the intersection linked list. Therefore, we add a reference to the `Combined Attributes` object of a shape to the `ArIntersectionList`. This `Combined Attributes` object is associated with the shape whose intersections with a ray are stored in the `ArIntersectionList` struct.

After the evaluation, the intersection list, whose head intersection has the minimal distance to the ray origin, is extracted from the intersection linked list, and the ray tracing procedure precedes as usual.

Limitations of this approach

Although it is possible to render CSG with the approach outlined in this subsection, we are aware that it is still far from being optimal. We have not implemented this functionality for non-user-defined geometry. When Embree calculates the intersections between a ray and a non-user-defined geometry, it returns only one intersection, either the closest or an arbitrary one [5].

The whole original scene graph is traversed to evaluate the intersections between a ray and a CSG represented by only a subgraph. Theoretically, only this subgraph would need traversing. However, even those subgraphs representing a particular CSG in the scene graph can be large and complex. To give an example, Figure 3.7 shows a rendered image of the "Villa Rotonda". The entire villa comprises only two CSG, which are constructed from a total number of 1,255 primitives.

Even evaluating the found intersections according to the subgraph associated with only one of the two CSG would be time-consuming. We realized the following: Even if we worked on optimizing the procedure outlined in this subsection, the lead we gained through Embree over Native ART would be compensated for by the subsequent scene graph traversal. It even could be that the performance of the overall ray tracing procedure of ART would be decreased for complex scenes, such as the Villa Rotonda scene in Figure 3.7.

Since the motivation behind the goal of this thesis, namely the integration of Embree into the CSG rendering framework ART, was the acceleration of ART's ray tracing procedure, we decided that further optimizations of this approach would not be profitable.

[5] However, [Kar16] describes a method of how all encountered intersections can be retrieved from Embree.

Figure 3.7 Rendered image of the Villa Rotonda model.

3.5.2 Initializing the complete CSG as user-defined geometry

The increased render times that resulted from the previous approach served as a motivation for developing different procedures for ray tracing CSG with Embree. We want to avoid a subsequent scene graph traversal and at the same time still treat Embree as a black box.

The idea of the approach described in this subsection lies in the initialization of the whole constructive solid geometry as a user-defined geometry instead of the geometric primitives it is composed of. The bounding box for such a CSG will be calculated with a function provided by ART and passed to Embree. If this bounding box would be stored in Embree's internal BVHs, intersected by an `RTCRay` during the ray casting procedure, ART's internal structures would be used for the continuative traversal to calculate the intersections with the geometric primitives. This undertaking seems like a satisfactory compromise between ART and Embree.

For our new approach, we define the *topmost CSG node* in the scene graph as the root node of the subgraph that represents the CSG. The geometric primitives represented by the leaf nodes of this subgraph are the geometric primitives of which the CSG is composed. For example, when considering the scene graph, assembled by ART to render the scene shown in Figure 1.10a, which itself is shown in Figure 2.1, the topmost CSG node of the CSG is the "`OR`-node". The single leaf of the subgraph rooted at that node is the shape node that represents a sphere object.

Listing 3.6 Updated GeometryData struct for CSG rendering.

```
// each geometry in the scene is associated with
// this stuct, it is needed for embree to
// perform user defined geometry intersection
// calculations
typedef struct GeometryData
{
  unsigned int _embreeGeomID; // geometry id of the shape
  ArNode * _shape; // ART's shape representation in memory
  ArTraversalState _traversalState; // state of the
        RayCaster

  ArNode<ArpRayCasting> * _combinedAttributes_or_csg_node;
        // node used for ray casting
  BOOL _isUserGeometry; // determines if geometry is user-
        defined or non-user-defined
}
GeometryData;
```

When during the initial assembly of ART's internal scene graph such a topmost CSG node is encountered, a class method of the ArnEmbree class, initEmbreeCSGGeometry, is called, which initializes a CSG as a user-defined geometry for Embree according to procedures explained in Subsection 3.3.2.

GeometryData structs are created for CSG, too. The struct itself was slightly adapted for CSG rendering. As can be seen in Listing 3.6, we renamed the variable storing a reference to the Combined Attribute node to _combinedAttributes_or_csg_node. Both the Combined Attributes node object and the topmost CSG node derive from the ArNode base class, and through the implementation of the ArpRayCasting protocol, they provide functionality for calculating bounding boxes and performing intersection testing. When a CSG is getting initialized as a user-defined geometry, a reference to the topmost CSG node is assigned to the _combinedAttributes_or_csg_node variable instead of a reference to the Combined Attributes object.

A flag gets activated after a successful initialization of a CSG for Embree during the scene graph assembly. When the traversal of the scene graph continues to the leaf nodes representing the geometric primitives of the CSG, they do not get initialized as user-defined geometry if this flag is activated.

During the intersection calculation by the embree_intersect callback function, the getIntersectionList function associated with the _combinedAttributes_or_csg_node object is called. Depending on this node being a Combined Attributes node or a topmost CSG node, the rendering procedure diverges: For a `Combined Attributes` node, the intersections are calculated directly on the shape, taking transformation information into account. For a topmost CSG node, the intersections with the underlying primitives are calculated by traversing the sub scene graph rooted at that node, as outlined in Section 2.2.3, only with the difference that with this approach, only a subgraph of the original scene graph is traversed.

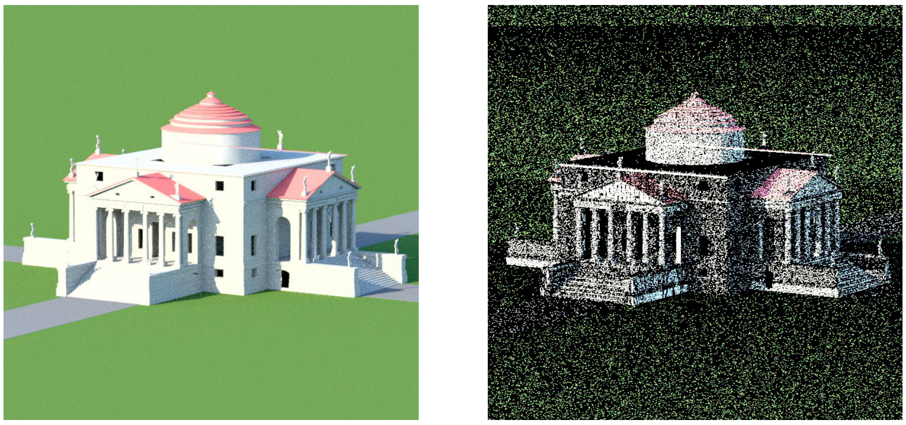
During the scene graph assembly for scenes with environment lighting, ART connects the infinite sphere with the rest of the geometries in the scene with a union operator. This is undesirable since our implementation would treat the entire geometries contained in the scene as one CSG. We can compensate for this by checking if a given topmost CSG node that is about to be initialized for Embree has an infinite sphere as a direct child. If so, we are not initializing this topmost CSG node for Embree.

With this approach, scenes containing constructive solid geometries can be successfully rendered, and the performance of the ray tracing process increased noticeably compared to the previous approach. Furthermore, the artifacts that result from rendering a scene ART by traversing of the original scene graph, as described in Subsection 2.2.3, such as the visible noise, are not present in images rendered with this approach. We cannot give a definite answer on why these artifacts are disappearing when rendering scenes with this approach. However, we believe this resolution must be related to the only partial traversal of the original scene graph.

Limitations of this approach

An inconvenience of this approach is that the fact that Embree treats the entire CSG as a single geometry must be taken into account by a user when modeling a scene for ART. For example, Figure 1.4 shows rendered images of the Cornell Box. For rendering the scene with Native ART, one could apply a union operator to the geometry of the box itself and the geometry describing the area light source (which is depicted as a box itself). Our approach, however, would interpret these two geometries as a single CSG and traverse the original scene graph to find intersections between them and a given ray. It would be better to describe the box of the Cornell Box and the area light as two disjoint geometrical entities.

An issue was encountered when rendering a specific CSG scene: The Villa Rotonda scene, shown in Figure 3.7. Figure 3.8a shows the final image, rendered by the described approach. The figure shows the villa with its roof missing (compare Figure 3.7). We have to admit that we were not able to resolve this issue yet. However, rendering the scene with Native ART by traversal of the original scene graph results in the same artifact (besides the visible noise). Due to this and the fact that we did not encounter this issue with other scene files containing CSG



(a) Villa Rotonda scene rendered with the implementation outlined in this subsection.

(b) Villa Rotonda scene rendered with Native ART by traversing the original scene graph.

