FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Fast CPU Ray-Triangle Intersection Method

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik

eingereicht von

Thomas Alois Pichler, BSc
Matrikelnummer 01126560

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.nat. Dr.techn. Hannes Kaufmann
Mitwirkung: Projektass. Mag. Dr.techn. Peter Kán

Wien, 31. Juli 2018

Thomas Alois Pichler	Hannes Kaufmann

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Fast CPU Ray-Triangle Intersection Method

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Media Informatics

by

Thomas Alois Pichler, BSc
Registration Number 01126560

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.nat. Dr.techn. Hannes Kaufmann
Assistance: Projektass. Mag. Dr.techn. Peter Kán

Vienna, 31st July, 2018

Thomas Alois Pichler	Hannes Kaufmann

Erklärung zur Verfassung der Arbeit

Thomas Alois Pichler, BSc
Panoramastraße 13, 2871 Zöbern

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 31. Juli 2018

Thomas Alois Pichler

Danksagung

Besonders danken möchte ich meiner Familie und meinem Freundeskreis, die es mir nachgesehen haben, dass ich mich während der letzten Monate quasi eingesperrt habe, um diese Diplomarbeit zu einem Ende zu bringen. Mein Dank gilt auch Peter Kán, meinem großartigen Betreuer. Zu guter Letzt danke ich noch allen, die mich irgendwie, irgendwann während der letzten Monate inspiriert oder ermutigt haben.

Acknowledgements

I want to thank my family and friends who pardoned the fact that I more or less locked myself up these past few months to finish this thesis. Thanks to Peter Kán for being a really awesome supervisor. Finally, thanks to everyone who somehow, at some time, inspired or encouraged me.

Kurzfassung

Raytracing ist eine Technik, mit der hochgradig realistische Bilder erzeugt werden können. Dieser Realismus wird durch das Simulieren der physikalisch korrekten Ausbreitung von Lichtstrahlen erzielt. Anwendungsgebiete für Raytracing sind beispielsweise Filmproduktion, Visualisierung, die Automobilindustrie, Kunst und Spieleentwicklung. “Rays” stellen in diesem Kontext Blickrichtungen und Lichtstrahlen dar, deren Verlauf in einer Szene verfolgt wird. Rays interagieren mit Objekten in dieser Szene (sie durchdringen sie, werden reflektiert, gebrochen etc.) und tragen im Endeffekt zu den jeweiligen Pixelwerten des generierten Bildes bei. Eine der wichtigsten Operationen in einem Raytracer ist die Berechnung von Schnittpunkten der Rays mit geometrischen Primitiven. Eines der meistgenutzten Primitiven ist das Dreieck. Schnittpunktberechnungen zwischen Rays und Dreiecken sind verhältnismäßig einfach, und komplexe Objekte werden oft durch Dreiecksnetze approximiert. Aus diesem Grund werden Schnittpunktberechnungen zwischen Rays und Dreiecken relativ häufig ausgeführt. Raytracing wird generell als eine eher langsame Methode der Bildgenerierung angesehen, weswegen in den letzten Jahren mehrere Algorithmen zur schnellen Ray-Dreieck-Schnittpunktberechnung vorgestellt wurden.

Im Rahmen dieser Diplomarbeit wird ein neuer, schneller Algorithmus zur Ray-Dreieck-Schnittpunktberechnung in das CPU-basierte Raytracing-Framework PBRT implementiert. Er basiert auf Early-Termination-Strategien und transformiert Ray-Ebene-Schnittpunkte in ein 2D-Koordinatensystem. Für diese Transformation werden zwei verschiedene Ansätze beschrieben. Verschiedene Optimierungsmaßnahmen für den Algorithmus werden erforscht. Der Algorithmus wird hinsichtlich seiner Effizienz mit dem in PBRT implementierten Default-Algorithmus sowie mit einem anderen State-of-the-art-Algorithmus verglichen. Für die Tests werden realistische Szenen mit unterschiedlicher Komplexität und verschieden hohen Ray-Dreieck-Trefferraten verwendet. Die Ergebnisse zeigen, dass der neue Algorithmus hinsichtlich seiner Effizient den Default-Algorithmus bei jeder Szene übertrifft.

Abstract

Ray tracing is a technique capable of synthesizing highly realistic images. It accurately simulates the physical distribution of light in a scene and can be used for movie production, scientific visualization, in the automotive industry, or by artists or game developers. In this context, rays of light, as well as lines of sight, are represented by "rays" which are traced throughout the scene. Rays interact with objects in the scene (they penetrate them, are reflected, refracted, etc.) and ultimately define the pixel values of the generated image. Thus, one crucial functionality of a ray tracing application is calculating the intersection of rays with scene primitives. One of the most often used primitives is the triangle. It enables relatively simple intersection calculations, and complex objects can be tesselated into triangles. Thus, a ray tracer's ray-triangle intersection routine is called a considerable number of times per scene. Ray tracing is generally seen as a rather slow method of image generation. Thus, several algorithms for fast ray-triangle intersection have emerged within the last years.

In the scope of this thesis, a novel, fast ray-triangle intersection algorithm is implemented into the CPU-based ray tracing framework PBRT. The algorithm features early termination strategies and transforms the ray-plane intersection point into a 2D coordinate system. For this transformation, two different approaches are discussed. Different optimizations are explored to further improve the algorithm's performance. The algorithm is evaluated against PBRT's default algorithm and against another state-of-the-art ray-triangle intersection algorithm in terms of efficiency. Realistic scenes with different ray-triangle hit-rates and different scene complexity are used for the tests. The results show that the new algorithm outperforms the default algorithm for every scene.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Ray tracing is a technique for generating high-qualitative, highly realistic images. It is capable of realistically simulating the physical process of light transportation and, therefore, enables synthesizing effects like shadows, refraction and reflection in a sophisticated fashion [GRHS08]. Furthermore, ray tracing displays logarithmic complexity regarding scene size, making it feasible for large scenes [DWWS04].

Ray tracing is of use in various fields. It can provide highly realistic effects for movie production [Wal04] or be used by artists. Dietrich et al. (2004) emphasize the benefit of ray tracing in the automotive industry. The generated, realistic images can help avoiding the need of producing physical prototypes, which are more expensive [DWWS04] or would have to be built in great numbers to evaluate every possible product configuration [GRHS08]. Furthermore, as ray-traced images realistically depict physical lighting processes, they provide a higher level of trust to the users when evaluating cricital design decisions [DWWS04]. For example, one can dependably evaluate lighting-related design issues for cars (visibility of lights, reflections), or the readability and ergonomic quality of displays and instruments in critical environments like cockpits [GRHS08].

With the emergence of hardware which enables real-time ray tracing, the field also became attractive for game developers [Bik07][FGD+06] and may be used for interactive visualization [BSP06]. The Arauna game engine, developed at the NHTV Breda University of Applied Science, uses ray tracing for in-game features like grenade bouncing, collision detection and determining a line of sight as well [Bik07]. Parker et al. (2010) note that their OptiX engine can also be used for simulating sound propagation [PRS+10].

1.2 Problem statement

This section explains the basic principles of ray tracing. A description of the ray-object intersection problem is given, revealing the need of fast ray-triangle intersection algorithms.

1.2.1 Ray tracing fundamentals

The idea behind ray tracing is following the path of a ray of light through a scene. The scene contains geometric objects the rays are interacting with. The rays may (partially) pass, be reflected, refracted, absorbed, etc. when hitting objects. This simulates realistic, physical distribution of light throughout the scene. Therefore, objects must exhibit features describing the scattering of light on their surface, like materials, surface normals, and so on. Consider an object with shiny material like metal. Its surface has to simulate reflecting light. This is done by creating rays on the object's surface and, again, tracing them throughout the scene [PJH16a].

The scene is being viewed from a viewing point, often called camera [PJH16a][GS08][FF15] [WPS+03][Wal04][BSP06][Ben06]. The amount of light reaching the viewer eventually defines the pixels of the rendered image [PJH16a]. A straightforward, but infeasible approach would be to trace rays originating from the light sources until they reach the viewer. This would lead to a vast majority of the rays not contributing to the final image, thus wasting computational effort. The more sophisticated approach is to trace rays in the opposite direction [Whi80]. Cameras generate rays as well, and these are traced throughout the scene, delivering the light information needed for rendering the image. For example, the case of a ray generated from a point on an object's surface which does not reach a light source provides shadow information (these rays are called shadow rays in the literature) [PJH16a]. Typically, thousands of rays are shot for each pixel to calculate the final image. Using too few rays results in visible noise in the image.

1.2.2 The intersection problem

Light can only reach a certain point in space when there is an unobstructed path between the point and the light source. Similarly, the viewer can only see a point if it is not occluded by obstacles. A ray tracer must compute whether a ray (generated by the camera) hits an object, i.e., whether it intersects some kind of geometric surface. Typically, the first intersection along the ray is reported, though there may be desired exceptions of this rule as well [PJH16a]. The intersected object gives further information about the light scattering on its surface. Its properties contribute to pixel's color values, define how rays bounce off, etc.

A brute-force approach to find the objects intersected by a ray would be to test each object in the scene for intersection. This approach is very slow even for scenes with low complexity [PJH16a]. Ray tracers thus employ acceleration structures to quickly discard geometric primitives which are not hit by the ray.

Finding the nearest intersection of rays with objects is one of the core operations of any ray tracing application [DEG+12][Whi80][RW80] and consumes a substantial part of a ray tracer's computing time [Wal04][DK06]. Depending on the type of geometric primitive, different intersection approaches are needed [KS06][PJH16d].

1.2.3 Ray-triangle intersection

Triangles are considered as the standard primitives in Computer Graphics. They are often seen as a common base of modelers and renderers [KS06]. Complex objects can be tessellated into triangle meshes. This may increase the number of objects to be intersected, but can significantly speed up computation [AC97][SB87]. The simplicity of triangles enables fast operations even for high-performance applications [JONP14]. Subsequently, ray-triangle intersections are computed numerous times in a ray tracing application. Thus, using a fast, efficient ray-triangle intersection algorithm is of high importance [HH10][Ben06][BW16][KS06]. Application of ray-triangle intersection is not limited to ray tracing, but can also be used for inclusion tests, boolean operations, object modeling, physics simulation, and collision detection [JONP14][SF01].

The desired result of a ray-triangle intersection test is either just a boolean value (true, if the ray intersects the triangle, and false, if it does not), or the location of the intersection point [JONP14]. Over time, many different algorithms for ray-triangle intersection have emerged (see also Chapter 3). As there have been approaches to achieve real-time ray tracing in recent years, novel algorithms enabling even faster ray-triangle intersection are still being researched [Wal04][Ben06][GRHS08]. However, ray tracing in general is seen as a rather time-inefficient way of image synthesis. Georgiev and Slusallek (2008) note that ray tracing is "famous for its high computational requirements" [GS08]. Bikker (2007) states that a ray tracer can be written in 100 lines of code, however, depending on its implementation, it may be very slow [Bik07]. According to Jiménez et al. (2010), a ray-triangle intersection algorithm's performance depends on many factors, with some of them being difficult to predict [JSF10].

1.3 Aim of the work

In the scope of this thesis, a novel ray-triangle intersection algorithm proposed by Ferko and Ferko (2015) [FF15] is implemented into the physically based ray tracing framework PBRT, version 3, designed by Pharr et al. [PJH16d]. Furthermore, different optimizations are explored to speed up the intersection process. The algorithm is tested against the standard PBRT ray-triangle intersection algorithm and the algorithm proposed by Baldwin and Weber (2016) [BW16] in terms of efficiency (see Chapter 3 for a detailed description of these algorithms). Tests include evaluating the difference in quality of the rendered image after a given rendering time and the consumed time for achieving results of comparable quality.

Löfstedt and Akenine-Möller (2005) state that there is no single, most efficient ray-triangle

intersection algorithm as the efficiency depends on a variety of factors [LAM05]. Differences in the scene configurations can impact the results. For example, some algorithms perform better when the probability of the ray missing the triangle is high [Ben06][JSF10]. Thus, the efficiency of the implemented algorithm regarding different scene configurations is evaluated as well.

1.4 Methodological approach

The methodology of the thesis consisted of the following steps:

1. Literature review: We deepen our knowledge concerning the topics of ray tracing, ray-triangle intersection, and details of different intersection approaches.

2. Implementation: We describe the intersection algorithm designed by Ferko and Ferko [FF15] and implement it into the PBRT framework. We optimize the algorithm to produce faster results using the understandings gained from the literature review.

3. Evaluation: We test the algorithm on different scenes and compare it to the default algorithm implemented in PBRT-V3 [PJH16c] and to the Baldwin-Weber algorithm [BW16] in terms of perfomance. We analyze how scene configuration can affect the results.

1.5 Contribution to the field of research

There exists an unpublished paper by Ferko and Ferko (2015) [FF15] proposing a ray-triangle intersection algorithm utilizing early termination strategies. Two different versions of the algorithm are proposed, each suggesting a different transformation into two-dimensional (2D) space and subsequent point-in-triangle tests. These are a unit triangle approach and a pruning approach. Ferko and Ferko further describe an optimization for the pruning approach using probability estimation. In addition to GPU tests, Ferko and Ferko implemented the unit triangle approach for the CPU into a testing framework. They ran tests with synthetic data and real data as well. They did not use any acceleration structure, but tested every ray against every triangle [FF15]. Baldwin and Weber (2016) proposed another ray-triangle intersection algorithm and implemented it into the CPU-based ray tracing framework PBRT, version 2 [BW16].

In the scope of this thesis, we implement Ferko and Ferko's algorithm into PBRT, version 3. This allows us to test in a realistic environment, including acceleration structures, and to conduct direct tests against PBRT's default ray-triangle intersection algorithm. We experiment on different optimizations of Ferko and Ferko's algorithm. We calculate a tighter fitting bounding sphere for every triangle to optimize early termination tests. We implement an additional bounding circle and a ray-circle intersection test as an

alternative early termination test. We experiment with the order of tests. In addition, we implement the pruning approach, which requires the following steps:

1. Constructing a transformation matrix that enables similarity preserved mapping of triangles into 2D space.
2. Implementing the pruning tests, i.e., the point-in-triangle tests for the pruning approach.
3. Calculating probabilities with which rays are expected to hit certain areas of the triangle's bounding circle.
4. Reordering the pruning tests according to the probabilities.

The pruning approach has not yet been implemented by Ferko and Ferko. Furthermore, we expand the two possible variants of the pruning approach. Ferko and Ferko discuss a 3-fold pruning method and a 4-fold pruning method [FF15]. We experiment with weighting of pruning tests and with a fast-forward version. The latter skips the bounding circle test while assuming the estimated probabilities to be sufficiently accurate. In sum, we test eight configurations of pruning tests:

- 3-fold pruning
- 3-fold pruning with probability estimation
- 3-fold pruning with probability estimation and weights
- 3-fold pruning with probability estimation and skipped bounding circle test
- 4-fold pruning
- 4-fold pruning with probability estimation
- 4-fold pruning with probability estimation and weights
- 4-fold pruning with probability estimation and skipped bounding circle test

We further explore possible speed-ups. Depending on the application, the pruning approach does not provide sufficient information for rendering on its own. Thus, we consider a combination of the unit triangle approach and the pruning approach. In this hybrid approach, we use the pruning approach when possible, and the unit triangle approach when necessary. Finally, we also adapt Baldwin and Weber's algorithm for PBRT-V3 to have another algorithm to test against.