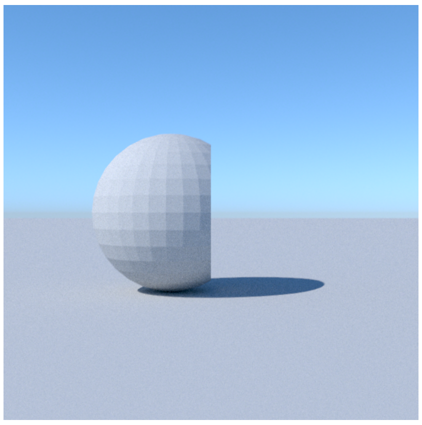
Figure 3.8 Artifact in the Villa Rotonda scene: The villa is missing its roof.

(compare Chapter 4), we assume that this is the issue is due to a bug occurring during the scene graph assembly for this scene in ART.

Furthermore, the rendering of CSG that is composed of at least one triangle mesh leads to artifacts. As mentioned, triangle meshes are associated in ART with individual KD trees that accelerate the intersection calculations solely between a ray the particular triangle mesh. When we initialize a triangle mesh for Embree, we omit the construction of individual mesh KD trees because they are not needed when ray tracing triangle meshes in general. However, when treating a CSG constructed from triangle meshes as a user-defined geometry and thus using ART's internal structures to calculate the intersections with the primitives, we depend on these individual KD trees for triangle meshes. Resurrecting these individual KD trees alone is not enough to resolve the artifact since the transmission from the original scene graph to the individual KD Tree at the Shape node representing the triangle mesh is not seamless.

3.5.3 Creation of KD trees for CSG

The artifact shown in the rendered image of the Villa Rotonda scene, which is visible in Figure 3.8a, as well as the artifacts that arise when rendering CSG that is composed of triangle meshes, as shown in Figure 3.9, motivated us to develop further our approach outlined in the previous section. The initial idea for resolving these artifacts was to build ART's internal KD tree over the entire scene and, when a bounding box of a CSG is intersected by Embree, to continue the traversal


(a) Scene rendered with native ART by traversing the internal KD tree.

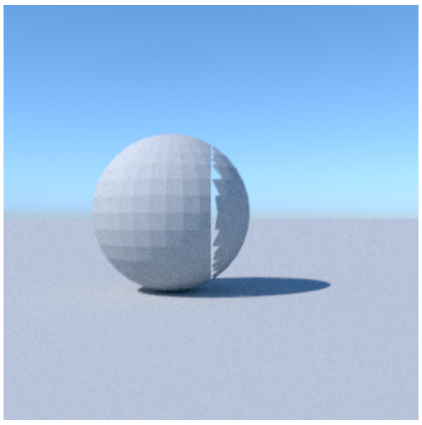

(b) Artifact in the scene rendered with the approach outlined in this section.

Figure 3.9 CSG composed of triangle meshes. The figures show a scene with two spheres described as triangle meshes. The right sphere is "subtracted" from the left sphere via the Boolean set operation `OR`. The scene can be regarded as the counterpart of the scene shown in Figure 1.10c for triangle mesh primitives.

to the CSG primitives in the subtree of the KD tree describing the spatial area in which the CSG is located. However, associating a Shape node in the scene graph to the area subdivided by the split planes of the KD trees, in which the shape is located, is not obvious since multiple geometries can be located in such an area.

The approach outlined in this subsection, which can be regarded as an extension for the approach described in the previous subsection, was inspired by the individual KD trees for the triangle meshes and the work of Petr Zajiček, documented in his master thesis [Zaj12]. In this thesis, Zajiček describes the application of KD trees to scenes containing CSG.

For the scenes containing CSG, instead of constructing one KD tree that subdivides the bounding box of the entire scene, we construct multiple smaller KD trees for subdividing the bounding box of the CSG present in the scene. Therefore, we add a Boolean variable to the `GeometryData` struct shown in Listing 3.6, indicating whether the associated geometry is a CSG or not. Before committing the `RTCScene` that will trigger the construction of Embree's internal BVH, we iterate through the linked list storing all `GeometryData` structs that are associated with the scene geometry. With the help of the `GeometryData` struct, we check whether the associated geometry is a CSG. If so, we calculate the bounding box that encloses the CSG and construct a KD tree that further subdivides the bounding volume. To calculate the bounding box and the KD tree, we use functions provided by ART. Furthermore, we add a reference to the root of the constructed KD tree to the `GeometryData` associated with the CSG.

During ray tracing with Embree, we retrieve the `GeometryData` corresponding to the intersected shape and verify if that shape is a CSG or not. If not, the ray tracing procedure continues as outlined in Subsection 3.4.1. Otherwise, we traverse the associated KD tree beginning at the root node, calculating the intersections with the CSG primitives and storing the resulting intersection list in the intersection linked list.

By traversing the individual KD trees to calculate intersections with CSG primitives, the issue regarding the Villa Rotonda scene described in the previous section is resolved [6]. Furthermore, we are able to render CSG that is composed of triangle meshes.

Limitations of this approach

A disadvantage of this approach is that it depends on the internal KD trees associated with triangle meshes by ART. Their construction was originally omitted when initializing triangle meshes for Embree. However, by constructing those mesh KD trees for this approach, the decrease of the time needed to prepare the scene for ray tracing is compensated.

[6]Figure 4.1c shows the scene rendered with the additional KD trees

Chapter 4

Results and discussion

Our approach of integrating Embree into ART, as outlined in the previous chapter, was tested on a variety of scene files. This chapter provides an overview and analysis of the performance of our implementation. It is divided into three sections. Subsection 4.1 discusses the execution time of ART when rendering scenes containing constructive solid geometries. A comparison is drawn between our two approaches for realizing CSG operations with Embree, the initialization of an entire CSG as a user-defined geometry and the traversal through the original scene graph or a dedicated KD tree. We will exclude the provision of results concerning the rendering of CSG through the collection of intersection points and their subsequent evaluation, outlined in Subsection 3.5.1. This exclusion is due to the significantly decreased rendering performance resulting from this approach, and thus, making it impractical.

Furthermore, we decided to test our implementation on virtual scenes, using mesh geometry, to analyze and compare the performance of rendering non-user-defined and user-defined geometries.

Section 4.2 provides the results on various scenes rendered by a hybrid implementation combining the approaches outlined in Subsections 3.5.2 and 3.5.3.

In Section 4.3, we test and evaluate our implementation on scenes exclusively composed of triangles with varying amount of geometry.

The following experiments were conducted on an Asus N551JX laptop with a quad-core Intel Core i7−4720HQ processor clocked at 2.6 GHz and 8 GB of RAM. The images shown in this chapter were rendered in ART with Embree support at a resolution of 700x700, a path length of 20, and 128 samples per pixel. The internal calculations regarding the image synthesis of these scenes were performed via multi-threading with eight threads.

4.1 Evaluation of our integration for scenes containing CSG

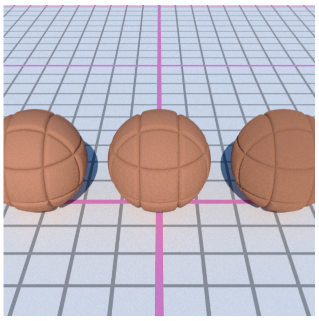
We tested the functionality of our implementation concerning CSG rendering with Embree on six scenes, which were made publicly available by the developers of ART [7]. All the scenes mentioned in this section contain an infinite sphere, serving as a skydome to light the scene.

For this evaluation, our implementation was tested on the following scenes:

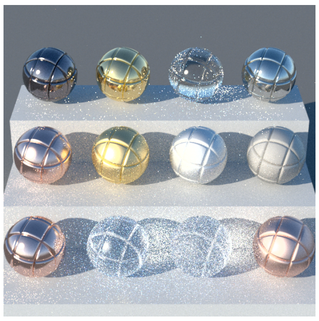
- The scene shown in Figure 4.1a consists of a single CSG, a procedurally modeled shell, and multiple cubes aligned to form a checkerboard pattern. The CSG, namely the shell, is composed of a total amount of 2,144 sphere primitives. The shell is modeled by arranging the spheres in spiral form and decreasing order according to their sizes. The largest sphere at the beginning of this sequence of spheres is "subtracted" by the `SUB` operator from the rest of the spheres. Then, the next largest sphere is "unified" by the `OR` operator with its preceding sphere in the sequence. This procedure is repeated for the remaining spheres in the sequence.
- Figure 4.1b shows the rendered image of a scene composed of a cylinder serving as the ground on which three so-called *grooved spheres* are placed. The grooves on the sphere result from applying a `SUB` operator to a group of six tori and the sphere in question. The purpose of this scene is to showcase ART's implementation of the Oren–Nayar reflectance model [ON94] with different roughness grades.
- We have already encountered the Villa Rotonda scene in Subsection 3.5.1. As mentioned there, the model of the Villa Rotonda is composed of two CSG with a total number of 1,255 primitives.
- The scene displayed in Figure 4.1d is composed of twelve grooved spheres that are placed on three deformed cubes with different heights that together form a staircase. Furthermore, a cylinder serves as the ground. The initial purpose of this particular scene was the demonstration of the implementation of the Torrance–Sparrow reflectance model [TS67] with varying roughness grades.
- Figure 4.1e shows a rendered image of a scene composed of a cylinder acting as the ground, and a biplane, composed of multiple CSG. The total

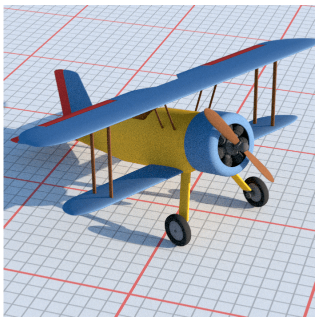
[7] Most of the scenes shown in this chapter can be found in the `Gallery` folder of the ART repository, which is submitted together with this thesis as an electronic attachment. Scenes or 3D models that are not taken from this folder will be cited.

(a) Shell


(b) Oren-Nayar Sphreres

(c) Villa Rotonda


(d) Torrance-Sparrow Spheres


(e) Parked Biplane

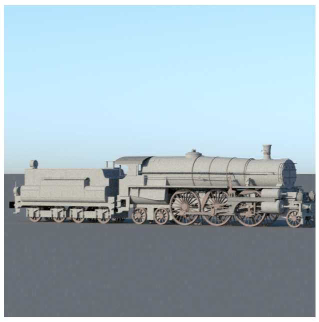

(f) Locomotive

Figure 4.1 Scenes containing CSG, rendered with our implementation. The image of the Villa Rotonda shown in Figure 4.1c was rendered by traversing KD trees associated with the CSG in this scene. Unfortunately, the issue described in Subsection 3.5.2 remains when rendering the scene by traversing the original scene graph.

number of topmost CSG nodes in the scene graph assembled by ART for this particular scene amounts to 28. The total number of geometric primitives of the 28 CSG is 336.

- Lastly, the rendered scene shown in Figure 4.1f shows a model of an Austrian steam locomotive. The single model is composed of multiple CSG, a total amount of 354 topmost CSG nodes are present in the scene graph associated with this scene The amount of geometric primitives present in the scene is 3,594.

The results we obtained from these tests show that rendering CSG geometry by traversing the original scene graph is competitive with Native ART. For the scene shown in Figure 4.1d, the speedup resulting from the support of Embree is only marginal. However, the increased acceleration shown in the table for the scene shown in Figure 4.1b is indeed noteworthy. We want to emphasize that the

Scene	#Topmost CSG nodes	Native ART	Org scene graph	KD tree	Org scene graph speedup	KD tree speedup
Figure 4.1a	1	2,611.75 sec	2,437.81 sec	4,151.01 sec	**6.66 %**	**-58.94 %**
Figure 4.1b	3	791.71 sec	522.97 sec	528.77 sec	**33.94 %**	**33.21 %**
Figure 4.1c	2	738.25 sec	629.03 sec	765.44 sec	**14.79 %**	**-3.68 %**
Figure 4.1d	12	910.56 sec	896.97 sec	902.61 sec	**1.49 %**	**0.87 %**
Figure 4.1e	28	522.58 sec	498.25 sec	546.79 sec	**4.66 %**	**-4.63 %**
Figure 4.1f	354	312.94 sec	272.08 sec	280.83 sec	**13.06 %**	**10.26 %**

Table 4.1 Comparison between the performances of Native ART and ART with Embree support. The performance of both methods of traversing the original scene graph and traversing the dedicated KD tree is compared to the performance of Native ART.

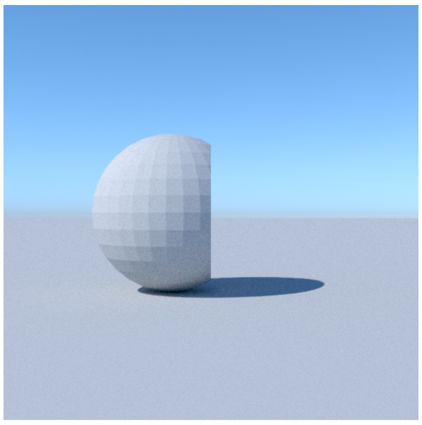
overall acceleration shown in the table is not just thanks to Embree but also the fast traversal of the subgraphs of ART's interior scene graph representing the CSG in the scene.

The results we obtained by traversing KD trees that are associated with CSG are highly scene-dependent. For the scenes shown in Figures 4.1b, 4.1d, and 4.1f, this approach is comparable to the approach of traversing the original scene graph. However, for scenes shown in Figures 4.1a, 4.1c, and 4.1e the rendering time increased compared to Native ART. The decrease seen in the table for Figures 4.1c and 4.1e can arguably be regarded as competitive with Native ART. Nevertheless, the decrease of performance regarding the rendering of the scene shown in Figure 4.1a is significant. These three scenes have in common that they contain CSG being constructed from a large number of geometric primitives. Therefore, these CSG are associated with complex KD trees whose traversal impacts the performance. In contrast, the locomotive model shown in Figure 4.1f, which is constructed by a large number of primitives, too, is modeled by the assembly of multiple CSG instead of one single large one. This explains why we did not experience a decrease in performance regarding this scene.

We conclude that our approach of traversing KD trees associated with CSG does work well with CSG composed of a manageable amount of geometric primitives but does not work well with more complex CSG.

Special case: Rendering CSG composed of triangle meshes

For testing our implementation on scenes that contain CSG, which are composed of triangle meshes, we modeled two scenes. Figure 4.2a shows the rendered image of a scene, composed of a quadrangle and two spheres that are described by triangle meshes. The right sphere is "subtracted" from the left sphere by apply-


(a) Scene rendered with native ART by traversing the internal KD tree.

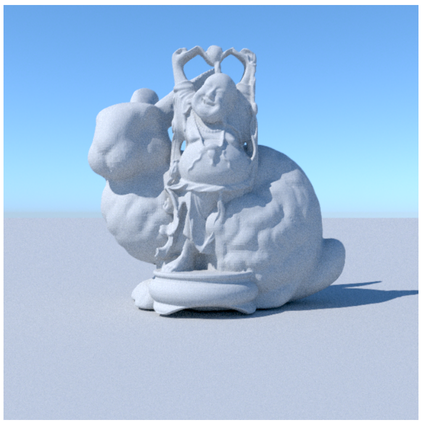

(b) Artifact in the scene rendered with the approach outlined in this section.

Figure 4.2 CSG composed of triangle meshes. The figures show a scene with two spheres described as triangle meshes. The right sphere is "subtracted" from the left sphere via the Boolean set operation `OR`. The scene can be regarded as the counterpart of the scene shown in Figure 1.10c for triangle mesh primitives.

ing the difference set operation. Figure 4.2b shows the rendered image of a scene that contains again of a quadrangle, and two triangle meshes provided by the Stanford 3D Scanning Repository [Ply], namely the Stanford Bunny, composed of 69,451 triangles and the Happy Buddha, composed of 1,087,716 triangles. The results are shown in Table 4.2. As described in Subsection 3.5.2, ray tracing these types of CSG is not possible with our approach of traversing the original scene graph. Therefore, the results shown refer to the approach of traversing individual KD trees associated with the CSG.

The results in the table indicate a significant decrease in ray tracing performance when rendering CSG composed from triangle meshes. This drop-off can be explained the following way: Embree was originally developed for rendering scenes containing complex geometry, being described by a high number of primitives. Therefore, it does not perform well with scenes containing a small number of geometries. In our scenes, only two geometries are present, a plane and the entire CSG. Furthermore, the combining of Embree's BVHs and ART's internal KD trees for such simple scenes does not accelerate the intersection calculation process. It, in fact, complicates it.

We do not recommend using our approach to render CSG composed of triangle meshes at the current stage.

Scene	Native ART Preparation	Native ART Ray Tracing	Embree Preparation	Embree Ray Tracing	Ray Tracing Speedup
Figure 4.2a	0.09 sec	237.25 sec	0.09 sec	284.86 sec	**-20.07 %**
Figure 4.2b	4.58 sec	376.20 sec	61.82 sec	709.77 sec	**-88.67 %**

Table 4.2 Comparison between the performances of Native ART and ART with Embree support. The time needed for preparing the ray tracing process by constructing the internal acceleration data structures and the time needed for the ray tracing process itself.