1.6 Structure of the work

Chapter 2 describes recurring ray tracing concepts found in the literature. Chapter 3 focusses on the analysis of existing approaches. It describes ray-triangle intersection algorithms and analyzes further statements on the topic. Finally, it gives a summary of the different approaches, noting their simliarities and differences.

Chapter 4 describes Ferko and Ferko's algorithm. It provides information concerning the concepts, methods, and language used in the implementation as well as the analysis methods. Chapter 5 gives the implementational details.

Chapter 6 presents the test results and provides a critical reflection of the algorithm. The algorithm is compared to the other approaches described in Chapter 3. Finally, Chapter 7 summarizes this thesis and makes suggestions for future work.

CHAPTER 2

Theoretical background

This chapter explains basic concepts used in ray tracing. This is done for the sake of better understanding in the following chapters. The described concepts include rays and triangles (Section 2.1), barycentric coordinates (Section 2.2), Plücker coordinates (Section 2.3), Cramer's rule (Section 2.4), the ray-plane intersection test (Section 2.5), and bounding volumes of a triangle (Section 2.6).

2.1 Rays and triangles

A three-dimensional (3D) ray is typically defined in the literature by a starting point and a direction. A ray R can be expressed by the parametric equation

$$R(t) = O + t\vec{D},$$

where O is the ray's origin in 3D, $\vec{D}$ is the ray's direction in 3D, and t is a parameter ranging from 0 to infinity. Another way is to view the ray as a line segment S between two points Q_1 and Q_2, represented by

$$S(t) = Q_1 + t(Q_2 - Q_1),$$

where t is in the range $[0, 1]$ [JONP14]. As Jiménez et al. (2014) describe the latter as a specific case of the former [JONP14] and most of the literature reviewed for this thesis uses the parametric form (see also Section 3.2), without loss of generality, the following intersection fundamentals will be explained on the basis of the parametric form.

A 3D triangle is typically described as a set of vertices and/or edges. Sometimes, it might be desired to also store the triangle normal [JONP14] and/or other information like triangle bounding volumes or transformation matrices which are used in the intersection procedure.

2.2 Barycentric coordinates

A helpful tool in ray tracing algorithms are barycentric coordinates. An object's barycenter is its "center of gravity" [O'R98]. For a triangle $T = ABC$, this means that there exist three real numbers u, v and w, and any point P relative to T's vertices can be expressed as

$$P = Au + Bv + Cw,$$

while

$$u + v + w = 1 \text{ [O'R98].}$$

In T's barycenter, u, v, and w are equal. For any point inside the triangle, u, v, and w are positive. This fact can be used to determine whether a point P lies inside the triangle or not. Bogart (1988) views barycentric coordinates as areas of subtriangles of a triangle with area 1. Thus, if the sum of all three barycentric coordinates of a possible hit point exceeds 1, the point lies not within the triangle [Bog88]. Barycentric coordinates are further used in ray tracing applications to interpolate vertex information like color, texture coordinates, normals, or tangent vectors [JONP14][AC97][SSK07].

2.3 Plücker coordinates

Plücker coordinates are a way of defining oriented lines via six-dimensional vectors. Let L be a line passing through two points $P_1 = (a, b, c, d)$ and $P_2 = (w, x, y, z)$ (P_1 and P_2 are both described with homogeneous coordinates). The Plücker coordinates of L are:

$$(L_1, L_2, L_3, L_4, L_5, L_6) = (ax - bw, ay - cw, by - cx, az - dw, bz - dx, cz - dy)$$

For Plücker coordinates, the following equation always holds true:

$$L_1L_6 - L_2L_5 + L_3L_4 = 0$$

Consider a line M represented by Plücker coordinates $(M_1, M_2, M_3, M_4, M_5, M_6)$. Now,

$$I = L_1M_6 - L_2M_5 + L_3M_4 + L_4M_3 - L_5M_2 + L_6M_1$$

describes how L interacts with M. If $I = 0$, then L intersects M. If $I < 0$, then L goes clockwise around M. If $I > 0$, then L goes counterclockwise around M.

For ray tracing, I can be calculated to determine the interaction of the ray with each of the triangle's edges. The ray only hits the triangle if it hits one of its edges (i.e., $I = 0$ for this edge), or if it either goes clockwise or counterclockwise around every edge (i.e., I has the same sign for all three edges) [Eri97].

2.4 Cramer's rule

Cramer's rule is a method for solving systems of linear equations. Consider a system

$$a_1x_1 + b_1x_2 + \ldots + n_1x_n = A_1$$

$$a_2x_1 + b_2x_2 + \ldots + n_2x_n = A_2$$

$$\vdots$$

$$a_nx_1 + b_nx_2 + \ldots + n_nx_n = A_n$$

which should be solved for x_1. The coefficients of $x_1 \ldots x_n$ correspond to a coefficient matrix M with the dimensions $n \times n$. The right side of the system forms an $n \times 1$ column vector A. Let M_i be a matrix obtained from M by replacing its ith column with the column vector A, then

$$x_i = \frac{det(M_i)}{det(M)}, i = 1, 2, \ldots n,$$

where $det(X)$ is the determinant of matrix X. Cramer's rule can only be applied if M is non-singular, i.e., invertible [Sho07][AR13].

2.5 Ray-plane intersection test

A plane can be defined via its normal vector $\vec{n}$ and a point P_0 lying in the plane. When subtracting P_0 from any point P in the plane, the resulting vector $(P - P_0)$ also lies in the plane, thus has to be perpendicular to $\vec{n}$. As the dot product of two perpendicular vectors equals 0, the vector equation of a plane is:

$$(P - P_0) \cdot \vec{n} = 0$$

The plane equation can also be written as $a(x - x_0) + b(y - y_0) + c(z - z_0) = 0$. This form is known as the point-normal form of the plane equation [AR13].

To define the plane a triangle ABC lies in, the normal vector $\vec{n}$ of this plane and an arbitrary point P in ABC has to be known. Conveniently, P can be either of the triangle's vertices. Calculating the cross product of two arbitrary triangle edges results in a vector perpendicular to these edges, i.e., the triangle's normal vector. Using, for example, the edges $\overrightarrow{AB}$ and $\overrightarrow{AC}$, $\vec{n}$ can be defined as

$$\vec{n} = \overrightarrow{AB} \times \overrightarrow{AC}.$$

A ray's parametric form is

$$R_o + \overrightarrow{R_d} \cdot t = p,$$

where R_o is the ray's origin and $\overrightarrow{R_d}$ is its direction. t is the parametric distance from the ray's origin to the point p along the ray. Furthermore, t is positive, which is one of the criteria intersection routines test against.

If the ray intersects the plane, they share exactly one point P_I, so the ray equation can be inserted into the plane equation:

$$(R_o + \overrightarrow{R_d} \cdot t - P_0) \cdot \vec{n} = 0$$

Notes that the the second $\cdot$ symbol in this equation indicates a dot product. Rearranging this equation enables us to solve it for t:

$$\overrightarrow{R_d} \cdot t \cdot \vec{n} + (R_o - P_0) \cdot \vec{n} = 0$$

$$t = \frac{-(R_o - P_0) \cdot \vec{n}}{\overrightarrow{R_d} \cdot \vec{n}}$$

$$t = \frac{(P_0 - R_o) \cdot \vec{n}}{\overrightarrow{R_d} \cdot \vec{n}}$$

The denominator $\overrightarrow{R_d} \cdot \vec{n}$ is 0 when the ray and the plane are parallel, i.e., no intersection point exists, or when the ray lies in the plane and an infinite number of intersection points exist. In both cases, a ray tracer can discard this triangle.

Now that t is calculated, the ray-plane intersection point is:

$$P_I = R_o + t \cdot \overrightarrow{R_d}$$

2.6 Bounding volumes of a triangle

Bounding volumes, like bounding boxes or bounding spheres, are primitive objects which are used to enclose other geometric objects. Their intersection with rays can be calculated very quickly [Wal04]. For ray-triangle intersection algorithms, bounding volumes of triangles are of importance. A triangle's bounding volume contains the whole triangle, thus, if a ray does not intersect the bounding volume, it also cannot intersect the triangle. This triangle can then be discarded without having to conduct further intersection tests. A bounding volume is ideally as small as possible [PRS+10].

CHAPTER 3

Analysis of existing approaches

This chapter describes ray-triangle intersection approaches and general, relevant remarks on ray tracing found in the literature. In Section 3.1, approaches for ray-triangle intersection, evaluation results, and general ray-tracing statements are discussed. Section 3.2 summarizes and directly compares the algorithms described in Section 3.1.

3.1 Literature studies and analysis of ray-triangle intersection algorithms

As triangles are planar, the first step in an interesction routine can be to check whether the ray intersects the plane in which the triangle lies. This is done, for example, by Badouel's method (1990) [Bad90]. The case of a ray-plane intersection point P being inside the triangle $T = ABC$ can be expressed as

$$\overrightarrow{AP} = \alpha\overrightarrow{AB} + \beta\overrightarrow{AC},$$

where α, β, and $1 - \alpha - \beta$ are the barycentric coordinates of P. This gives the following system of equations in 3D:

$$P_\mathrm{x} - A_\mathrm{x} = \alpha(B_\mathrm{x} - A_\mathrm{x}) + \beta(C_\mathrm{x} - A_\mathrm{x})$$

$$P_\mathrm{y} - A_\mathrm{y} = \alpha(B_\mathrm{y} - A_\mathrm{y}) + \beta(C_\mathrm{y} - A_\mathrm{y})$$

$$P_\mathrm{z} - A_\mathrm{z} = \alpha(B_\mathrm{z} - A_\mathrm{z}) + \beta(C_\mathrm{z} - A_\mathrm{z}),$$

where A_x is the x-value of vertex A, A_y is its y-value, and so on. P is inside the triangle if $\alpha \geq 0$, $\beta \geq 0$, and $\alpha + \beta \leq 1$.

To simplify this system of equations, Badouel transforms the intersection problem into 2D by projecting the plane intersection point onto one of the coordinate planes.

The plane used is the one perpendicular to the triangle's normal vector's dominant component [Bad90]. Badouel's algorithm calculates the parametric distance t of the intersection point along the ray at the beginning, which is seen as a disadvantage by Jiménez et al. (2010). They note that calculations should only be done if truly necessary [JSF10].

Möller and Trumbore (1997) describe a ray-triangle intersection routine that uses barycentric coordinates as well, but does not store precomputed plane equations, thus reducing memory cost. Their algorithm is based on Badouel's. For a given ray and triangle, they transform the triangle into the triangle-specific barycentric coordinate system and then test for intersection, i.e., the intersection problem is transformed in such a way that

- the triangle becomes a unit triangle on the yz-plane,
- one triangle vertex lies in the origin, and
- the ray is aligned with the x-axis.

Again, this transformation eases solving the system of equations. Furthermore, cases in which there is no intersection can be identified early [MT97]. The algorithm is used by Pharr and Humphreys (2010) in their physically based ray tracing framework PBRT, version 2 [PH10a]. It is considered as the standard ray-triangle intersection routine for ray tracing by Baldwin and Weber (2016) [BW16] and as the fastest intersection algorithm for generic cases by Jiménez et al. (2014) [JONP14].

However, Baldwin and Weber claim to produce faster results "under ideal experimental conditions" [BW16]. They store a transformation matrix within each triangle that would transform the triangle to a 2D unit triange with vertices on $(0,0),(1,0),(0,1)$. During run-time, only the ray is transformed by this transformation. Afterwards, Baldwin and Weber intersect the transformed ray with the unit triangle. They compute the t value, i.e., the distance from the ray's origin to the intersection point. Then, they calculate the barycentric coordinates of the intersection point and check whether the point lies inside the triangle. Baldwin and Weber's approach minimizes the required storage as well as the number of operations for ray transformation by constructing the transformation matrix in such a way that one of its columns is a "free vector", consisting only of zeroes and a single one [BW16] (see also Section 5.5). Baldwin and Weber did some stand-alone tests and integrated their algorithm into PBRT, version 2, comparing it against the Möller-Trumbore algorithm (which is, as stated above, the standard ray-triangle intersection algorithm implemented in PBRT-V2 [PH10b]). They found their own algorithm to be faster, except at high triangle-hit-rates [BW16].

Schmittler et al. (2004) also transform the ray into a 2D coordinate system where the triangle vertices are mapped to $(1,0)$, $(0,1)$, and $(0,0)$, respectively. This simplifies calculation of the intersection parameter t and the barycentric coordinates of the intersection point [SWW+04]. This approach is adopted by Woop et al. (2004) [WSS05].

Benthin (2006) analyzes the possibility of real-time ray tracing on CPUs. He discusses coding guidelines for the implementation of ray tracing algorithms as well as performance issues. Benthin presents a fast ray tracing algorithm which uses a modified Plücker test instead of barycentric coordinates for intersection [Ben06].

Dammertz and Keller (2006) use a different approach altogether. They test rays against the axis-aligned bounding boxes of scene objects. These are recursively subdivided when intersected by the ray, refining an intersection interval in each step. This way, intersections are not determined along the ray, but in object space [DK06].

Woop et al. (2013) address the problem of ray-triangle tests not being watertight at edges and/or vertices. Their algorithm is watertight at edges and vertices and robust for all triangle configurations. This robustness is achieved by transforming the intersection problem in such a way that the ray starts at the origin and is directed along the z-axis, thus reducing it to a 2D problem. At run-time, this transformation is applied to the triangle vertices. For any intersection point, the x- and y-components must be 0. Thus, the point $(0, 0)$ is checked to determine whether it lies within the triangle. Woop et al. use Benthin's test [Ben06] to compute scaled barycentric coordinates. These correspond to twice the signed areas formed by the point with each of the triangle's edges [PJH16c]. The sign of the result of

$$(B_\mathrm{x} - A_\mathrm{x})(P_\mathrm{y} - A_\mathrm{y}) - (P_\mathrm{x} - A_\mathrm{x})(B_\mathrm{y} - A_\mathrm{y})$$

determines on which side of the edge $\overrightarrow{AB}$ the 2D point P lies. P being (0,0) simplifies the calculations. If each of the scaled barycentric coordinates has the same sign, the ray intersects the triangle.

Woop et al. calculate these coordinates in the following manner. Let $T = ABC$ be the 2D triangle. The scaled barycentric coordinates U, V, and W are:

$$U = C_\mathrm{x}B_\mathrm{y} - C_\mathrm{y}B_\mathrm{x}$$

$$V = A_\mathrm{x}C_\mathrm{y} - A_\mathrm{y}C_\mathrm{x}$$

$$W = B_\mathrm{x}A_\mathrm{y} - B_\mathrm{y}A_\mathrm{x}$$

$det = U + V + W$ is the determinant of the system of equations. Woop et al. reject the triangle if either $U < 0$, $V < 0$, $W < 0$, or $det = 0$. If these tests pass, they calculate the scaled parametric distance T. This is done by multiplying the z-components of the transformed triangle vertices:

$$T = UA_\mathrm{z} + VB_\mathrm{z} + WC_\mathrm{z}$$

Note that the vertices' z-components do not need to be transformed before the earlier tests have passed. As Woop et al.'s barycentric coordinates are not normalized, T still has to be divided by det. Because divisions are computationally expensive, they first test whether the distance T is valid. If the distance is not positive (i.e., $T \leq 0$), the

intersection would be before the ray. In this case, the intersection test can be aborted. If the intersection would be behind an already found intersection (i.e., $T \geq det \cdot t_{hit}$, where t_{hit} is the (unscaled) parametric distance of the already found intersection, or "hit"), the intersection test can be aborted as well. Only after these two distance tests, the scaled parametric distance $t = \frac{T}{det}$ is calculated. Woop et al. use single-precision floating point operations, with a fallback to double-precision in special cases. They found the performance of their algorithm to be comparable to the Möller-Trumbore algorithm [WBW13].

Woop et al. furthermore analyze other algorithms in terms of watertightness. An algorithm using the barycentric coordinates u, v, and w cannot provide watertight results if the third triangle edge is tested against $u + v$, i.e., if w is represented by $1 - u - v$. This can lead to different results when the same edge is tested again for a neighboring triangle in the mesh. Woop et al. state that Dammertz and Keller's algorithm [DK06] provides watertight results, but its subdivision-based nature is inefficient in terms of performance. Woop et al. show that their own algorithm is significantly faster than Dammertz and Keller's [WBW13].

Pharr et al. (2016) [PJH16d] do no longer use the Möller-Trumbore algorithm in PBRT, version 3. Instead, they use the same algorithm as Woop et al. [WBW13], differing only in terminology and in the direction the triangle edges are viewed from [PJH16c].

Amanatides and Choi (1997) state that an approach storing 2D transformation matrices (and their inverses) requires too much memory for triangle meshes [AC97]. They amortize computation over neighboring triangles. They present two approaches. The first one, instead of triangle vertices, stores three plane equations, one per triangle edge. These are used to conduct the intersection test. If the ray intersects the triangle, the parametric t value is calculated. Their second approach utilizes Plücker coordinates to check on which side of the triangle's edges the ray is. If the ray is not on the same side of all three edges, it cannot intersect the triangle. The more effective approach is the one using Plücker coordinates. It is about 35% faster than the boundary plane approach. [AC97].

Instead of conducting a ray-plane test followed by a 2D test, Kensler and Shirley (2006) directly test for intersection in 3D. They analyze existing 3D intersection tests (like Plücker tests and other signed volume/area approaches) and state that most of them are mathematically equivalent. They present an optimization method for 3D intersection tests. They first perform operation counting to minimize the number of operations. Then, they perform an exhaustive search for how their code should be written by letting a genetic algorithm modify the code. Their algorithm performs better than the Möller-Trumbore algorithm, however, they did not examine if the direct 3D test is as efficient as a ray-plane test followed by a 2D test [KS06].

Shevtsov et al. (2007) propose a hybrid intersection test based on both barycentric coordinates and an optimized version of the Plücker test. They first perform a fast hit Plücker test to check whether the ray intersects the triangle. If there is an intersection, they use barycentric coordinates to calculate the intersection point. Furthermore, they

use precomputed values and pay special attention to axis-aligned triangles. They flag these triangles, because for these an intersection point can be calculated in a simpler way [SSK07]. Shevtsov et al. show that their intersection test is faster than Benthin's and claim to achieve real-time performance on modern CPUs. Furthermore, they report that flagging axis-aligned triangles and processing them separately has enhanced their algorithm's performance, especially for scenes with a great number of axis-aligned triangles. This is the case if scenes contain, for example, buildings with accordingly aligned doors, windows, and walls. Even for indoor scenes, Shevtsov et al.'s tests reported that more than 50% of the rays hit axis-aligned triangles [SSK07].

Wald (2004) analyzes the cluster-based, real-time ray tracing system RTRT/OpenRT of Saarland University. He describes a projection method which is an optimization of the barycentric coordinate test. A modified version of it is used in the RTRT core [Wal04].

Havel and Herout (2010) present an intersection algorithm based on Wald's and Shevtsov et al.'s algorithms, using precomputed values. They calculate the intersection point in 3D instead of 2D. Their algorithm is precomputing faster and is benefitting from better performance on modern CPUs. It outperforms other algorithms, especially for single-ray intersection. Havel and Herout further calculated the mean squared error of the intersection coordinates for all three algorithms, finding that Shevtsov et al.'s method is the most and Wald's the least accurate, however, the differences are marginal [HH10].

Segura and Feito (1998, 2001) first calculate the ray-plane intersection point, then transform the triangle and this intersection point into a 2D coordinate system. Their algorithm avoids complex operations like divisions and is based on the study of signs [SF98]. Instead of representing a ray by its origin and direction, they treat it as a "segment", defined by two of its points. Their algorithm performs better when the number of intersections is low. Furthermore, not relying on divisions, trigonometric functions, or other complex operations increases the algorithm's robustness [SF01]. Segura and Feito's 1998 algorithm determines whether there is an intersection between the ray and the triangle, but not the actual intersection point [SF98][JONP14]. In their algorithm from 2001, they also primarily do not calculate the intersection point, but calculate it on demand if an intersection has been determined. Still, they state that their algorithm is best suited for inclusion tests or other applications that do not require the actual intersection point [SF01]. Jiménez et al. (2014) also note that, depending on the application, the computation of the explicit intersection point may not be necessary and can thus be omitted for performance gain, or only be calculated if it is certain that an intersection takes place [JONP14].

Jiménez et al. (2010) propose an algorithm that computes the parametric t value, and, subsequently, the barycentric coordinates of the intersection point. As single triangles are usually part of a larger triangle mesh, Jiménez et al. share information about triangles with neighboring triangles and reuse calculations. They emphasize the fact that their algorithm can compute wheter the intersection lies on a vertex, on an edge, or inside the traingle with no additional cost [JSF10].

Jiménez et al. (2014) [JONP14] implemented Badouel's [Bad90], Möller and Trumbore's [MT97], Segura and Feito's [SF98][SF01], Jiménez et al.'s [JSF10], Woop et al.'s [WBW13], and Kensler and Shirley's [KS06] algorithms for the GPU as well as, parallelized, for the CPU. They state that, in general, Möller and Trumbore's and Jiménez et al.'s are the most efficient [JONP14]. Jiménez et al. further compare different ray tracing features in CPU- and GPU-based algorithms. Some of them, e.g., the calculation of barycentric coordinates, show no noticable difference when being implemented for the CPU or the GPU [JONP14].

Held's (1997) ERIT system provides intersection routines for different types of primitives, including rays and triangles. It does not test against zero, but uses small epsilon values. To simplify computations, it transforms the intersection problem from 3D to 2D, using the coordinate plane resulting from dropping the triangle normal's highest-magnitude component [Hel97].

Bikker (2007) describes Arauna, a ray tracer developed at the NHTV Breda University of Applied Science with the goal of delivering real-time ray tracing for games. It originally followed Wald's [Wal04] approach and is implemented in C++ [Bik07]. Bikker further discusses other code-based optimizations like inlining functions and using the *const* keyword whenever possible [Bik07]. Another interactive ray tracer is Manta, which is designed for supporting engineering and scientific visualization [BSP06].

Georgiev and Slusallek (2008) present a generic library called RTfact that can be used for implementing ray tracing applications. It utilizes C++ features and follows generic programming paradigms [GS08]. Georgiev et al. (2008) created the real-time scene graph library RTSG, which they also tested with RTfact [GRHS08]. Georgiev and Slusallek name Pharr and Humphreys' early PBRT [PH04] and the Arauna system [Bik07] as two extremes of trading flexibility and performance, with PBRT being flexible but slow and Arauna being efficient but highly specialized. However, they also note that PBRT does not aim for performance, but physical realism [GS08].

Some algorithms, especially when being designed for real-time ray tracing, rely on SIMD (single instruction, multiple data) packaging and the SSE (streaming SIMD extensions) paradigm. According to Wald (2004), ray tracing can be sped up by packing together several rays and do parallel calculations [Wal04]. Havel and Herout (2010) discuss the fact that packed rays have to be coherent, which is only the case for primary rays, i.e., camera rays. Effectively using SIMD instructions becomes difficult after rays have been reflected by objects in multiple directions [HH10]. Noguera et al. (2009) also note that Wald's idea is valid for coherent, primary rays, but not for secondary rays. Based on his idea, they developed an algorithm that enables parallelization for incoherent rays as well. They compared their method with Möller and Trumbore's and found that their algorithm was faster [NUH09]. Bikker (2007) states that in Arauna, which is based on Wald's ideas, only shadows can be efficiently calculated using ray packets [Bik07]. Jiménez et al. (2014) also address the drawbacks of SSE-based programming, another one being the difficulty of correct ray packaging. The CPU programmer is responsible for how the rays are packed into SIMD packages and has to take scalability and granularity control

into account. Jiménez et al. describe high-performance GPU ray tracing as being more feasible [JONP14]. Wald et al. (2014) address the problem of packet tracing for incoherent rays in their open source CPU framework Embree. Embree employs hybrid techniques which allow switching between packet and single ray intersection methods [WWB+14].

Parker et al. (2010) note that the APIs of interactive ray tracers like Arauna [Bik07] or Manta [BSP06] are system-specific and not designed as general purpose solutions [PRS+10]. They describe OptiX, a high-performance, flexible, general purpose ray tracing engine. It offers a programmable ray tracing pipeline for implementing ray tracing applications. These are not limited to only the graphics domain, but may also be used for, e.g., collision detection or the simulation of sound propagation. Parker et al. further present several different, interactive applications built with OptiX [PRS+10].

Ferko and Ferko (2015) propose a ray-plane test, 2D transformations and a number of early termination tests. See Section 4.1 for a more detailed description of their algorithm. Löfstedt and Akenine-Möller (2005) state that "there does not exist such a thing as the fastest ray-triangle intersection algorithm" [LAM05]. Results differ depending on the used compiler, computer, the ray-triangle hit-rate, and the type of triangle data. They present an evaluation framework for ray-triangle intersection algorithms and published the source code of a simplified version of it called Ray-Triangle Advisor. Löfstedt and Akenine-Möller criticize the fact that ray tracing algorithms are not being evaluated in a consistent way, which makes it difficult to find the most efficient algorithm for a specific application. Among others, they report evaluation problems to be

- each tested algorithm having the same predefined set of precomputed data available,
- the algorithms not reporting the same amount of hits (i.e., not following a consistent definition of what a "hit" is), and
- the test set not covering different hit-rates.

Löfstedt and Akenine-Möller also state that some algorithms perform better for lower, some for higher hit-rates [LAM05].

3.2 Comparison and summary of existing approaches

There are generally two ways of tackling the ray-triangle intersection problem regarding the coordinate space in which the calculations are done. One is to compute intersections directly in 3D space, as is done by Kensler and Shirley [KS06], Havel and Herout [HH10], and Amanatides and Choi [AC97]. The other technique is to transform the intersection problem into 2D space, thus simplifying subsequent calculations. This is done by Held [Hel97], Woop et al. [WBW13][WSS05], Badouel [Bad90], Möller and Trumbore [MT97], Baldwin and Weber [BW16], Schmittler et al. [SWW+04], and Ferko and Ferko [FF15].

Badouel recommends projecting the intersection problem onto the plane perpendicular to the axis of the normal vector's highest-magnitude component. For example, if the z-component of the triangle normal has the highest absolute value, the xy-plane will be selected, and so on [Bad90]. This diminishes numerical problems [Hel97] and helps avoiding degenerate triangles in 2D space [Bad90][O'R98]. Baldwin and Weber define their "free vector" based on a similar method [BW16]. Woop et al. select a transformation that lets the ray start at the origin and being directed along one coordinate axis, with unit length. They choose the axis for which the ray's component has the highest magnitude. This means that the 2D transformation does not depend on the triangle, but on the ray. At run-time, the triangle vertices are transformed [WBW13]. Pharr et al. also use this transformation in PBRT, version 3 [PJH16c]. Möller and Trumbore choose the transformation so that it would transform the triangle to a unit triangle at the origin, with two of its edges aligned with the z- and y-axis, respectively, and that it would align the ray with the x-axis. At run-time, the ray origin is transformed [MT97]. Baldwin and Weber's [BW16], Schmittler et al.'s [SWW$^+$04], and Woop et al.'s [WSS05] transformation would transform the triangle onto the xy-plane with unit-length edges. At run-time, only the ray is transformed. Ferko and Ferko suggest two 2D transformations onto the xy-plane. Their unit triangle approach transforms the triangle vertices onto $(0,0),(0,1),(1,0)$. The pruning approach transforms the triangle's longest edge onto the x-axis with unit-length, while preserving similarity. At run-time, the ray-plane intersection point is transformed [FF15]. Amanatides and Choi also discuss 2D transformation onto the xy-plane so that the triangle's vertices become $(0,0),(0,1),(1,0)$ [AC97].

Another distinction can be made regarding how the intersection point is calculated (or whether there is an intersection or not in the first place). The two prevalent approaches are barycentric coordinates and Plücker coordinates. Barycentric coordinates are used by Havel and Herout [HH10], Badouel [Bad90], Möller and Trumbore [MT97], Jiménez et al. [JONP14], Baldwin and Weber [BW16], and Ferko and Ferko's unit triangle approach [FF15]. Plücker coordinates are employed by Kensler and Shirley [KS06], Benthin [Ben06], and Ferko and Ferko's pruning approach [FF15]. Held implemented both variants into his ERIT system and found that using Plücker coordinates is faster [Hel97]. Amanatides and Choi also found their algorithm to be more efficient when using Plücker coordinates [AC97]. A disadvantage of using Plücker coordinates is that this does not deliver the barycentric coordinates of the hit point, which may be required by a ray tracer's further operations. Woop et al. [WBW13] and Pharr et al. [PJH16d] use Benthin's Plücker approach and subsequently calculate the barycentric coordinates. Shevtsov et al. use Plücker coordinates for a fast hit test at first to determine whether there is an intersection or not, and if there is, they calculate the intersection point using barycentric coordinates [SSK07].

Some algorithms do not operate with rays represented by an origin and a direction, but rather view them as segments, defined by two points, and do a segment-triangle intersection test instead of a ray-triangle intersection test [Hel97][SF98][SF01][JSF10]. Jiménez et al. found that this only improves perfomance for Segura and Feito's [SF01]

and for Jiménez et al.'s [JSF10] algorithms, and only when they are implemented for the CPU. They further note that the segment notation is just a special case of the ray notation [JONP14]. Shevtsov et al. do not represent triangles via three vertices, but via one vertex and two edges [SSK07]. Shumskiy (2013), when comparing the Möller-Trumbore algorithm [MT97] with a 2D unit triangle approach, represents triangles directly by their 2D transformation matrix in the latter one [Shu13].

Some algorithms first calculate the intersection point between the ray and the plane containing the triangle, then test whether this point lies within the triangle. This is done by Badouel [Bad90] and Ferko and Ferko [FF15]. Amanatides and Choi also consider this approach [AC97]. Segura and Feito use a plane intersection test to determine the intersection point, if demanded [SF01].

Many implementations reported in the literature make use of single-precision floating type variables [BW09][Bik07][GS08][HH10][KS06]. Held's ERIT uses double-precision types [Hel97]. Woop et al. [WBW13] and Pharr et al. [PJH16d] principally use floating point variables, but employ a fallback to double values when higher precision is required. Parker et al.'s OptiX engine supports double-precision values for user-supplied programs, but stores rays in single-precision [PRS+10].