4.2 Hybrid implementation and evaluation

After examining the results of our tests concerning the rendering of CSG described in the previous section, we decided to abandon our approach of calculating the intersection points by traversing KD trees associated with the CSG in the scene. For now, we accept the drawback concerning the Villa Rotonda scene missing its roof, as outlined in Subsection 3.5.2, as a known issue.

However, we cannot completely abandon the approach of building and traversing individual KD trees for CSG since scenes with CSG that are composed of at least one triangle mesh strictly depend on these. Therefore, in our final implementation, we use a hybrid technique merging our approaches described in Subsection 3.5.2 and Subsection 3.5.3. During the preparation of the scene for ray tracing in ART, the scene graph is assembled. If, during the assembly, a topmost CSG node is encountered, we check if at least one of the leaves of its subtree is associated with a triangle mesh. If this is the case, a flag associated with the CSG in question is activated. Whenever such a flag is activated for a CSG, the internal KD tree for the triangle mesh primitive is built, and an individual KD tree is built for the CSG. When a ray intersects this particular CSG during the ray tracing procedure, the intersections will be calculated by the traversal of the associated KD tree.

The intersection calculations between rays and CSG that are not constructed of at least one triangle mesh are calculated by traversal of the scene subgraph rooted at the topmost CSG node corresponding to the particular CSG.

This section provides the results of tests conducted with this final implementation.

In the following, we provide an overview of the scenes used for testing the overall performance of ART with Embree support:

- The scene shown in Figure 4.3a contains a model of the Macbeth Col-

Scene	#Geometries	Native ART	Embree	Speedup
Figure 4.3a	26	382.70 sec	361.80 sec	**5.46 %**
Figure 4.3b	20	387.72 sec	227.74 sec	**41.26 %**
Figure 4.3c	13	397.46 sec	389.72 sec	**1.95 %**
Figure 4.3d	38	674.91 sec	634.52 sec	**5.98 %**

Table 4.3 Comparison between Native ART and our hybrid implementation on scenes that contain various types of geometries. In the table, triangle meshes and CSG are considered one geometry.

orChecker, a color calibration tool. This model comprises one larger deformed cube serving as the frame and 24 cubes depicting the colored samples. The chart is placed on a cylinder.

- Figure 4.3b shows a typical Cornell Box scene with two additional image textures applied to the rear of the box and two a quadrangle on the right. The scene is composed of a total number of 20 non-user-defined geometries.
- The scene shown in Figure 4.3c was created as part of the development of a model for describing the emission from glowing solid objects, described in [WW11]. It consists of twelve spheres and a cylinder on which the spheres are placed.
- Figure 4.3d shows an image of a scene that was originally published in [WH13]. The scene consists of CSG, user-defined geometries, and a triangle mesh. We will refer to this scene as *Exoplanet scene* since it was used to showcase illumination on earth-like exoplanets.

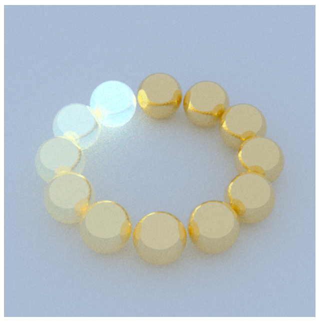
The results shown in Table 4.3 are similar to those presented in Section 4.1. The result obtained by rendering the scene shown in Figure 4.3b stands out from the remaining results. The performance speedup is due to this scene being modeled entirely with non-user-defined geometries, which are initialized with Embree's proprietary primitive types. Rendering non-user-defined geometry with Embree is more efficient than rendering user-defined geometry.

However, this increase in performance comes with a significant drawback. The rendered image exhibits visible noise at the edges between the ceiling of the box and the three walls. This artifact can be seen in Figure 4.4a. Although we cannot give a definite answer to what caused this artifact, we believe it is due to one of the two following reasons.

- When ray tracing non-user-defined geometry, double-precision floating-point numbers are cast into single-precision floating-point numbers, resulting in inaccuracies.

(a) Macbeth ColorChecker

(b) Cornell Box with texture mapping


(c) Glowing spheres

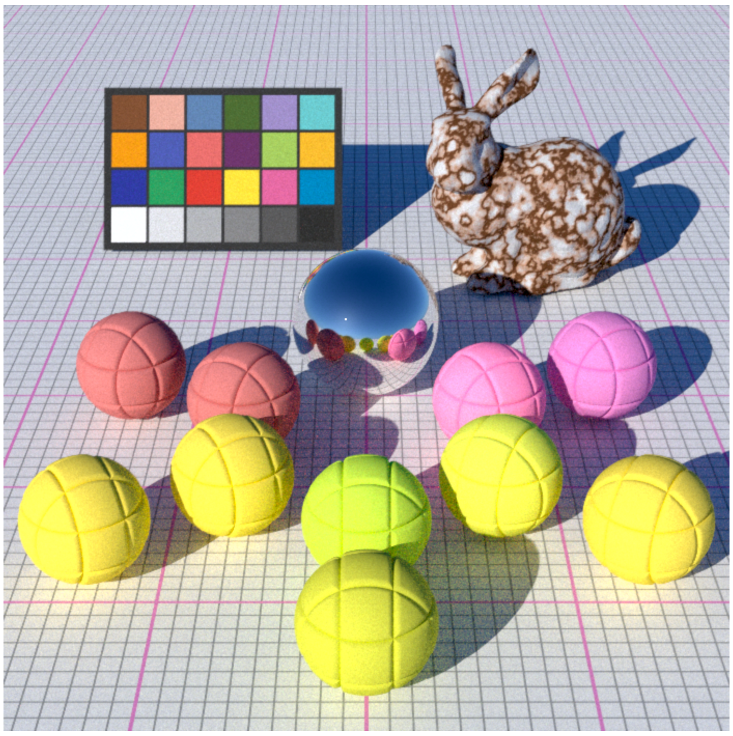
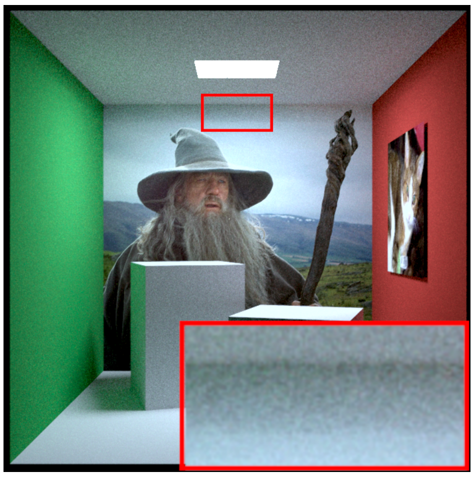
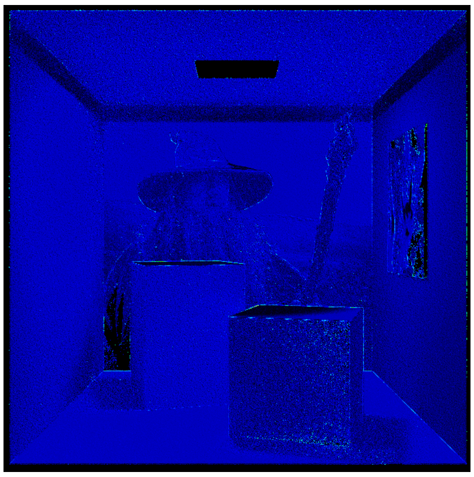

(d) Exoplanet scene

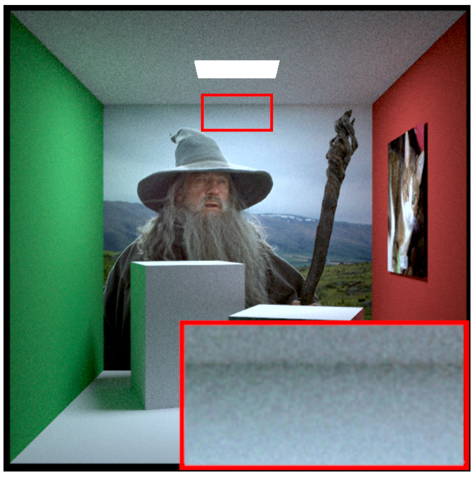
Figure 4.3 Scenes containing various types of shapes: CSG, user-defined geometry, and non-user-defined geometry.