Small epsilon values are widley used in the literature [AC97][DK06][Hel97][WBW13]. Epsilon values help avoiding self-intersection of triangles with secondary rays [DK06], enhance (but not verifiably ensure) watertightness [WBW13] and can increase bounding volumes so that small rounding errors do not lead to the triangle being incorrectly rejected [WBW13][FF15].

Many algorithms use precomputed data for ray-triangle intersection. Triangles may, for example, store their normals or 2D transformation matrices. Of course, rays may store ray-specific transformations as well [PJH16c]. Jiménez et al. (2014) emphasize the performance improvement achieved by precomputed data in CPU-based algorithms [JONP14].

Other algorithms, like Wald's [Wal04], speed up intersection tests using SIMD technology and ray packets to test multiple rays for intersection at the same time. Wald's ideas are adapted in the Arauna game engine [Bik07], by Havel and Herout [HH10], and by Noguera et al. [NUH09]. Some algorithms aim at real-time performance and interactivity, like the one used in the Arauna engine [Bik07] or the one used in the Manta ray tracer, which also uses SIMD instructions and ray packeting [BSP06]. Shevtsov et al. use SIMD instructions to speed up their Plücker coordinates operations. They claim to achieve real-time performance on modern CPUs [SSK07].

Finally, Jiménez et al. [JSF10], Baldwin and Weber [BW16], and Ferko and Ferko [FF15] explicitly state that they employ early termination strategies to discard a triangle which is not intersected by the ray as soon as possible. An advantage of early exit points is that costly operations (like divisions) can be delayed until it is known that they are necessary [MT97][WBW13][BW16].

CHAPTER 4

Methodology

This chapter describes the methodology of this thesis. Section 4.1 presents the ray-triangle intersection algorithm proposed by Ferko and Ferko [FF15]. Section 4.2 gives an overview of the methods and concepts used to implement this algorithm into PBRT. The following sections illustrate the construction of a bounding box (Section 4.3) and bounding sphere/circle (Section 4.4) of a triangle, the bounding box test (Section 4.5), the ray-sphere intersection test (Section 4.6), the ray-circle test (Section 4.7), and the 3D-to-2D transformation approaches and subsequent 2D tests (Section 4.8) in greater detail. Section 4.10 specifies the modifications in PBRT and the used language. Finally, Section 4.11 describes the methodological approach of analyzing and evaluating the algorithm.

4.1 Ferko and Ferko's algorithm

Ferko and Ferko (2015) [FF15] propose the following approach for fast ray-triangle intersection.

1. A ray-sphere intersection test determines whether the ray intersects the bounding sphere of the triangle. If there is no intersection, the triangle is rejected.

2. A ray-plane intersection test determines whether the ray intersects the plane in which the triangle lies. If there is no intersection, the triangle is rejected. If there is an intersection, the intersection point is stored as the potential ray-triangle intersection point.

3. A bounding box test determines whether the ray-plane intersection point is inside the bounding box of the triangle. If it is not, the triangle is rejected.

4. A 2D transformation is used to transform the ray-plane intersection point into a 2D coordinate system where additional, cheap tests can be performed. Two variants of 2D transformations and additional tests are proposed:

   a) Unit triangle approach: The intersection point is transformed using a unit triangle transformation matrix which would map the triangle vertices to $(0,0),(0,1),(1,0)$. A series of simple 2D tests determines whether the transformed intersection point lies within the 2D triangle. If so, the ray intersects the triangle at the calculated (3D) ray-plane intersection point.

   b) Pruning approach: The intersection point is transformed using a similarity preserving transformation matrix which would map the triangle's longest edge to the x-axis with length 1, and the third vertex into the coordinate system's first quadrant accordingly. Then, the 2D tests "prune" the triangle's surrounding areas to determine whether the intersection point lies within the triangle. This is done using Plücker-based signed edge tests.

4.2 Overview of used concepts

The basic concepts discussed in this thesis are that of rays and triangles, and the primary goal is calculating their intersection points. To implement Ferko and Ferko's algorithm, other used concepts are the bounding box and bounding sphere as well as their respective intersection tests and/or inside/outside tests. We also use bounding circles and bounding circle tests. Cramer's rule is used for solving systems of equations. The triangle plane has to be calculated and the ray-plane intersection point has to be computed. For the 2D transformation, we principally use 4×4 transformation matrices, but may not store and process columns and rows that are not needed for the intented transformation. The following sections describe these concepts in greater detail.

4.3 Bounding box of a triangle

A bounding box is a bounding volume in the shape of a cuboid. Ray tracing algorithms typically use axis-aligned bounding boxes (AABBs) [PJH16b][BW09][MW06][Shu13]. Another type of bounding boxes are oriented bounding boxes (OBBs), which are not necessarily axis-aligned [PJH16b].

A 3-dimensional AABB can be described by either pair of two opposite vertices. Another way to describe the AABB is to express one vertex and the respective length of its edges in x, y, and z direction [PJH16b]. Figure 4.1 depicts an AABB represented by a vertex pair.

A straightforward approach to construct the AABB of a shape is to find out the minimum and maximum x, y, and z coordinate values of its points. Let these values be x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, and z_{max}. Then the AABB spans between the two opposite vertices $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$. Note that different AABBs may be created

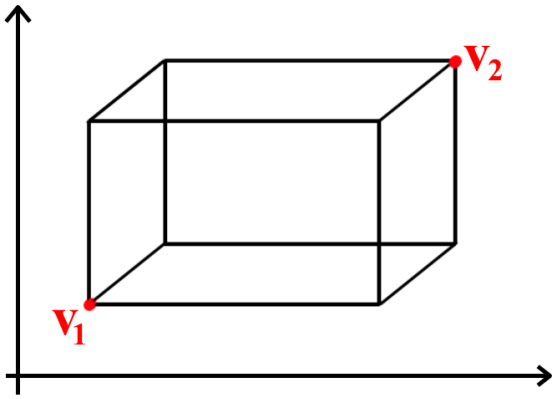


Figure 4.1: A bounding box represented by two opposite vertices v_1 and v_2.

depending on whether the minimum and maximum values are calculated in object space or in world space. For triangles, a tighter AABB may be computed if the triangle is first transformed from object space into world space and then its AABB is constructed, instead of constructing the AABB in the triangle's object space. For example, in PBRT, ray-triangle intersection requires the triangle's vertices to be in world space. If an AABB has been constructed in the triangle's object space, it would have to be transformed into world space. There, the bounding box will most likely not be axis-aligned anymore. Bounding this bounding box will result in an AABB in world space, however, directly constructing the AABB of the triangle in world space is the better option [PJH16c] (see also Figure 4.2).

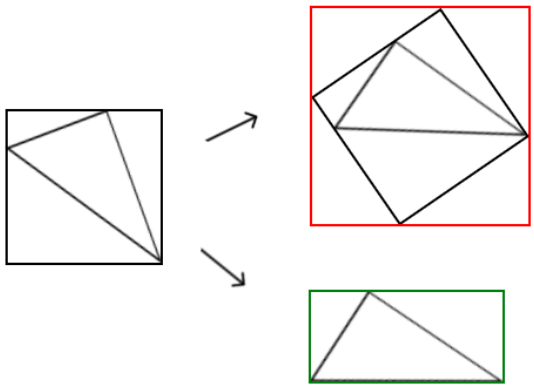


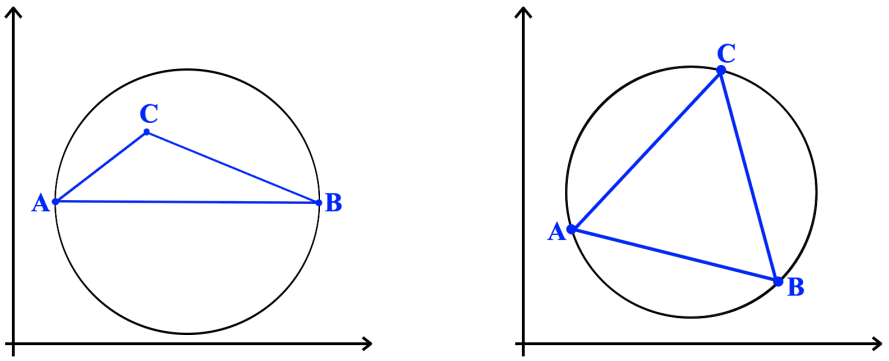
Figure 4.2: Creating an efficient bounding box of a triangle. Transforming the object space triangle's AABB (left) into world space and then finding the AABB of the bounding box (top right) may result in a bigger AABB than when the triangle's vertices are first transformed into world space and then bounded (bottom right). Image reproduced from Pharr et al. [PJH16c]

4.4 Bounding circle and bounding sphere of a triangle

The minimum bounding circle (sphere) of a triangle ABC is the smallest possible circle (sphere) encompassing each of the triangle's vertices A, B, and C. It has a center point P_C (which lies within the triangle plane) and a radius r.

Note that, for the sake of simplicity, the following considerations describe the construction of a 2D bounding circle. However, the same circumstances apply to a 3D bounding sphere. Ericson (2007) [Eri07] constructs a bounding circle as follows. Depending on the triangle's form, Ericson identifies two cases to consider (see Figure 4.3):

1. Two vertices of ABC lie on the circle and one vertex lies inside of the circle.
2. All three vertices A, B, and C lie on the circle.



(a) (b)

Figure 4.3: Two cases of the triangle vertices' location within a bounding circle: (a) Two vertices lie on the circle, one vertex lies inside of the circle. (b) All three vertices lie on the circle.

Regarding case 1, let the two vertices lying on the circle be A and B. For the bounding circle to be minimal, A and B must lie on the diameter of the circle. Furthermore, the edge $\overrightarrow{AB}$ is the longest edge of ABC. In case 2, the circle's center point P_C is the point equidistant to all three vertices.

To construct the bounding circle, Ericson first assumes case 2 and, if necessary, derives case 1 later. Let P be a point lying within the triangle plane. P can be expressed as

$$P = A + s(B - A) + t(C - A), \tag{4.1}$$

where s, t and $1 - s - t$ are the barycentric coordinates of P regarding ABC. They are calculated as follows. As P shall be equidistant to A, B, and C, the squared distance of P to each of the vertices has to be equal:

$$|P - B|^2 = |P - A|^2$$

$$|P - C|^2 = |P - A|^2$$

Rearranging these equations for P gives:

$$P = \frac{A^2 - B^2}{2(A - B)}$$

$$P = \frac{A^2 - C^2}{2(A - C)}$$

These terms now can be substituted for P in Equation 4.1. Rearranging terms results in the following system of equations:

$$2(A - B)(B - A)s + 2(A - B)(C - A)t + (A - B)(A - B) = 0$$

$$2(A - C)(B - A)s + 2(A - C)(C - A)t + (A - C)(A - C) = 0$$

Further rearranging yields:

$$(B - A)(B - A)s + (B - A)(C - A)t = \frac{1}{2}(B - A)(B - A)$$

$$(C - A)(B - A)s + (C - A)(C - A)t = \frac{1}{2}(C - A)(C - A)$$

This system can be solved for s and t using Cramer's Rule (see Section 2.4) [Eri07]. As this requires dividing by the determinant of the coefficient matrix, we must check if the determinant is 0. This is the case for triangles whose vertices lie on a line. In a ray tracing framework, these triangles can be marked as degenerate triangles which are never intersected by a ray.

There are generally three cases regarding the position of an encompassing circle's center point P which is equidistant to all three vertices of the triangle ABC (see Figure 4.4):

i) If P lies inside of ABC, then ABC is an acute triangle.

ii) If P lies outside of ABC, then ABC is an obtuse triangle.

iii) If P lies on ABC, then ABC is a right triangle.

Looking at the barycentric coordinates s, t, and $1 - s - t$, if $s > 0$, $t > 0$, and $1 - s - t > 0$, then P lies inside ABC and all three vertices A, B, and C lie on the circle. In this case, the bounding circle's center point is indeed P, thus $P_C = P$. As P is equidistant to A, B, and C, the circle's radius is the distance between P and one arbitrary triangle vertex.

If either s, or t, or $1 - s - t$ is negative, P lies outside of ABC. Figure 4.4b shows that, in this case, the corresponding circle is not the smallest possible bounding circle. However, the edge of ABC that P lies outside of is the longest edge of ABC (see Figure 4.5) [Eri07].

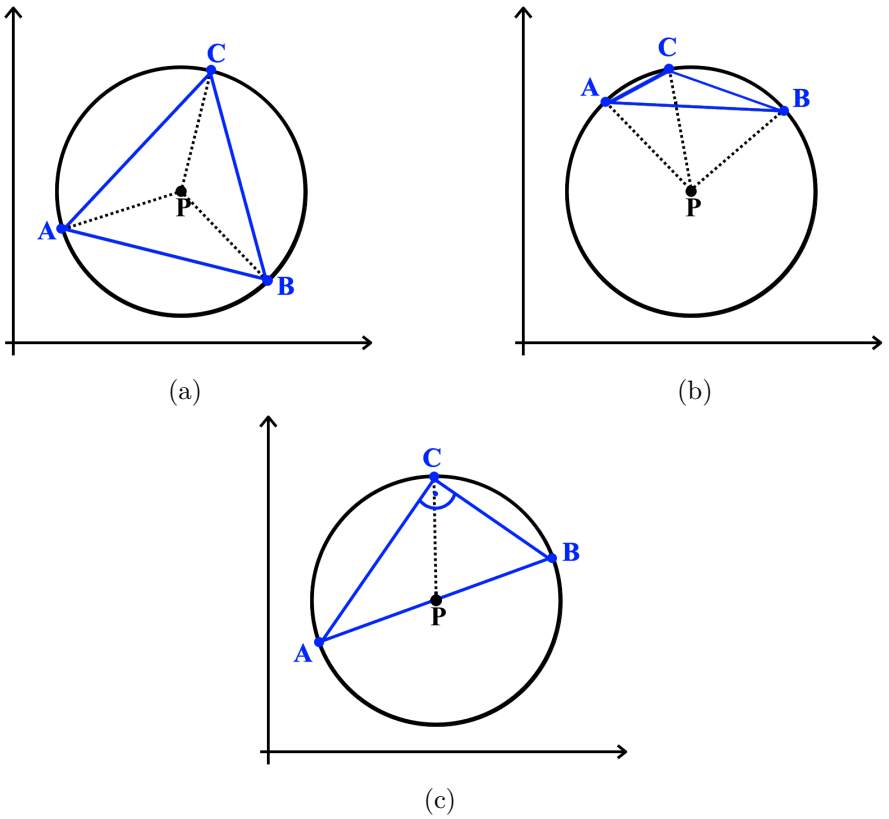


Figure 4.4: Three cases of an encompassing circle with a center point P equidistant to a triangle ABC's vertices: (a) If ABC is an acute triangle, P lies inside of ABC. (b) If ABC is an obtuse triangle, P lies outside of ABC. (c) If ABC is a right triangle, P lies on ABC. It can be seen that for b), the encompassing circle is not the smallest possible bounding circle.

This corresponds to case 1, as described above. The respective edge is the diameter of the circle. Without loss of generality, let this edge be $\overrightarrow{AB}$. P is the center point of an encompassing circle, but not the center point P_C of the smallest possible (bounding) circle. However, as the negative barycentric coordinate indicates the diameter of the bounding circle, its center point lies exactly between A and B, and the radius is the distance to either A or B [Eri07]. The different cases regarding the outcome of s, t, and $1 - s - t$ can be seen in Figure 4.6.

If the AABB of a triangle is known, the bounding sphere of the AABB can be computed as well. Let the AABB be represented by two opposite vertices B_{min} and B_{max}. The sphere's center point P_C lies in the center of the AABB, thus:

$$P_C = \frac{B_{min} + B_{max}}{2}.$$

(a) (b)

Figure 4.5: The encompassing circle in (a) with a center point P equidistant to A, B, and C is not the minimal bounding circle of an obtuse triangle ABC. P lies outside of the edge $\overrightarrow{AB}$. The diameter of the smallest possible bounding circle corresponds to this edge, as can be seen in (b). (a)'s circle has the radius $r = 2$. For (b), $r = 1.5$.

Now, the sphere's radius r is the distance to each of the AABB's vertices. Using B_{max},

$$r = |B_{max} - P_C|.$$

Note that the AABB's bounding sphere is likely not the smallest possible bounding sphere encompassing the triangle, however, it may still be useful for certain applications [PJH16b]. See Figure 4.7 for an example of a triangle's AABB's bounding sphere.

4.5 Bounding box test

Checking whether a point lies within an AABB is straightforward. A point P is inside a triangle's AABB defined by two opposite vertices $B_{min} = (B_{minx}, B_{miny}, B_{minz})$ and $B_{max} = (B_{maxx}, B_{maxy}, B_{maxz})$ if the following conditions are met:

$$B_{minx} \leq P_x \leq B_{maxx}$$

$$B_{miny} \leq P_y \leq B_{maxy}$$

$$B_{minz} \leq P_z \leq B_{maxz}$$

To test whether a ray intersects a bounding box, Pharr et al. [PJH16c] treat a bounding box as the intersection of the three regions between parallel planes of the box. They call these regions "slabs" and intersect the ray with each of the slabs, consecutively, to find two parametric t positions where the ray enters and leaves the slab. Along this range $[t_0, t_1]$, the ray is inside the bounding box. If necessary, the range is updated after a new slab has been intersected with the ray, so the value of t_0 found in the first intersection may increase and the value of t_1 may decrease with each intersection. If, at one point, the value of t_0 exceeds the value of t_1, the interval is degenerate and no intersection is computed. If, after intersecting all three slabs, a valid range $[t_0, t_1]$ has been found, t_0 and t_1 are the ray-box intersection points [PJH16c].

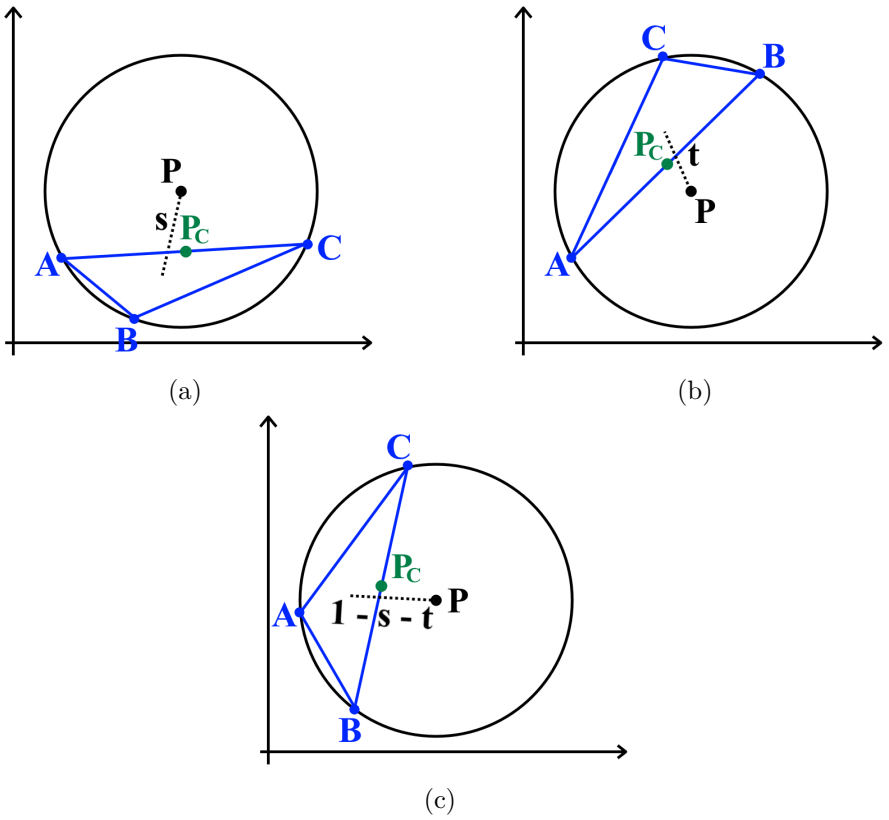


Figure 4.6: Three cases of how s and t influence the position of the bounding circle's center point P_C given the vertex-equidistant encompassing circle's center point P. (a) $s \leq 0$. P_C lies exactly between A and C: $P_C = 0.5(A+C)$. (b) $t \leq 0$. P_C lies between A and B: $P_C = 0.5(A+B)$. (c) $1-s-t \leq 0$. P_C lies between B and C: $P_C = 0.5(B+C)$.

4.6 Ray-sphere intersection test

We use a ray-sphere intersection test to determine whether a ray R intersects a triangle's bounding sphere. An easy approach to do this is to execute the intersection test in the sphere's object space, i.e., in a coordinate system where the sphere's center point lies at the origin. The implicit surface equation of such a sphere is:

$$x^2 + y^2 + z^2 - r^2 = 0,$$

where r is the sphere's radius [PJH16c]. R can be represented by:

$$p = R_o + \overrightarrow{R_d}t$$

where R_o is the ray's origin point, $\overrightarrow{R_d}$ is the ray's direction vector, and t is the parametric distance of the point p along the ray. Note that this approach needs R to be in the sphere's object space, so the corresponding transformation may have to be applied to R.

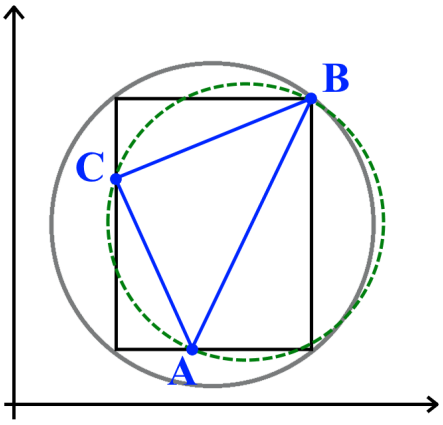


Figure 4.7: Constructing a bounding sphere from a triangle ABC's AABB. Note that this sphere is a looser fit than ABC's actual bounding sphere (represented by dashed lines).

Substituting the ray's equation into the sphere's equation gives

$$(R_{ox} + t\overrightarrow{R_{dx}})^2 + (R_{oy} + t\overrightarrow{R_{dy}})^2 + (R_{oz} + t\overrightarrow{R_{dz}})^2 = r^2.$$

Note that R_{ox} represents the x-component of R_o, R_{oy} the y-component of R_o, and so on. Solving for t gives the parametric positions along R where the ray intersects the sphere. There can be zero, one, or two intersection points depending on the number of non-imaginary solutions for t. For ray-sphere intersection in a ray tracing framework, determining the nearest valid intersection point is sufficient [PJH16c].

Let the sphere's center point be P_C. Another approach to determine whether a ray R intersects the sphere is to calculate the dot product of the ray's direction vector $\overrightarrow{R_d}$ and the vector $\overrightarrow{l}$, where

$$\overrightarrow{l} = P_C - R_o.$$

The dot product $\overrightarrow{a} \cdot \overrightarrow{b}$ corresponds to a projection of $\overrightarrow{a}$ onto $\overrightarrow{b}$. Take a look at Figure 4.8. Let $s = \overrightarrow{l} \cdot \overrightarrow{R_d}$ be the length of $\overrightarrow{l}$ projected onto $\overrightarrow{R_d}$. Then, we can use s, $|\overrightarrow{l}|$, and the Pythagorean theorem to calculate m. If m is smaller than or equal to the sphere's radius, the ray intersects the sphere. These calculations can also be done using squared values, i.e., the square roots do not have to be extracted.

If s is negative, an additional condition must be met for the ray to intersect the sphere. Consider a ray originating inside the sphere and pointing outwards. The dot product $s = \overrightarrow{l} \cdot \overrightarrow{R_d}$ is negative. The ray is intersecting the sphere (see Figure 4.9a). Now, consider a ray of the same orientation outside of the sphere. Again, s is negative, and m may be smaller than the sphere's radius. However, no intersection takes place (see Figure 4.9b). Thus, for the case that s is negative, we must check whether R_o lies within the sphere. This can be done by comparing $|\overrightarrow{l}|$ to the sphere's radius. R_o lies within the sphere if

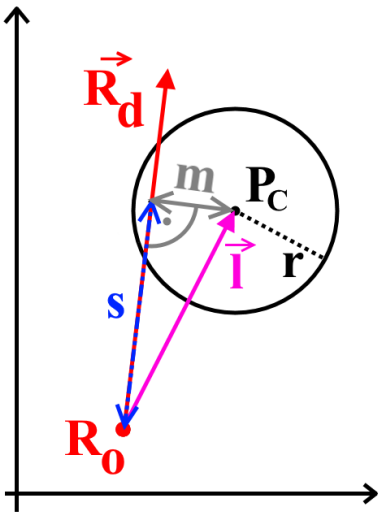
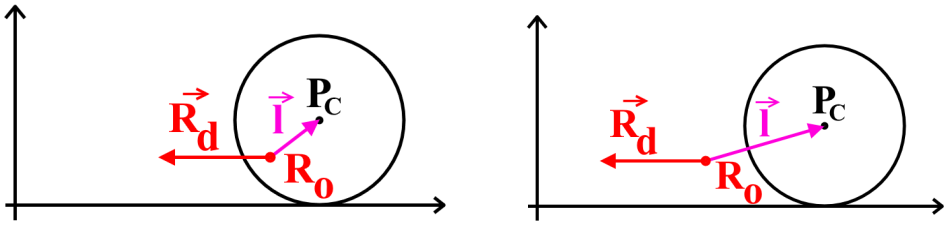


Figure 4.8: Using the dot product to project the vector $\vec{l}$ onto the ray's direction vector $\vec{R_d}$. We compare m to the sphere's radius r to determine whether the ray intersects the sphere. P_C is the sphere's center point. R_o is the ray's origin. s is the length of $\vec{l}$ projected onto $\vec{R_d}$.

$|\vec{l}| \leq r$ [AMHH08]. Note that the dot product method only determines whether there is an intersection or not, while the parametric method also determines the exact intersection point(s).



(a) (b)

Figure 4.9: Rays and spheres for which $\vec{l} \cdot \vec{R_d}$ is negative. In (a), the ray originates within the sphere and is pointing outwards, thus, an intersection takes place. In (b), the ray's direction is the same, but its origin is outside of the sphere, so there is no intersection. P_C is the bounding sphere's center point. R_o is the ray's origin.

4.7 Ray-circle test

As a bounding volume is most effective if it fits the underlying shape as tight as possible, we did not only implement a bounding sphere test, but also a bounding circle test. Since

a circle is planar, first the intersection point of the ray with the triangle plane has to be known. Then, the bounding circle test becomes a simple inside/outside test in 2D.

First, the squared distance $dist^2$ of the plane intersection point P_I to the center point of the bounding circle P_C is computed by calculating a dot product of the vector $(P_I - P_C)$ with itself:

$$dist^2 = (P_I - P_C) \cdot (P_I - P_C)$$

For P_I to lie inside the bounding circle, $dist$ hast to be smaller than or equal to the circle's radius r. To save the computational effort of extracting a square root, $dist^2$ can also be compared to r^2.

4.8 3D-to-2D transformation and 2D tests

Many ray-triangle intersection algorithms perform 3D-to-2D transformations to simplify further calculations. See Section 3.2 for an overview of possible transformations found in the literature. Ferko and Ferko [FF15] use a precomputed unit triangle transformation to transform a 3D point into a 2D coordinate system. This is the transformation which would transform the triangle vertices to the 2D points $(0,0),(0,1),(1,0)$. For their pruning approach, they propose a similarity preserving transformation [FF15]. The following subsections will describe both of these approaches. Note that the respective transformation matrix is constructed and stored for each triangle during pre-processing.

4.8.1 Unit triangle transformation and subsequent 2D tests

This subsection describes how to construct a matrix which transforms a 3D triangle to a 2D unit triangle. The unit triangle transform matrix (UTM) is supposed to transform the triangle's vertices to $(0,0),(1,0),(0,1)$. This way of ordering the vertices lets the coordinates of a transformed point correspond to PBRT's ordering of UV vertices. Thus, the transformed coordinates are the barycentric coordinates of the hit point with regard to the triangle. For the 2D tests it makes no difference whether the vertices are mapped to $(0,0),(0,1),(1,0)$, or to $(0,0),(1,0),(0,1)$.

Let the original triangle's vertices be $v_1 = (v_{1x}, v_{1y}, v_{1z})$, $v_2 = (v_{2x}, v_{2y}, v_{2z})$, and $v_3 = (v_{3x}, v_{3y}, v_{3z})$. We define a target matrix X. X's column vectors correspond to the points v'_1, v'_2, and v'_3 our vertices should be mapped to. Thus,

$$X = \begin{pmatrix} v'_{1x} & v'_{2x} & v'_{3x} & 0 \\ v'_{1y} & v'_{2y} & v'_{3x} & 0 \\ v'_{1z} & v'_{2z} & v'_{3x} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.2}$$

We gather the actual vertices of the triangle in a source matrix A:

$$A = \begin{pmatrix} v_{1x} & v_{2x} & v_{3x} & 0 \\ v_{1y} & v_{2y} & v_{3y} & 0 \\ v_{1z} & v_{2z} & v_{3z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.3}$$

For the UTM, the following dependency has to be met:

$$UTM \cdot A = X$$

To compute the UTM, we need the inverse of A, A^{-1}:

$$UTM = X \cdot A^{-1}$$

We implement and evaluate several ways of computing A^{-1} and, subsequently, the UTM (see also Section 5.5). After the plane intersection point has been transformed into a 2D coordinate system using the UTM, it has to lie within the unit triangle $(0,0),(1,0),(0,1)$. Note that only the intersection point has to be transformed. It can be shown that a 2D intersection point P has to meet the following conditions to lie within the triangle [FF15]:

$$P_x \geq 0$$

$$P_y \geq 0$$

$$|P_x + P_y| \leq 1$$

If these three conditions are met, the 3D plane intersection point is the ray-triangle intersection point (see Figure 4.10). Furthermore, the transformed intersection point's components P_x and P_y correspond to PBRT's barycentric coordinates b_1 and b_2, and $b_0 = 1 - b_1 - b_2$. Thus, P_x and P_y can be used for further operations like texture coordinates calculation or normal interpolation [SSK07].