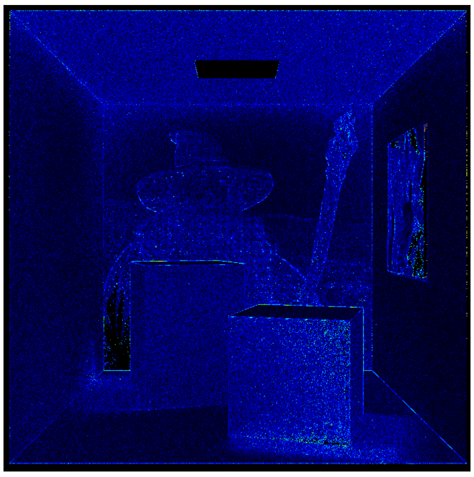
(a) Rendered image of the Cornell Box scene with the individual quadrangles being initialized with Embree's proprietary primitive types.



(b) Difference image between the image shown in Figure 4.4a and the corresponding image rendered by Native ART.



(c) Rendered image of the Cornell Box scene with the individual quadrangles being initialized as user-defined geometries.



(d) Difference image between the image shown in Figure 4.4c and the corresponding image rendered by Native ART.

Figure 4.4 Images of the Cornell Box scene rendered with different approaches.

Scene	Native ART	Embree	Speedup
Figure 4.3b	454.60 sec	292.25 sec	**35.71 %**

Table 4.4 Comparison of the performance of Native ART and ART with Embree support with regard to the Cornell Box scene shown in Figure 4.3b. The quadrangles, of which the scene is composed were initialized as user-defined geometries.

- The quadrangle used for the light source and the ceiling of the Cornell Box are coplanar. It seems that if an incident angle between the ray that is intersecting the light source and the light source is lower than a certain threshold, the contribution of light for that ray is disregarded.

One can work around this issue by initializing the quadrangles, of which the scene is composed, as user-defined geometries for Embree. A result of this approach can be seen in Figure 4.4c. We tested the rendering performance of this approach concerning this specific scene. The obtained result is shown in Table 4.4. The performance of rendering this scene, with its contained shapes initialized as user-defined geometries, is still increased compared to the performance of Native ART. However, the resulting speedup is lower than the one obtained when having the shapes initialized as non-user-defined geometry.

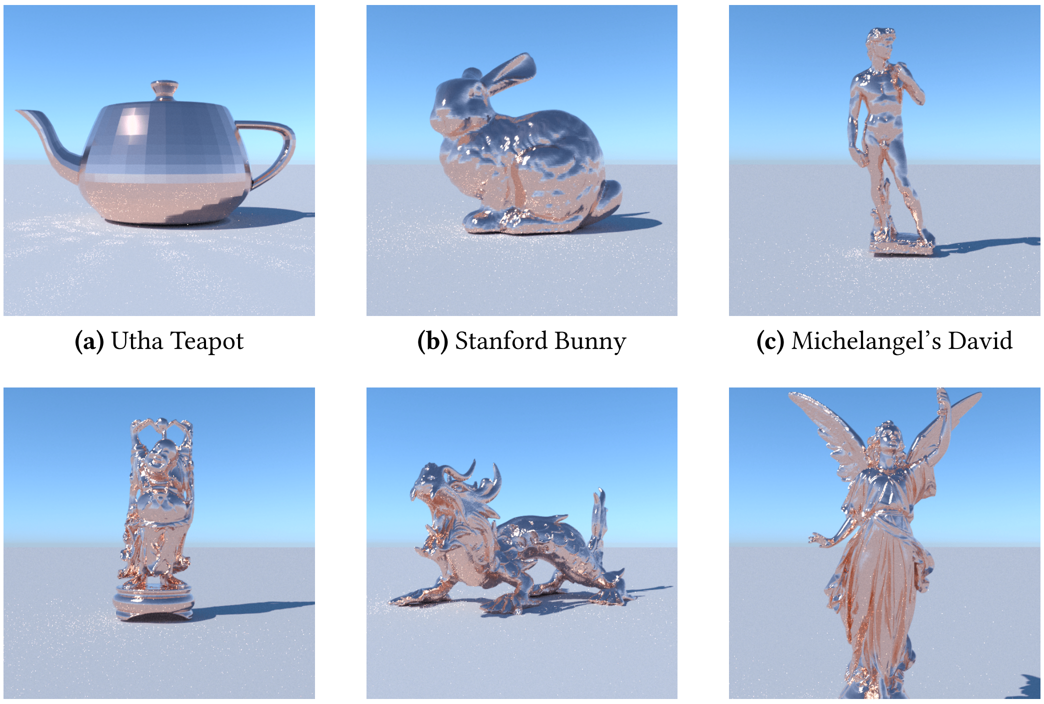
4.3 Evaluation of our implementation for scenes containing triangle meshes

Since the rendering of large triangle meshes is the most crucial use case for Embree, we tested our implementation on various triangle meshes.

Each scene shown in Figure 4.5 is composed of a quadrangle serving as ground, an infinite sphere acting as a sky dome for illumination and a single triangle mesh, that are loaded from a PLY file and assigned with a material associated with the Torrance–Sparrow reflectance model.

The following models were used for our tests:

- The **Utah Teapot** (4,032 triangles), provided by Ben Houston [Hound], shown in Figure 4.5a
- The **Stanford Bunny** (69,451 triangles), provided by the Stanford PLY repository, shown in Figure 4.5b
- **Michelangelo's David** (366,011 triangles), provided by Jerry Fisher [Fis15], shown in Figure 4.5c


(a) Utha Teapot **(b)** Stanford Bunny **(c)** Michelangel's David

(d) Happy Buddha **(e)** Asian Dragon **(f)** Lucy

Figure 4.5 Scenes for triangle meshes

- The **Happy Buddha** (1,087,716 triangles), provided by the Stanford PLY repository, shown in Figure 4.5d
- The **Asian Dragon** (7,219,045 triangles), provided by the Stanford PLY repository, shown in Figure 4.5e
- **Lucy** (28,055,742 triangles), provided by the Stanford PLY repository, shown in Figure 4.5f

The results in Table 4.5 show a general increase in the performance of ART when supported by Embree. Another advantage of our implementation is the drastic speedup when building the acceleration structure before rendering.

However, we could not render the Asian Dragon and Lucy meshes with Native ART on our local machine. This is because these meshes are large, and so is ART's internal KD tree for grouping the individual triangles into spaces bound by split planes. For triangle meshes composed of a number of triangles higher than a certain threshold, the required memory needed for the construction of the mesh KD tree exceeds the available system memory, which, in our case, results in a termination of the program by the Linux kernel.

Scene	Native ART Preparation	Embree Preparation	Native ART Ray Tracing	Embree Ray Tracing	Ray Tracing Speedup
Figure 4.5a	0.29 sec	0.04 sec	353.83 sec	304.26 sec	**14.01 %**
Figure 4.5b	4.28 sec	0.25 sec	379.02 sec	305.00 sec	**19.53 %**
Figure 4.5c	18.40 sec	1.85 sec	302.52 sec	254.03 sec	**16.03 %**
Figure 4.5d	57.28 sec	6.48 sec	351.92 sec	276.00 sec	**21.57 %**
Figure 4.5e	(no data)	64.43 sec	(no data)	304.29 sec	(no data)
Figure 4.5f	(no data)	302.19 sec	(no data)	327.59 sec	(no data)

Table 4.5 Comparison between the performances of Native ART and ART with Embree support regarding the rendering of triangle meshes.

At this point, we would like to emphasize that our implementation allows for the rendering of much more complicated scenes than Native ART on the same hardware.

Conclusion

This thesis has shown how to integrate Intel's high-performance raycasting library Embree into the CSG rendering framework ART, whose internal structures and core mechanics diverge significantly from those of other common image synthesis systems, such as PBRT or Mitsuba 2.

We have described how to initialize different geometry types for Embree, namely shapes described by vertices and indices (such as triangle meshes) with Embree's own primitive types and analytical surfaces as user-defined geometry, and have outlined how intersections can be calculated with the help of Embree.

Despite Embree not directly supporting CSG rendering, we support this functionality in ART by initializing a given CSG, composed of multiple primitives, as a single user-defined geometry for Embree. Embree's internal BVH is traversed during the ray tracing process until it intersects the bounding box enclosing the CSG in question. From there, a subgraph of ART's internal scene graph, which is associated with the particular CSG, is traversed to calculate the intersections with the primitives of which the CSG is composed. This procedure proved itself as a satisfactory compromise between ART and Embree's target application: The main advantage of using the Embree library, namely the acceleration of the ray tracing process could be preserved for the vast majority of virtual scenes, on which our approach was tested on.

Another notable accomplishment of our implementation is that the time needed for constructing ART's internal acceleration data structures is drastically decreased for scenes containing large triangle meshes. The reason for this is the omission of constructing internal KD trees for triangle meshes when initializing it with Embree's internal triangle primitives.