Note that, strictly speaking, calculating the absolute value of $P_x + P_y$ is not necessary if the 2D tests are done in the above order. This test is only called if both P_x and P_y are positive, otherwise the intersection algorithm will terminate beforehand. Adding two positive values can only result in another positive value, so one can save the call of an *abs()* function and just test for:

$$P_x + P_y \leq 1.$$

4.8.2 Similarity preserving transformation and pruning

Ferko and Ferko discuss another form of 3D-to-2D transformation, along with a different way of determining whether the ray hits the triangle. In their unpublished paper, they outline the idea of a similarity preserving transformation of the triangle, where the longest side of the triangle is mapped onto the x-axis and the third vertex is mapped into the first quadrant accordingly. Ferko and Ferko present several steps to "prune" at each side

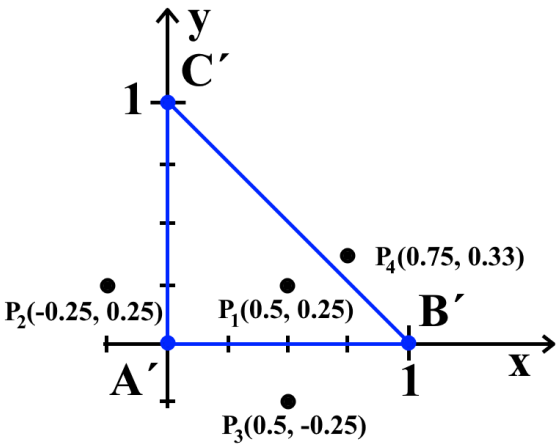


Figure 4.10: To lie within the unit triangle $A'B'C'$ with vertices $(0,0),(1,0),(0,1)$, a point P must fulfill the conditions: $P_x \geq 0$, $P_y \geq 0$, $P_x + P_y \leq 1$. This is only the case for P_1.

of the triangle. If the intersection point passes each of the pruning tests, it lies within the triangle. Furthermore, Ferko and Ferko mention the possibility of optimizing the pruning tests. They propose calculating the probabilities with which the ray will hit certain areas of the triangle's bounding circle [FF15].

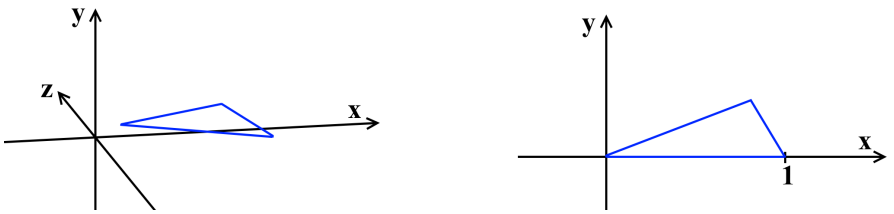
4.8.2.1 The similarity preserving transformation matrix

The unit triangle transformation matrix which maps the 3D triangle's vertices to the 2D points $(0,0),(1,0),(0,1)$ is most likely to deform the triangle in some way. In contrast, the similarity preserving transformation matrix (SPTM) transforms the 3D triangle into a 2D coordinate space while preserving similarity. The minimum requirements for the transformed triangle are as follows:

1. The triangle's longest edge is to be mapped onto the x-axis. Let this edge be $\overrightarrow{AB}$, with A and B being its vertices, and let A' and B' be the transformed points A and B.
2. A' has to be on $(0,0)$, while B' has to be on $(1,0)$.
3. Let C' be the third vertex C after being transformed with the SPTM. C' has to be mapped into the first quadrant of the 2D coordinate system so that the resulting triangle $A'B'C'$ is similar to ABC (see also Figure 4.11).

Several transformation matrices are needed to construct the SPTM. To transform a triangle like described above, the following steps need to be taken:

1. Translate the triangle so that A ends up at the origin.



(a) (b)

Figure 4.11: A 3D triangle (a) is transformed into 2D using the similarity preserving transformation matrix (b).

2. Rotate the triangle around the z-axis so that the y-value of B becomes 0.
3. Rotate the triangle around the y-axis so that the z-value of B becomes 0.
4. Rotate the triangle around the x-axis so that the z-value of C becomes 0. At this point, the intersection problem becomes a 2D problem.
5. Proportionally scale the triangle so that the x-value of B becomes 1.
6. If, after these steps, the y-value of C is negative, mirror C around the x-axis so that it is positive. The resulting vertices are $A' = (0, 0)$, $B' = (1, 0)$, and C' with positive x- and y-values.

These steps are sufficient for a functional similarity preserving transformation. However, another step can be added to improve the effectiveness of the later pruning: If C' lies in the left half of the triangle's circumscribed circle, i.e. if the x-value of C' is smaller than 0.5, C' can be mirrored around the line $(0.5/y)$. This moves C' into the right half while still preserving similarity, which increases the probability that the left pruning test will detect a non-hit. As will become obvious later in this section, the left pruning test is computationally cheaper than the right pruning test. In the following, constructing the SPTM according to these steps will be described.

Translate the triangle so that A ends up at the origin.

The following rotations require the vertices to be already translated, so first, we translate A, B, and C by $-A_x$ in x-, $-A_y$ in y-, and $-A_z$ in z-direction. For better distinction, the vertices will be named A_T, B_T and C_T throughout the whole transformation process.

$$A_T = (0, 0, 0)$$

$$B_T = B - A$$

$$C_T = C - A$$

Rotate the triangle around the z-axis so that the y-value of B becomes 0.

A matrix R_z for rotation around the z-axis by an angle ϕ has the following elements:

$$R_z = \begin{pmatrix} cos(\phi_z) & -sin(\phi_z) & 0 & 0 \\ sin(\phi_z) & cos(\phi_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.4}$$

For this particular rotation, ϕ_z can be seen as the angle that is spanned by the vector $\overrightarrow{A_T B_T}$ and the x-axis, while disregarding B_T's z-component. Let $\vec{X}$ be the vector $(1, 0, 0)$ and $\overrightarrow{A_T B_{Tn}}$ the normalized vector $(B_{Tx} - A_{Tx}, B_{Ty} - A_{Ty}, 0)$. Since A_T already lies in the coordinate system's center, $\overrightarrow{A_T B_{Tn}}$ can be written as:

$$\overrightarrow{A_T B_{Tn}} = \overrightarrow{B_{Tn}} = (\frac{B_{Tx}}{\sqrt{B_{Tx}^2 + B_{Ty}^2}}, \frac{B_{Ty}}{\sqrt{B_{Tx}^2 + B_{Ty}^2}}, 0)$$

Now, we can calculate $cos(\phi_z)$ and $sin(\phi_z)$ using the dot product and cross product, respectively:

$$cos(\phi_z) = \overrightarrow{B_{Tn}} \cdot \vec{X}$$
$$sin(\phi_z) = |\overrightarrow{B_{Tn}} \times \vec{X}|$$

As $\vec{X} = (1, 0, 0)$, these terms can be simplified to:

$$cos(\phi_z) = \overrightarrow{B_{Tnx}}$$
$$sin(\phi_z) = -\overrightarrow{B_{Tny}}$$

PBRT provides a *Transform* class for transforming points and vectors, but as only six values of the matrix are different from 0 (and two of these are 1), one can easily calculate the rotated vertices directly. A_T does not need to be transformed as it will stay at $(0, 0, 0)$. B_T's y-coordinate will result in 0, while its z-coordinate does not change at all. B_{Tx} can be calculated as follows:

$$B_{Tx} = cos(\phi_z) \cdot B_{Tx} - sin(\phi_z) \cdot B_{Ty} = \overrightarrow{B_{Tnx}} B_{Tx} + \overrightarrow{B_{Tny}} B_{Ty}$$

C_T will be transformed using a helper variable to store the original x-value:

$$tempX = C_{Tx}$$
$$C_{Tx} = cos(\phi_z) \cdot tempX - sin(\phi_z) \cdot C_{Ty} = \overrightarrow{B_{Tnx}} tempX + \overrightarrow{B_{Tny}} C_{Ty}$$
$$C_{Ty} = sin(\phi_z) \cdot tempX + cos(\phi_z) \cdot C_{Ty} = -\overrightarrow{B_{Tny}} tempX + \overrightarrow{B_{Tnx}} C_{Ty}$$

Note that this step has to be omitted if B_T already lies on the coordinates $(0, 0, z)$. In this case, the rotation is not only unnecessary, but it would result in a division by zero when normalizing $(B_{Tx} - A_{Tx}, B_{Ty} - A_{Ty}, 0)$.

Rotate the triangle around the y axis so that the z-value of B becomes 0.

The previously calculated points can now be used to compute the angle ϕ_y, spanned by the new vector $\overrightarrow{A_T B_T}$ and the x-axis. $\overrightarrow{A_T B_{Ty}}$ is 0, so

$$\overrightarrow{A_T B_{Tn}} = \overrightarrow{B_{Tn}} = (\frac{B_{Tx}}{\sqrt{B_{Tx}^2 + B_{Tz}^2}}, 0, \frac{B_{Tz}}{\sqrt{B_{Tx}^2 + B_{Tz}^2}}).$$

The matrix for rotation around the y-axis by the angle ϕ_y is:

$$R_y = \begin{pmatrix} cos(\phi_y) & 0 & sin(\phi_y) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\phi_y) & 0 & cos(\phi_y) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.5}$$

We use the dot product and the cross product to get the cosine and sine values:

$$cos(\phi_y) = \overrightarrow{B_{Tn}} \cdot \vec{X}$$

$$sin(\phi_y) = |\overrightarrow{B_{Tn}} \times \vec{X}|$$

These terms simplify to:

$$cos(\phi_y) = \overrightarrow{B_{Tnx}}$$

$$sin(\phi_y) = \overrightarrow{B_{Tnz}}$$

Again, both B_T and C_T need to be transformed for the next steps. B_{Tz} will become 0, and

$$B_{Tx} = cos(\phi_y) \cdot B_{Tx} + sin(\phi_y) \cdot B_{Tz} = \overrightarrow{B_{Tnx}} B_{Tx} + \overrightarrow{B_{Tnz}} B_{Tz}.$$

For C_T, we use a helper variable to store the original x-value:

$$tempX = C_{Tx}$$

$$C_{Tx} = cos(\phi_y) \cdot tempX + sin(\phi_y) \cdot C_{Tz} = \overrightarrow{B_{Tnx}} tempX + \overrightarrow{B_{Tnz}} C_{Tz}$$

$$C_{Tz} = -sin(\phi_y) \cdot tempX + cos(\phi_y) \cdot C_{Tz} = -\overrightarrow{B_{Tnz}} tempX + \overrightarrow{B_{Tnx}} C_{Tz}$$

Contrary to the previous rotation, there is no risk of dividing by 0 when normalizing. $|(B_{Tx}, 0, B_{Tz})|$ could only become 0 if $B_{Tx} = B_{Tz} = 0$. As we already transformed B_T so that $B_{Ty} = 0$, this would only be the case if $B = A$. This would mean that the triangle is degenerate, which has to be checked even before the construction of the SPTM when the vertices are reordered so that $\overrightarrow{AB}$ is the longest edge of the triangle.

Rotate the triangle around the x-axis so that the z-value of C becomes 0.

The matrix for rotation around the x-axis is

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\phi_x) & -sin(\phi_x) & 0 \\ 0 & sin(\phi_x) & cos(\phi_x) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.6}$$

This time, the angle ϕ_x is spanned by the newly generated vector $\overrightarrow{B_T C_T}$ and the y-axis. As we want to rotate around the x-axis, we disregard the x-components of B_T and C_T. B_{Ty} and B_{Tz} are both 0, so $\overrightarrow{B_T C_T} = (0, C_{Ty}, C_{Tz})$, and normalizing $\overrightarrow{B_T C_T}$ yields

$$\overrightarrow{B_T C_{Tn}} = \overrightarrow{C_{Tn}} = (0, \frac{C_{Ty}}{\sqrt{C_{Ty}^2 + C_{Tz}^2}}, \frac{C_{Tz}}{\sqrt{C_{Ty}^2 + C_{Tz}^2}}).$$

If we represent the y-axis by the vector $\vec{Y} = (0, 1, 0)$, we can use the dot product and the cross product to calculate the cosine and sine values:

$$cos(\phi_x) = \overrightarrow{C_{Tn}} \cdot \vec{Y}$$

$$sin(\phi_x) = |\overrightarrow{C_{Tn}} \times \vec{Y}|$$

As $\vec{Y}$ is $(0, 1, 0)$, these terms can be simplified to:

$$cos(\phi_x) = \overrightarrow{C_{Tny}}$$

$$sin(\phi_x) = -\overrightarrow{C_{Tnz}}$$

Only C_T will be affected by this transformation. C_{Tz} will become 0, and

$$C_{Ty} = cos(\phi_x) \cdot C_{Ty} - sin(\phi_x) \cdot C_{Tz} = \overrightarrow{C_{Tny}} C_{Ty} + \overrightarrow{C_{Tnz}} C_{Tz}.$$

This step has to be omitted if $C_{Ty} = B_{Ty} = C_{Tz} = B_{Tz} = 0$, i.e., C_{Tz} has already been eliminated. In this case, not only is the rotation unnecessary, but it would result in a division by zero when normalizing.

Scale the triangle so that the x-value of B becomes 1.

Now, the triangle does not have a single z-value different from 0 left, reducing the problem to 2D space. We use the x-value of B_T to uniformly scale the triangle so that B_{Tx} is 1 (i.e., the rightmost vertex of the transformed triangle is at point $(1, 0)$). The scaling matrix is:

$$S = \begin{pmatrix} \frac{1}{B_{Tx}} & 0 & 0 & 0 \\ 0 & \frac{1}{B_{Tx}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.7}$$

At this point, it is sufficient to only scale the C_T vertex as it is the only one which has to be stored for the intersection algorithm. This can be done by dividing its components by B_{Tx}:

$$C_{Tx} = \frac{C_{Tx}}{B_{Tx}}$$
$$C_{Ty} = \frac{C_{Ty}}{B_{Tx}}$$

After each of the transformation matrices are known, they can be multiplied to yield the (temporary) SPTM:

$$SPTM = S \cdot R_x \cdot R_y \cdot R_z \cdot T$$

Mirroring.

In the last step, the transformed vertex C_T has to be analyzed. If $C_{Ty} < 0$, we have to mirror the triangle along the x-axis to map C_T into the first quadrant, i.e.:

$$C_{Ty} = C_{Ty} \cdot (-1)$$

For the SPTM, this corresponds to multiplying the first column (except the very first element) with (-1). Let $SPTM_{ij}$ be the element in the ith row and the jth column of the SPTM, and let the row and column indices start from 0, then

$$SPTM_{10} = SPTM_{10} \cdot (-1)$$
$$SPTM_{20} = SPTM_{20} \cdot (-1)$$
$$SPTM_{30} = SPTM_{30} \cdot (-1)$$

Furthermore, if $C_{Tx} < 0.5$, we can speed up the pruning by mirroring it along the line $(0.5, y)$. This means that the distance between C_T and the origin should become the distance between C_T and the point $(1, 0)$ instad. For C_{Tx}, this mirroring corresponds to

$$C_{Tx} = 1 - C_{Tx},$$

and the SPTM needs to be changed like this:

$$SPTM_{03} = SPTM_{03} - 1$$
$$SPTM_{00} = SPTM_{00} \cdot (-1)$$
$$SPTM_{01} = SPTM_{01} \cdot (-1)$$
$$SPTM_{02} = SPTM_{02} \cdot (-1)$$
$$SPTM_{03} = SPTM_{03} \cdot (-1)$$

Now, the SPTM is complete and C has been transformed to the 2D vertex $C_T = C'$. We store the SPTM and C' in the triangle's data structure.