However, we have to admit that our approach is not entirely free from flaws. These are addressed in the next section together with suggested ideas for their resolution.

Our implementation will soon become publicly available with the release of the new version of ART.

Future work

We do not claim that our approach is the ideal integration of Embree into a CSG rendering framework, nor do we claim that our implementation is the best, concerning, e.g., performance and building times of interior data structures. In this section, we provide an overview of known issues, some of them discussed in previous chapters, and suggest potential methods of resolution:

- **Rendering CSG composed of triangle meshes**

  The most significant drawback of our implementation is the inefficient rendering of CSG composed of at least one triangle mesh. ART's interior structures need to be built and traversed for the intersection calculations when treating such CSG as a single user-defined geometry. Therefore, all functionality regarding Embree is omitted. However, we strongly believe that resolving this issue is possible. One suggestion of us is, when a Shape node corresponding to a triangle mesh is encountered during the original scene graph traversal, to cast a secondary "helper `RTCRay`" and use Embree to find intersections along the ray segment that is intersecting the triangle mesh's bounding box.

- **Investigating the original scene graph traversal when rendering CSG**

  In Subsection 3.5.2, we mentioned an issue we encountered while rendering the Villa Rotonda scene by traversing the original scene subgraphs rooted at the two topmost CSG nodes of the scene graph. As mentioned before, we believe that this issue results from traversing the subgraphs during rendering and may not necessarily be caused by our implementation. From all the scenes on which we conducted our experiments, only the Villa Rotonda scene exhibited this artifact. Nevertheless, it can be possible that this issue arises with other scenes, too. A verification of this procedure being stable would be desirable.

- **Consecutive intersection of user-defined and non-user-defineg geometry**

  In Subsection 3.4.2, we described an issue that arises when consecutively intersecting user-defined and non-user-defined geometry with Embree and how we resolved it. The modification outlined in this subsection resolved this issue for the scenes on which our overall implementation was tested (compare Chapter 4). However, we do believe this issue will remain in more complex scenes, where a large number of user-defined and non-user-defined geometries are intersected in an arbitrary order. The obvious solution for these scenes would be the initialization of all shapes supported by the rendering system as user-defined geometries at the cost of a decrease in performance time.

- **"Obvious" optimizations**

  Our implementation utilizes two linked list data structures, one in which the `GeometryData` structs that are associated with individual scene geometry, and one in which the intersections between a ray and the scene geometry are stored. The location for a specific item in the lists is performed via linear search, which might decrease the performance of rendering scenes containing a high number of geometries. Furthermore, nodes inserted in the linked list storing the intersections are dynamically allocated on the heap during the ray tracing process, which might negatively impact the performance. A solution consists of the replacement of these linked lists with more sophisticated data structures and query algorithms. Concerning the collection of intersections, Embree provides a function, `rtcSet-GeometryIntersectFilterFunction`, with which a callback function is dedicated to the filtering of the collected intersections can be passed to Embree. We did not implement this callback function due to the return of just a single intersection. We needed all intersections to evaluate them according to the scene graph. After we abandoned this approach, we did not implement this callback filter function because we focused on implementing the two other approaches on rendering CSG, described in Subsections 3.5.2 and 3.5.3.

Personal note to the potential user from the author

The main purpose of the work described in this thesis is (besides the author's graduation) our desire to provide functionality that is of actual use for both computer graphics researchers and computer graphics enthusiasts using ART. In Chapter 4, we have shown that our integration of Embree into ART is competitive with native ART, and in some cases, can indeed accelerate the rendering process of virtual scenes.

Despite the overall positive results described in Chapter 4, we cannot completely rule out that bugs will arise during the rendering of your own custom scenes. Therefore we kindly ask you to report any encountered issues via an email to the ART development team (the contact email address can be found on their website, which is https://cgg.mff.cuni.cz/ART/about/). Any feedback and critique are welcomed as well. Thank you very much in advance!

We sincerely hope that our work will be useful to you!

Bibliography

[App68] Arthur Appel.
"Some techniques for shading machine renderings of solids".
In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference.* 1968, pp. 37–45.

[Ben18] Carsten Benthin.
Embree Ray Tracing Kernels 3.X: Overview and Features. 2018.

[Ben75] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching".
In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[Ble18] Blender Online Community.
Blender - a 3D modelling and rendering package. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018.
URL: `http://www.blender.org`.

[Fen17] Tom Fenton.
Running Graphical Programs on Windows Subsystem on Linux. 2017. URL: `https://virtualizationreview.com/articles/2017/02/08/graphical-programs-on-windows-subsystem-on-linux.aspx` (visited on 06/13/2021).

[Fis15] Jerry Fisher. *David by Michelangelo.* 2015.
URL: `https://sketchfab.com/3d-models/david-by-michelangelo-8f4827cf36964a17b90bad11f48298ac` (visited on 07/12/2021).

[Gla89] Andrew S Glassner. *An introduction to ray tracing.*
Morgan Kaufmann, 1989.

[Gor+84] Cindy M Goral et al.
"Modeling the interaction of light between diffuse surfaces".
In: *ACM SIGGRAPH computer graphics* 18.3 (1984), pp. 213–222.

[HH11] Michal Hapala and Vlastimil Havran. "Kd-tree traversal algorithms for ray tracing". In: *Computer Graphics Forum.* Vol. 30. 1. Wiley Online Library. 2011, pp. 199–213.

[Hound] Ben Houston. *Utah Teapot.* n.d. URL: `https://clara.io/view/8d9a8181-f1ce-4340-b24f-e36bbaf318f7#` (visited on 07/12/2021).

[Hug+13] John F. Hughes et al. *Computer Graphics: Principles and Practice.* 3rd ed. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN: 978-0-321-39952-6.

[Int21] Intel. *Intel Embree High Performance Ray Tracing Kernels - User Documentation.* 2021. URL: `https://github.com/embree/embree/blob/master/readme.pdf` (visited on 05/30/2021).

[Int21] Intel Newsroom. *Intel Embree Wins Academy Scientific and Technical Award.* 2021. URL: `https://newsroom.intel.de/news/intel-embree-wins-academy-scientific-and-technical-award/#gs.2q50qi` (visited on 05/30/2021).

[Kaj86] James T Kajiya. "The rendering equation". In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques.* 1986, pp. 143–150.

[Kar16] Markéta Karaffová. "Efficient Ray Tracing of CSG Models". MA thesis. Czech Technical University in Prague, 2016.

[KS16] Lars Kiesow and Nigel Stewart. *RPly.* `https://github.com/lkiesow/librply`. 2016.

[Kř19] Jaroslav Křivánek. *Computer Graphics III - Physically based rendering (NPGR010).* 2019. URL: `https://cgg.mff.cuni.cz/~jaroslav/teaching/2019-npgr010/index.html` (visited on 06/24/2021).

[Mei+21] Daniel Meister et al. "A Survey on Bounding Volume Hierarchies for Ray Tracing". In: *Computer Graphics Forum.* Vol. 40. 2. Wiley Online Library. 2021, pp. 683–712.

[MFW18] Michal Mojzík, Alban Fichet, and Alexander Wilkie. "Handling Fluorescence in a Uni-directional Spectral Path Tracer". In: *Computer Graphics Forum*. Vol. 37. 4. Wiley Online Library. 2018, pp. 77–94.

[MT97] Tomas Möller and Ben Trumbore. "Fast, minimum storage ray-triangle intersection". In: *Journal of graphics tools* 2.1 (1997), pp. 21–28.

[ND+19] Merlin Nimier-David et al. "Mitsuba 2: A retargetable forward and inverse renderer". In: *ACM Transactions on Graphics (TOG)* 38.6 (2019), pp. 1–17.

[NVI18] NVIDIA. *The Compleat Angler – First demonstration of recursive ray tracing in computer graphics animation YouTube video.* 2018. URL: `https://www.youtube.com/watch?v=0KrCh5qD9Ho` (visited on 05/30/2021).

[ON94] Michael Oren and Shree K Nayar. "Generalization of Lambert's reflectance model". In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques.* 1994, pp. 239–246.

[Ped19] Jon Peddie. *Ray Tracing: A tool for all.* Vol. 5. Springer, 2019.

[Pho75] Bui Tuong Phong. "Illumination for computer generated pictures". In: *Communications of the ACM* 18.6 (1975), pp. 311–317.

[PJH16] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation.* Morgan Kaufmann, 2016.

[Ply] *The Stanford 3D Scanning Repository.* 2014. URL: `http://graphics.stanford.edu/data/3Dscanrep/` (visited on 07/08/2021).