4.8.2.2 Pruning and probability estimation

As we already know the plane intersection point P_I, it can be transformed into the 2D coordinate system using the SPTM, and this is, in fact, the only SPTM transformation that needs to be done during run-time. Let Q be the transformed plane intersection point. C' is the vertex of the triangle which does not belong to its longest edge and has already been transformed by the SPTM. Now, we prune the triangle to determine whether Q lies within the triangle's area.

There exist several ways of pruning a triangle. Ferko and Ferko [FF15] propose a 3-fold pruning approach (below the triangle's longest edge $\overrightarrow{A'B'}$, above $\overrightarrow{A'C'}$, and above $\overrightarrow{C'B'}$). If one of the following inequations holds true, the intersection point lies in the area which is pruned and, therefore, the ray does not intersect the triangle.

$$Q_y < 0 \tag{4.8}$$

$$-Q_x C'_y + Q_y C'_x > 0 \tag{4.9}$$

$$C'_y(Q_x - 1) + Q_y(1 - C'_x) > 0 \tag{4.10}$$

Note that the inequations 4.9 and 4.10 can be derived from the Plücker-based signed-area functions that are also employed by Woop et al. [WBW13] and Pharr et al. [PJH16c]. Ferko and Ferko further mention that for obtuse triangles it can be feasible to use a 4-fold pruning approach by adding a fourth pruning test which prunes above the C' point [FF15]:

$$Q_y > C'_y \tag{4.11}$$

Ferko and Ferko also describe how the pruning can be optimized using probability estimation. Probability estimation refers to assuming a normal distribution of ray intersection points and calculating the respective probabilities with which a ray would hit certain areas of the triangle's bounding circle. The calculated probabilities are used to change the order of pruning tests so that the areas with higher probability where a ray could miss the triangle are checked earlier. This allows for early terminations. For 3-fold pruning, the areas the circle is subdivided in are: the triangle itself, below the triangle, above the $A'C'$ line, and above the $C'B'$ line (see Figure 4.12). For 4-fold pruning, another area above the C' vertex is considered, most likely reducing the areas above the $A'C'$ line and above the $C'B'$ line (see Figure 4.13).

The probability of a ray hitting an area is the ratio of the area size to the area of the triangle's bounding circle. Note that this is only the case if the algorithm can guarantee that the ray will hit the bounding circle. Thus, in order to calculate reasonable probabilities, a bounding sphere test or bounding circle test has to make sure that the plane intersection point indeed lies within the triangle's bounding circle. Furthermore, it shall be noted that the bounding circle which results from the AABB's bounding sphere is not sufficient as it may not be the tightest fit for the triangle (see also Section 4.4).

Still, for the case that the additional bounding circle test takes too much time, we implemented pruning variants that simply assume the ray did pass a bounding circle test,

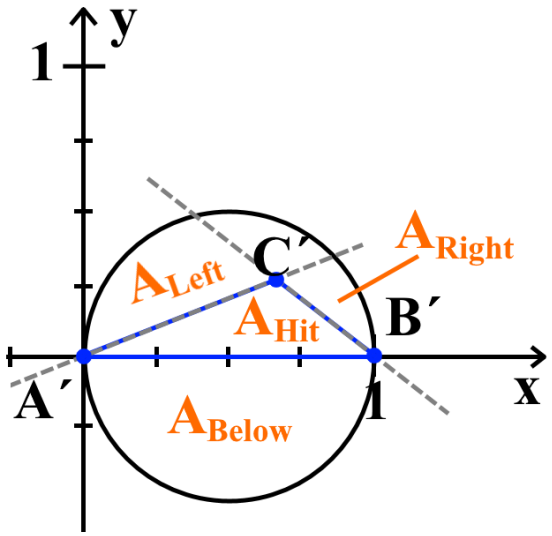


Figure 4.12: Four areas the bounding circle of the triangle $A'B'C'$ can be subdivided in using 3-fold pruning: A_{Hit}, which is the triangle area itself and the area where a ray should hit to pass the pruning tests, and three pruning areas A_{Below} (below the triangle's $A'B'$ line), A_{Left} (above the $A'C'$ line), and A_{Right} (above the $C'B'$ line). It is reasonable to include the area above C' (which is above the $A'C'$ line as well as above the $C'B'$ line) into A_{Left} since the left pruning test is computationally cheaper than the right pruning test.

and calculate the probabilities accordingly. These variants assume that the probabilities are "accurate enough" to produce fast results. The next sections describe the mathematical steps we found for calculating the probability values.

Probability estimation for 3-fold pruning.

The probabilities are calculated as ratios of the respective areas to the triangle's bounding circle's area [FF15]. For obtuse and right triangles, the diameter of this circle is exactly 1 and corresponds to the longest edge $\overrightarrow{A'B'}$ which is mapped to the x-axis in 2D (see also Section 4.4). According to Thales' Theorem, the right-angled corner of a right triangle can be represented by a point on a circle where the circle's diameter is the triangle's hypotenuse. Thus, for right triangles, the vertex C' lies exactly on this circle. For obtuse triangles, it lies within the circle. This reduces the complexity of probability calculation. For example, half of the circle lies below the x-axis, so the lower pruning area is always assumed to be hit with the probability of 0.5 (see Figure 4.12). For acute triangles, the longest edge $\overrightarrow{A'B'}$ does not correspond to the diameter of the bounding circle, as can be seen in Figure 4.14. In this case, a different approach to calculate the areas has to be taken. This means we have to distinguish between obtuse or right triangles and acute triangles, and, if necessary, transform the bounding circle's center point. A triangle $A'B'C'$, which has been transformed with the previously described SPTM, is

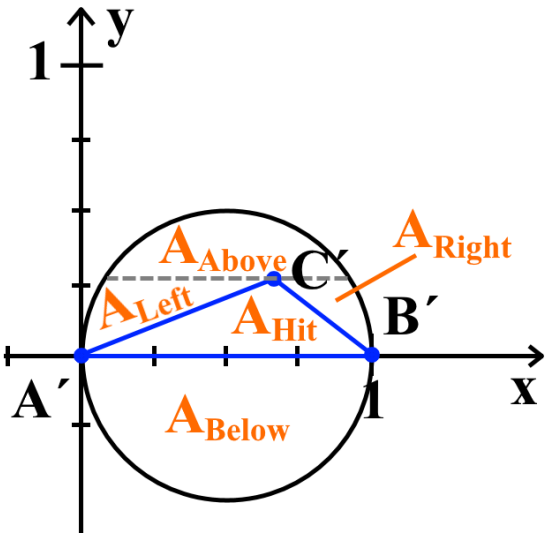


Figure 4.13: Five areas the bounding circle of triangle $A'B'C'$ can be subdivided in using 4-fold pruning. In addition to the pruning areas of 3-fold pruning, A_{Above} marks the area above C'. Naturally, A_{Left} and A_{Right} are smaller than when using 3-fold pruning for the same triangle.

obtuse if $C'_y < 0.5$, right if $C'_y = 0.5$, and acute if $C'_y > 0.5$. We calculate the respective probabilities for the circle sections:

1. P_{Hit}: the probability that normally distributed rays will hit the triangle
2. P_{Below}: the probability that normally distributed rays will pass through the area below the triangle (the lower pruning area)
3. P_{Left}: the probability that normally distributed rays will pass through the area above the $A'C'$ line (the left pruning area)
4. P_{Right}: the probability that normally distributed rays will pass through any other circle area, i.e., through the area above the $C'B'$ line (the right pruning area)

Probabilities for obtuse and right triangles.

For an obtuse or right triangle, one can be sure that the radius r of the triangle's bounding circle is 0.5 and that the circle's diameter lies on the x-axis. Therefore, we do not have to transform the 3D bounding circle center point, nor do we have to recalculate the radius for the transformed circle. Furthermore, as the circle is halved by the x-axis and the triangle does not extend below the x-axis, P_{Below} is 0.5.

The triangle's area is the length of one of its edges multiplied by the height of the third vertex with respect to this edge, divided by two. The $\overrightarrow{A'B'}$ edge's length is 1 and the

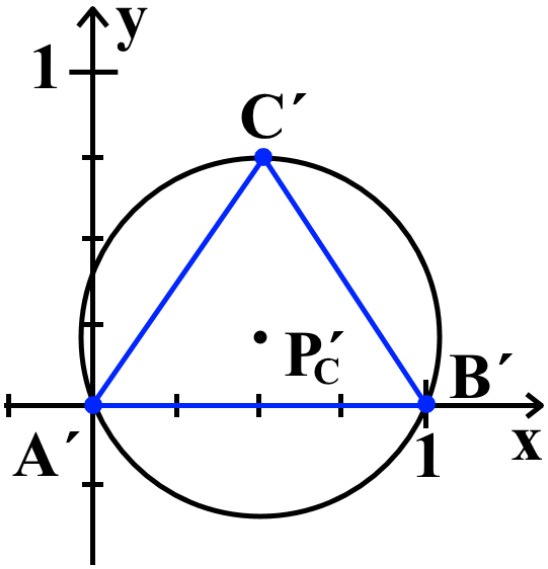


Figure 4.14: For an SPTM-transformed acute triangle $A'B'C'$, the bounding circle's center point $P_C{}'$ lies above the x-axis.

height is C'_y. Thus, the triangle's area A_{Hit} is:

$$A_{Hit} = \frac{1 \cdot C'_y}{2} = 0.5C'_y$$

P_{Hit} is the ratio of A_{Hit} to the circle's area A_O. With A_O being $r^2\pi$ and r being 0.5:

$$P_{Hit} = \frac{0.5C'_y}{0.5^2\pi} = 2\frac{C'_y}{\pi}$$

To calculate P_{Left}, we will take a look at the area that is "cut away" by the left pruning test. As can be seen in Figure 4.15, it is the area of the circle segment A_{Seg}. The angle β is the segment's central angle. Using the law of cosines, we get b (the length of the circle segment, which, as can be seen, exceeds the edge $\overrightarrow{A'C'}$):

$$a^2 = b^2 + c^2 - 2bc \cdot cos(\alpha)$$

Rearranging terms, we get

$$b^2 = a^2 - c^2 + 2bc \cdot cos(\alpha), \tag{4.12}$$

whereby α is the angle spanned by $\overrightarrow{A'C'}$ and the x-axis. Thus, $cos(\alpha)$ is the dot product

$$cos(\alpha) = \overrightarrow{A'C'} \cdot \vec{X}$$

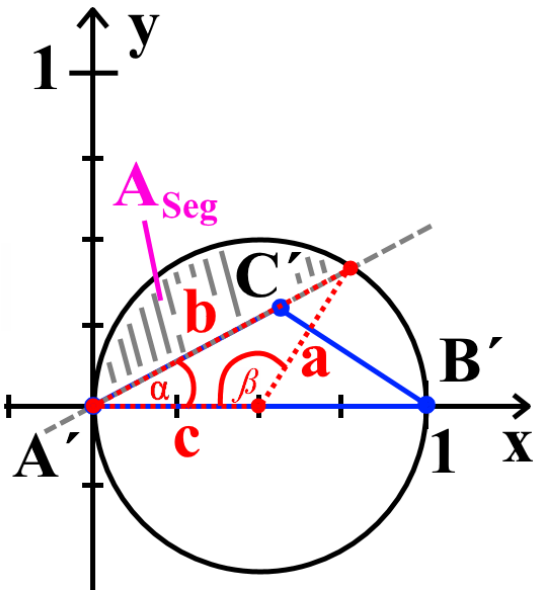


Figure 4.15: Calculating P_{Left}. Left pruning will clip off the hatched part of the circle, i.e. the area A_{Seg}. a, b, and c form another triangle. a and c are spanning β, the central angle of the circle segment. α is the angle spanned by $\overrightarrow{A'C'}$ and the x-axis.

Since A' is at the origin, $\overrightarrow{A'C'}$ can be seen as $\vec{C'}$. $\vec{C'}$ has to be normalized, and because the (normalized) vector $\vec{X}$ is $(1, 0)$, the equation simplifies to:

$$cos(\alpha) = \frac{\vec{C'}_x}{|C'|}$$

We insert this into Equation 4.12. Furthermore, a and c correspond to the circle's radius r, i.e., $a = c = 0.5$ (a and c represent the distance to the second and first intersection of the $A'C'$ line with the circle, respectively). This yields

$$b^2 = 0.5^2 - 0.5^2 + 2 \cdot b \cdot 0.5 \frac{\vec{C'}_x}{|\vec{C'}|},$$

which simplifies to

$$b = \frac{\vec{C'}_x}{|\vec{C'}|}.$$

b can now be used to calculate the central angle β. Again, we use the law of cosines, this time coming from b^2:

$$b^2 = a^2 + c^2 - 2ac \cdot cos(\beta)$$

Substituting a, b, and c yields:

$$\frac{\vec{C'}_x^2}{|\vec{C'}|^2} = 0.5^2 + 0.5^2 - 2 \cdot 0.5 \cdot 0.5 \cdot cos(\beta)$$

$$\frac{\overrightarrow{C'^2_x}}{|\overrightarrow{C'}|^2} = 0.5 - 0.5 \cdot cos(\beta)$$

$$\frac{\overrightarrow{C'^2_x}}{|\overrightarrow{C'}|^2} = 0.5(1 - cos(\beta))$$

We now rearrange for $cos(\beta)$:

$$cos(\beta) = 1 - 2\frac{\overrightarrow{C'^2_x}}{|\overrightarrow{C'}|^2},$$

whereby $|\overrightarrow{C'}|^2$ spares us extracting the square root of the vector length calculation. β is:

$$\beta = arcos(1 - \frac{2\overrightarrow{C'^2_x}}{|\overrightarrow{C'}|^2})$$

Now, we can use β to calculate A_{Seg}. A circle segment's area can be calculated as follows:

$$A_{Seg} = \frac{r^2}{2} \cdot (\beta - sin(\beta)) = 0.125(\beta - sin(\beta)),$$

if β is in radians. Using trigonometric C++ library functions which deliver their results in radians, we do not have to transform β in any way. Subsequently, P_{Left} is:

$$P_{Left} = \frac{A_{Seg}}{A_O} = \frac{A_{Seg}}{r^2\pi}$$

Now, P_{Right} can be calculated as follows:

$$P_{Right} = 1 - P_{Below} - P_{Hit} - P_{Left} = 1 - 0.5 - P_{Hit} - P_{Left}$$

According to the probabilities P_{Below}, P_{Left}, and P_{Right}, the order of pruning tests is changed. The pruning tests itself are the same as described above.

Probabilities for acute triangles.

We can test whether a triangle is acute by checking whether C'_y is greater than 0.5. As noted earlier, probabilities for an acute triangle cannot be calculated using a circle with its diameter lying on the x-axis and with a radius of 0.5. In this case, C', which marks the acute angle of the triangle, would exceed the circle and the calculations used for obtuse and right triangles would be incorrect. Instead, we use the bounding circle described in Section 4.4. With the bounding circle's diameter not being the longest edge of the triangle, each triangle vertex lies on the circle. This means, pruning below will no longer clip off exactly the lower half of the circle. The bounding circle's diameter lies above the x-axis (see Figure 4.14). Therefore, we also view the lower pruning area as a circle segment.