[Rot82] Scott D Roth. "Ray casting for modeling solids". In: *Computer graphics and image processing* 18.2 (1982), pp. 109–144.

[TS67] Kenneth E Torrance and Ephraim M Sparrow. "Theory for off-specular reflection from roughened surfaces". In: *Josa* 57.9 (1967), pp. 1105–1114.

[Wal+14] Ingo Wald et al. "Embree: a kernel framework for efficient CPU ray tracing". In: *ACM Transactions on Graphics (TOG)* 33.4 (2014), pp. 1–8.

[WH06] Ingo Wald and Vlastimil Havran. “On building fast kd-trees for ray tracing, and on doing that in O (N log N)”. In: *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE. 2006, pp. 61–69.

[WH13] Alexander Wilkie and Lukas Hošek. “Predicting sky dome appearance on earth-like extrasolar worlds”. In: *Proceedings of the 29th Spring Conference on Computer Graphics*. 2013, pp. 145–152.

[Whi79] Turner Whitted. “An improved illumination model for shaded display”. In: *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*. 1979, p. 14.

[Wil+09] Alexander Wilkie et al. “Predictive rendering”. In: *ACM SIGGRAPH ASIA 2009 Courses*. 2009, pp. 1–428.

[Wil+14] Alexander Wilkie et al. “Hero wavelength spectral sampling”. In: *Computer Graphics Forum*. Vol. 33. 4. Wiley Online Library. 2014, pp. 123–131.

[Wil+nda] Alexander Wilkie et al. *The ARM File Reference Manual*. n.d. URL: `https://cgg.mff.cuni.cz/ART/assets/ARM_Interface.pdf` (visited on 05/30/2021).

[Wil+ndb] Alexander Wilkie et al. *The ART Handbook*. n.d. URL: `https://cgg.mff.cuni.cz/ART/assets/ART_Handbook.pdf` (visited on 05/30/2021).

[WW11] Alexander Wilkie and Andrea Weidlich. “A physically plausible model for light emission from glowing solid objects”. In: *Computer Graphics Forum*. Vol. 30. 4. Wiley Online Library. 2011, pp. 1269–1276.

[Zaj12] Petr Zajiček. “Acceleration of Ray-Casting for CSG scenes”. MA thesis. Charles University in Prague, 2012.

[Zot21] Zottie. *Illustration of CSG tree. Created and rendered in POV-Ray.* 2021. URL: `https://commons.wikimedia.org/wiki/File:Csg_tree.png` (visited on 06/22/2021).

List of Figures

List of Tables

Listings

Appendix A

Electronic attachments

Attached to this thesis in electronic form, one can find the ART repository together with additional Objective-C source files that implement the functionality described in this thesis.

The following directory tree shows the files of the attachment that are the most interesting to novel users:

```
/
└─ ART (Repository folder)
    ├─ [...]
    ├─ Gallery (Folder containing individual .tex files )
    ├─ Documentation (Folder containing several LaTeX source
    │  document files that compose the documentation)
    └─ [...]
```

The ART Handbook can be compiled by executing the `gen_arm_doc.py` python script which can be found in the `Documents` folder, or alternatively, be squired together with the ARM Scene File Reference Manual from the official homepage of ART which is https://cgg.mff.cuni.cz/ART/download/.

Appendix B

User guide for software installation

In this appendix, a user guide for both compiling and running ART with Embree support is provided.

B.1 Installing Embree

The first thing one has to consider is the installation of Embree. Embree can be obtained via its homepage *https://www.embree.org/*, which provides a link to the dedicated Github repository. It can either be built from source or installed from pre-compiled binaries. A detailed description of the installation procedure for Windows (32-bit and 64-bit), Linux (64-bit), and macOS (64-bit) can be found in the `README.md` file in the repository or the included user documentation [Int21].

B.2 Compiling ART with Embree support

After a successful installation of Embree, ART is ready to be used. A limitation of ART is the fact that it does only support macOS and Linux systems. However, by installing a Linux subsystem, making ART work on Windows 10 is possible. The next chapter is dedicated to this. The installation procedure of ART with Embree support is almost identical to the one described in the ART handbook [Wil+ndb]. Since, at the time of writing this thesis, the source code of ART is hosted on a private GitLab server, a zipped folder of the ART source code is provided as an attachment to this thesis. It furthermore contains the ART Handbook as a PDF file. For building ART on Windows 10, jump to the next section. Otherwise, execute the following steps:

1. Extract the provided zipped folder of the ART source code to a convenient location on your system.

2. Open the PDF file containing the ART handbook and follow the instructions of Chapter 1. Ignore the section *1.2 Getting the Source*, since the provided GitHub URL is obsolete and the source code in question is already included in the extracted folder.

3. Follow the instructions of Chapter 2 until section *2.2.1 Run CMake*. Here lies the crucial difference: To directly quote from this section:

   > "The only decisions you have to make at this point is whether you want to install the finished ART binaries globally for all users [...] . To make a choice, go to the source directory and type either
   >
   > `$ cmake .`
   >
   > or
   >
   > `$ cmake . -DCMAKE_INSTALL_PREFIX=~`

   The decision to build ART with Embree support is left to the user. Therefore, to build ART with Embree support, one has to additionally set a Boolean CMake variable `ENABLE_EMBREE_SUPPORT`, which is set to `OFF` by default. This can be done by typing

   ```
   $ cmake . -DENABLE_EMBREE_SUPPORT=1
   ```

   or

   ```
   $ cmake . -DCMAKE_INSTALL_PREFIX=~ -DENABLE_EMBREE_SUPPORT=1
   ```

   Alternatively, *CCMake* can be used. CCMake offers a GUI-like interface in which different CMake variables can be edited. All one has to do is to install CCMake, e.g., on Ubuntu by the commands

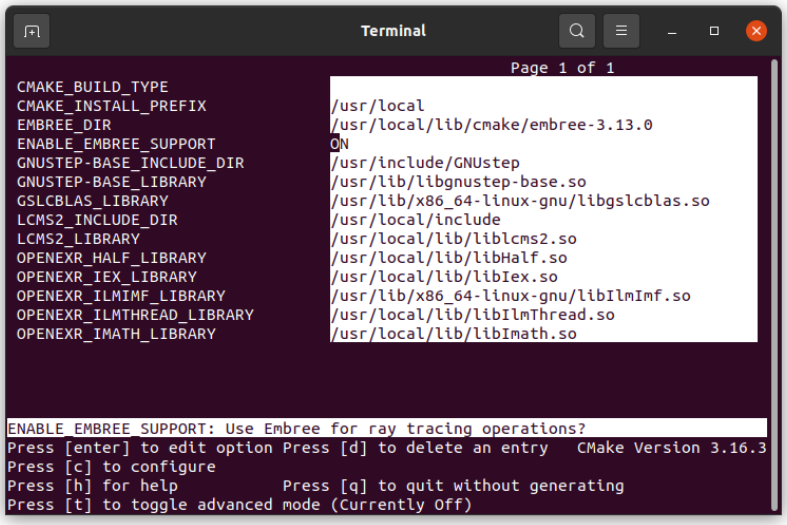
   ```
   $ sudo apt update
   $ sudo apt-get install cmake-curses-gui
   ```

   and then, while being in the source directory, type

   ```
   $ ccmake .
   ```

   When typing this for the first time, a menu-like screen should appear with the message `EMPTY CACHE`. Configure the project by pressing `c` on the ke. Some helpful feedback messages are printed, exit this screen by typing `e`.

Now a table of CMake variables and their values like in Figure B.1 should appear. One can navigate through the table with the arrow keys. Locate



```
                                                 Page 1 of 1
CMAKE_BUILD_TYPE
CMAKE_INSTALL_PREFIX            /usr/local
EMBREE_DIR                      /usr/local/lib/cmake/embree-3.13.0
ENABLE_EMBREE_SUPPORT           ON
GNUSTEP-BASE_INCLUDE_DIR        /usr/include/GNUstep
GNUSTEP-BASE_LIBRARY            /usr/lib/libgnustep-base.so
GSLCBLAS_LIBRARY                /usr/lib/x86_64-linux-gnu/libgslcblas.so
LCMS2_INCLUDE_DIR               /usr/local/include
LCMS2_LIBRARY                   /usr/local/lib/liblcms2.so
OPENEXR_HALF_LIBRARY            /usr/local/lib/libHalf.so
OPENEXR_IEX_LIBRARY             /usr/local/lib/libIex.so
OPENEXR_ILMIMF_LIBRARY          /usr/lib/x86_64-linux-gnu/libIlmImf.so
OPENEXR_ILMTHREAD_LIBRARY       /usr/local/lib/libIlmThread.so
OPENEXR_IMATH_LIBRARY           /usr/local/lib/libImath.so



ENABLE_EMBREE_SUPPORT: Use Embree for ray tracing operations?
Press [enter] to edit option Press [d] to delete an entry   CMake Version 3.16.3
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Figure B.1 Setting the ENABLE_EMBREE_SUPPORT in CCmake.

the variable ENABLE_EMBREE_SUPPORT and press ENTER in order to set this variable from OFF to ON. Press c once again to re-configure the project and exit the "feedback screen" with e. Now, in the list of options at the bottom, an option `Press [g] to generate and exit` should be visible. Press g to generate the project and exit CCMake.

4. Continue following the steps in Chapter 2.

5. The functionality and usage of ART is the central topic of Chapter 3. Arm scene files are rendered by invoking the *artist* command line tool. For an overview of the features and command line options, type

   ```
   $ artist -h
   ```

   A scene file, say foo.arm, with native ART can be rendered by typing

   ```
   $ artist foo.arm
   ```

   If Embree support is desired, artist has to be executed with an additional -e flag:

   ```
   $ artist foo.arm -e
   ```

B.3 Installation of ART on Windows 10

As briefly mentioned earlier, ART naturally does not support Windows platforms. However, with the help of a so-called Linux Subsystem for Windows, running ART on Windows 10 is possible. This section guides the process of setting up such a subsystem and additional tools for displaying rendered images. In this section, the subsystem of choice is "Ubuntu 20.04 LTS".

1. Begin by opening the Microsoft Store and type "Ubuntu" in the search bar on the top right. This should result in a display of multiple apps. Select the "Ubuntu 20.04 LTS" app like shown in Figure B.2 and install it.

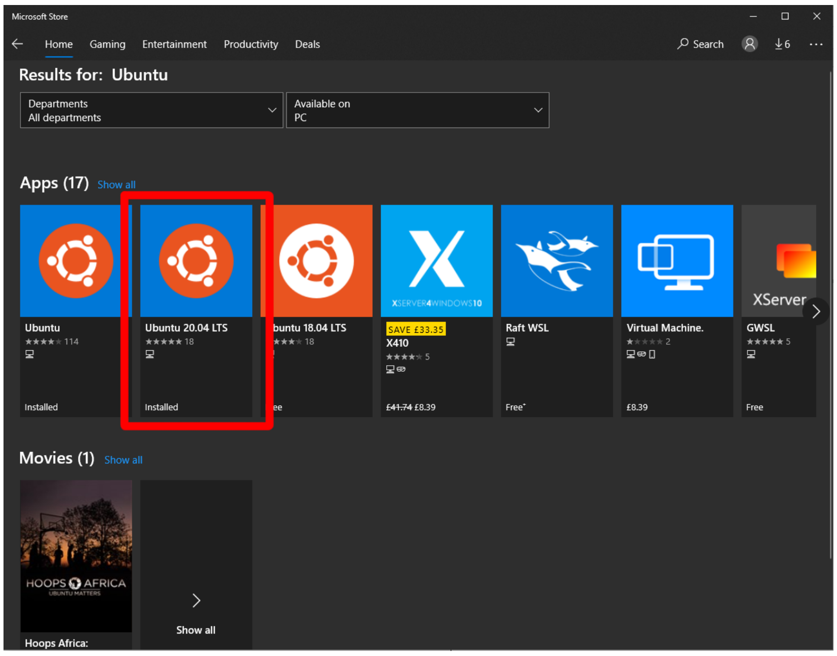


Figure B.2 The Ubuntu subsystem we are using in the Microsoft Store.

2. After the installation, click the `Launch` button in the Microsoft Store. A terminal will open. Wait a few minutes until the setup is complete, and provide a user name and password. After that, the subsystem is up and ready.

3. While being in the *home/<username>* directory, type

```
$ explorer.exe .
```

to open the Windows file explorer. Extract the zipped folder containing the ART source code here, or create a suitable location for it in the directory and extract it there.

4. The Linux subsystem does not natively support graphical user interface applications, although these are crucial for viewing rendered images. However, displaying GUI applications, in general, can be done by running an X server on Windows 10, which communicates to the Linux subsystem. The following procedure is taken from the guide *Running Graphical Programs on Windows Subsystem on Linux* [Fen17]. There is a variety of such X servers. Like the author of the guide, we decided to use *Xming*. It can be downloaded via *https://sourceforge.net/projects/xming/*. Download the executable and follow the installation wizzard, leave the settings as default. Start the application. An `XLaunch` should be visible in the Windows toolbar.

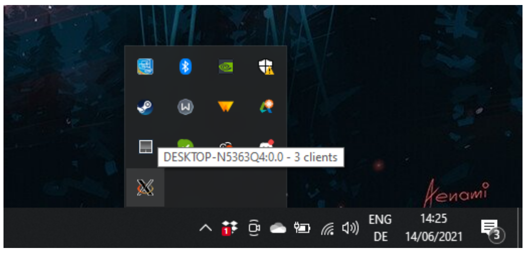


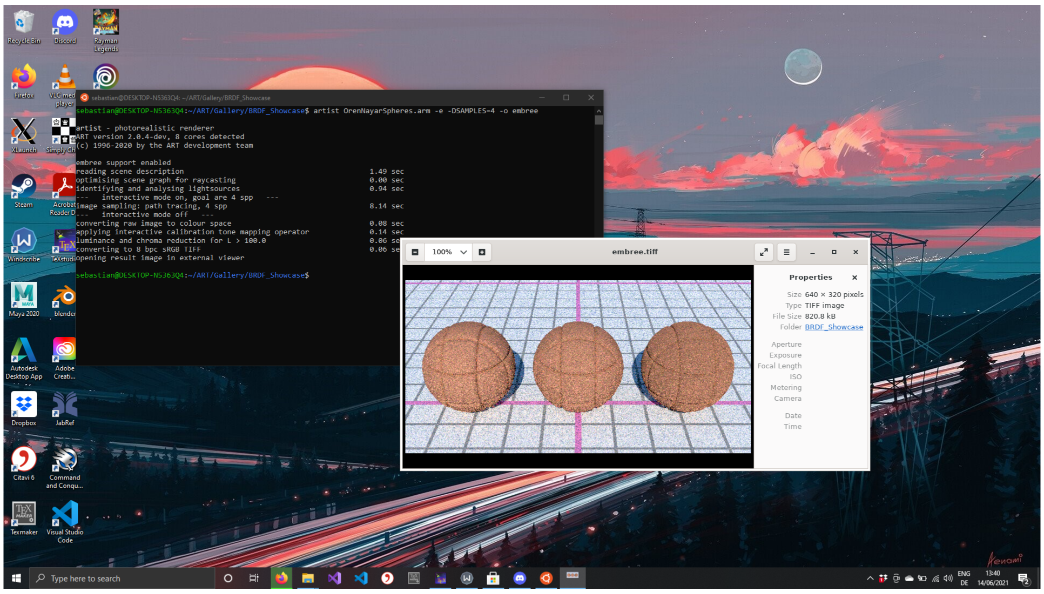
Figure B.3 XLaunch running in the background.

In the Ubuntu terminal, type

```
$ export DISPLAY=:0
```

This will enable the display of graphical user interfaces from the Linux subsystem in Windows.

5. Follow the instructions in Section B.1 and B.2 to setup Embree and ART.



```
sebastian@DESKTOP-N5363Q4:~/ART/Gallery/BRDF_Showcase$ artist OrenNayarSpheres.arm -e -DSAMPLES=4 -o embree

artist - photorealistic renderer
ART version 2.0.4-dev, 8 cores detected
(c) 1996-2020 by the ART development team

embree support enabled
reading scene description                                   1.49 sec
optimising scene graph for raycasting                       0.00 sec
identifying and analysing lightsources                      0.94 sec
---   interactive mode on, goal are 4 spp   ---
image sampling: path tracing, 4 spp                         8.14 sec
---   interactive mode off   ---
converting raw image to colour space                        0.08 sec
applying interactive calibration tone mapping operator      0.14 sec
luminance and chroma reduction for L > 100.0                0.06 se
converting to 8 bpc sRGB TIFF                               0.06 se
opening result image in external viewer

sebastian@DESKTOP-N5363Q4:~/ART/Gallery/BRDF_Showcase$
```

Size	640 × 320 pixels
Type	TIFF image
File Size	820.8 kB
Folder	BRDF_Showcase
Aperture	
Exposure	
Focal Length	
ISO	
Metering	
Camera	
Date	
Time	

Figure B.4 Rendering with ART on Windows 10.