While the bounding circle's area is constant for all obtuse and right triangles ($A_O = 0.5^2\pi = 0.25\pi$), acute triangles each have a different circle area. We transform the circle's center point P_C with the SPTM to get the point $P_C{}'$. Here, we notice that for an SPTM-transformed triangle, the x-component of the bounding circle's center point will always become 0.5:

$$P_C{}'_x = 0.5$$

The radius r of the transformed, 2D circle has to be calculated by either measuring the distance from A' to $P_C{}'$, which is

$$r = \sqrt{P_C{}'_x{}^2 + P_C{}'_y{}^2},$$

or by scaling the original radius R by $\frac{1}{B_{Tx}}$, which has been our scaling value for the SPTM:

$$r = \frac{R}{B_{Tx}}$$

This is the radius r we use in the circle's area formula:

$$A_O = r^2\pi$$

The probability of normally distributed rays hitting the triangle is, again, the triangle area divided by A_O. The triangle area is $\frac{C'_y}{2}$. Thus,

$$P_{Hit} = \frac{C'_y}{2A_O} = \frac{0.5C'_y}{A_O}.$$

We calculate the area of the left circle segment as depicted in Figure 4.16 to get P_{Left}. We can make use of the fact that A' lies at the origin. We already know the circle's transformed center point $P_C{}'$ and the radius r. This time, we use a different formula for the circle segment area for which we only need the radius and the height of the segment:

$$A_{Seg1} = r^2 arcos(\frac{r - h_1}{r}) - (r - h_1)\sqrt{2rh_1 - h_1^2} \tag{4.13}$$

The height h_1 is the length of the edge normal starting exactly between A' and C' and connecting to the circle. Let this starting point be H_1:

$$H_1 = (\frac{C'_x}{2}, \frac{C'_y}{2})$$

Thus, the length $r - h_1$ is the distance between H_1 and $P_C{}'$:

$$r - h_1 = \sqrt{(H_{1x} - P_C{}'_x)^2 + (H_{1y} - P_C{}'_y)^2}$$

$$r - h_1 = \sqrt{(\frac{C'_x}{2} - P_C{}'_x)^2 + (\frac{C'_y}{2} - P_C{}'_y)^2},$$

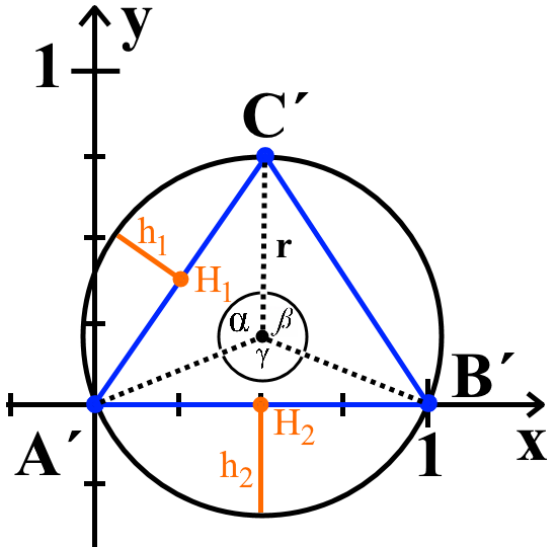


Figure 4.16: The triangle area and three circle segments form the area of the acute triangle's bounding circle. h_1 is the height of the left segment, measured from H_1. h_2 is the height of the lower segment, measured from H_2.

i.e.,

$$h_1 = r - \sqrt{(\frac{C'_x}{2} - P_{C'_x})^2 + (\frac{C'_y}{2} - P_{C'_y})^2}$$

With this, the area of the left circle segment can be calculated by substituting into Equation 4.13, and the corresponding probability is:

$$P_{Left} = \frac{A_{Seg1}}{A_O}$$

$$P_{Left} = \frac{r^2 arcos(\frac{r-h_1}{r}) - (r-h_1)\sqrt{2rh_1 - h_1^2}}{A_O}$$

To benefit from A' lying at the origin, we calculate P_{Below} next. As can be seen in Figure 4.16, the height h_2 originates between A' and B' at the point H_2. Since $A' = (0/0)$ and $B' = (1/0)$, this is the point $(0.5/0)$, so we calculate $r - h_2$ and h_2 as follows:

$$r - h_2 = \sqrt{(0.5 - P_{C'_x})^2 + {P_{C'_y}}^2}$$

$$h_2 = r - (r - h_2)$$

We calculate the circle segment area and its hit probability:

$$A_{Seg2} = r^2 arcos(\frac{r-h_2}{r}) - (r-h_2)\sqrt{2rh_2 - h_2^2}$$

$$P_{Below} = \frac{A_{Seg2}}{A_O}$$

Finally, we calculate

$$P_{Right} = 1 - P_{Hit} - P_{Left} - P_{Below}.$$

Probability estimation for 4-fold pruning.

In addition to the pruning tests used for 3-fold pruning, with 4-fold pruning, the triangle will be pruned above C'. The pruning line can be represented by (x, C'_y). Again, we have to distinguish between obtuse or right triangles and acute triangles. Ferko and Ferko state that 4-fold pruning is only reasonable for obtuse triangles [FF15]. Although it is possible to prune above the (x, C'_y) line for acute triangles using a bounding circle with vertex-equidistant center point, the pruned area can become considerably small as C' will always lie on the circle (see Figure 4.17). Thus, for acute triangles, the pruning is reduced to 3-fold pruning, as is described above. A triangle is acute if $C'_y > 0.5$.

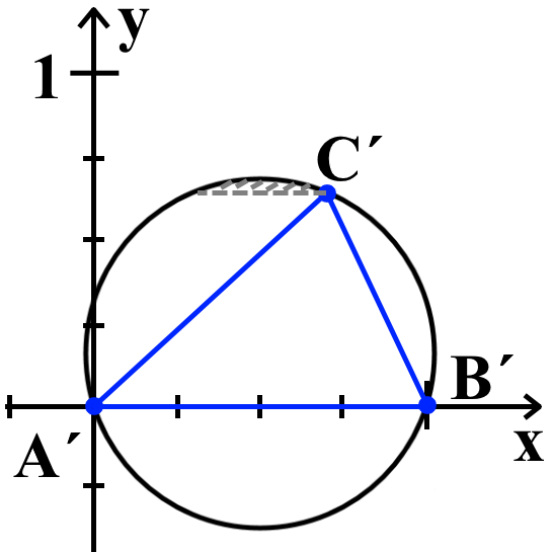


Figure 4.17: Since C' lies on the circle, the upper pruning area (hatched area in the figure) is relatively small for an acute triangle $A'B'C'$.

Pruning obtuse and right triangles.

Consider Figure 4.13 for 4-fold pruning of an obtuse triangle. The longest edge $\overrightarrow{A'B'}$ of obtuse or right triangles corresponds to the diameter of their bounding circle. We do not need to transform the circle's 3D center point or to recalculate its radius, as the radius is known to be 0.5 and the center point lies at $(0.5, 0)$. Like with 3-fold pruning of obtuse or right triangles, half of the circle lies below the x-axis, so $P_{Below} = 0.5$.

We calculate P_{Hit} by dividing the triangle area $\frac{C'_y}{2}$ by the circle area $0.5^2\pi$. This can be simplified to:

$$P_{Hit} = 2\frac{C'_y}{\pi}$$

Pruning above C'_y corresponds to placing a horizontal line through C'_y. The pruning area is the circle segment clipped off by this line. We use the area formula for a circle segment with known radius and segment height:

$$A_{Seg} = r^2 arcos(\frac{r-h}{r}) - (r-h)\sqrt{2rh-h^2}$$

r is 0.5. h is the distance of the horizontal pruning line to the uppermost point of the circle. Trivially, this is

$$h = r - C'_y = 0.5 - C'_y$$

Dividing the segment area by the circle area, we get:

$$P_{Above} = \frac{0.25 \cdot arcos(2C'_y) - C'_y\sqrt{0.25 - {C'_y}^2}}{0.25\pi}$$

$$P_{Above} = 4\frac{0.25 \cdot arcos(2C'_y) - C'_y\sqrt{0.25 - {C'_y}^2}}{\pi}$$

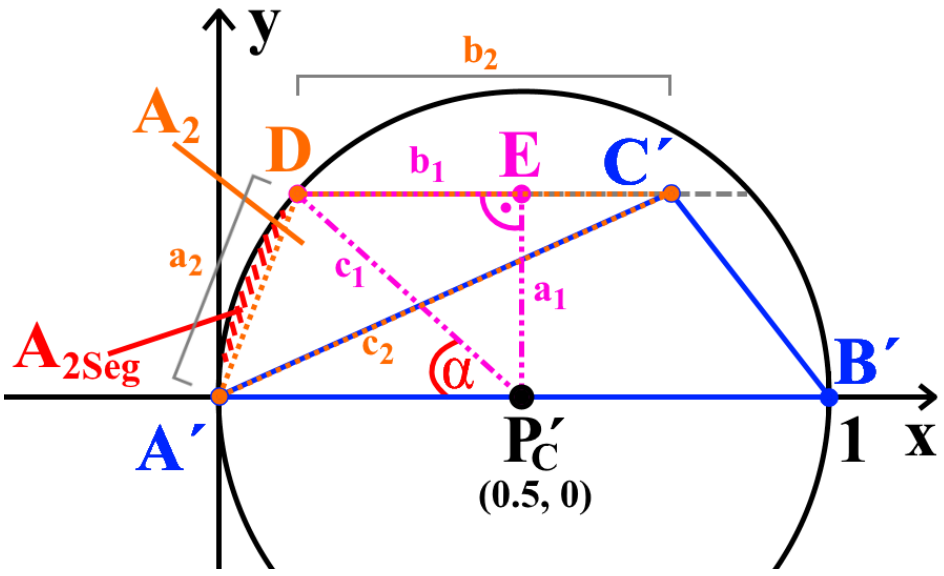


Figure 4.18: Calculating $A_2 + A_{2Seg}$ using two helper triangles $EDP_C{}'$ and $A'C'D$. A_2 is the area of $A'C'D$. A_{2Seg} is the circle segment area (hatched area in the figure). a_1, b_1, and c_1 are the edge lengths of $EDP_C{}'$. a_2, b_2, and c_2 are the edge lengths of $A'C'D$. α is the central angle of the circle segment.

If we take a look at Figure 4.13, we notice that part of the left pruning area (as would be calculated for 3-fold pruning) is now included in the upper pruning area. We need a different approach to calculate P_{Left}. Looking at Figure 4.18, we notice that the left pruning area A_{Left} consists of a triangle $A'C'D$ with area A_2 and another circle segment with area A_{2Seg}. $A'C'D$ shares the vertices A' and C' with the SPTM-transformed triangle. Its third vertex, D, is the (left) point where the horizontal pruning line through C' intersects the circle. We calculate D first. Naturally,

$$D_y = C'_y.$$

As for D_x, if we can calculate the horizontal distance b_1 of D to the circle's center point ${P_C}'$, we can write:

$$D_x = r - b_1 = 0.5 - b_1$$

To calculate b_1, we construct another triangle $EDP_C{}'$, as can be seen in Figure 4.18. $EDP_C{}'$ is a right triangle. One of its edges connects the circle's center point ${P_C}'$ with the point E, which has the center point's x-compoment (which is 0.5) and lies on the horizontal pruning line with y-component C'_y. Its third vertex is D, which has the same y-component. The distance between D and ${P_C}'$ is $c_1 = r$. We can calculate b_1 using the Pythagorean theorem. r is the hypotenuse and $a_1 = C'_y$ is the length of the other leg of the triangle (the distance of ${P_C}'$ to E).

$$b_1 = \sqrt{r^2 - {C'_y}^2} = \sqrt{0.5^2 - {C'_y}^2} = \sqrt{0.25 - {C'_y}^2}$$

We now know both components of D and, therefore, all vertices of $A'C'D$. We calculate the lengths of its edges, a_2, b_2, and c_2. b_2 is the horizontal distance of C' to D:

$$b_2 = C'_x - D_x = C'_x - 0.5 + b_1$$

a_2 is the distance between A' and D. c_2 is the distance between A' and C'. Because A' is at the origin, we can write:

$$a_2 = |D - A'| = \sqrt{D_x^2 + {D_y}^2}$$

$$a_2 = \sqrt{(0.5 - b_1)^2 + {C'_y}^2}$$

$$c_2 = |C' - A'|$$

$$c_2 = \sqrt{{C'_x}^2 + {C'_y}^2}$$

We insert this three lengths in Heron's Formula and calculate the area of $A'C'D$, A_2.

$$s = \frac{a_2 + b_2 + c_2}{2}$$

$$A_2 = \sqrt{s(s - a_2)(s - b_2)(s - c_2)}$$

To calculate A_{2Seg}, we use the formula

$$A_{2Seg} = \frac{r^2}{2} \cdot (\alpha - sin(\alpha)) \tag{4.14}$$

$sin(\alpha)$ can be determined using the cross product of the vector $\overrightarrow{P_C{}'D}$ and the vector starting at $P_C{}'$ and pointing to the origin. Again, it is necessary to normalize both of these vectors. As α opens up to the left, the normalized vector from $P_C{}'$ to the origin is

$$\vec{X} = (-1, 0, 0)$$

The vector $\overrightarrow{P_C{}'D}$ can be described by b_1 and C'_y:

$$\overrightarrow{P_C{}'D} = (D_x - 0.5, D_y, 0)$$

$$\overrightarrow{P_C{}'D} = (0.5 - b_1 - 0.5, C'_y, 0)$$

$$\overrightarrow{P_C{}'D} = (-b_1, C'_y, 0)$$

$\overrightarrow{P_C{}'D}$ can be normalized by dividing its components by

$$P_C{}'D_{Length} = \sqrt{{b_1}^2 + {C'_y}^2}.$$

Now, $sin(\alpha)$ is given by:

$$sin(\alpha) = |(\frac{-b_1}{P_C{}'D_{Length}}, \frac{C'y}{P_C{}'D_{Length}}, 0) \times (-1, 0, 0)|$$

$$sin(\alpha) = \frac{C'_y}{P_C{}'D_{Length}}$$

Using trigonometric C++ functions that work with radians, we can substitute into Equation 4.14 and calculate:

$$A_{2Seg} = 0.125(\alpha - sin(\alpha))$$

The left pruning area A_{Left} is the sum of A_2 and A_{2Seg}:

$$A_{Left} = A_2 + A_{2Seg}$$

$$A_{Left} = \sqrt{s(s - a_2)(s - b_2)(s - c_2)} + 0.125(\alpha - sin(\alpha))$$

Dividing by A_O yields P_{Left}:

$$P_{Left} = \frac{A_{Left}}{A_O}$$

Finally, we get P_{Right} by subtracting all other probabilities from 1:

$$P_{Right} = 1 - P_{Below} - P_{Hit} - P_{Above} - P_{Left}$$

Pruning test	Operations	Weight
Below	1	11
Left	5	7
Right	6	6

(a)

Pruning test	Operations	Weight
Below	1	12
Above	1	12
Left	5	8
Right	6	7

(b)

Table 4.1: Defining the weights for 3-fold (a) and 4-fold (b) pruning tests. A pruning test's weight is defined by the sum of operations (comparisons, additions, subtractions, and multiplications) of all other tests.

Pruning acute triangles.

In the case of acute triangles, pruning above C'_y will be skipped. We assume that $P_{Above} = 0$. The algorithm resorts to 3-fold pruning for acute triangles as described earlier. This means that we need to transform the 3D bounding circle's center point with the SPTM and calculate the radius in 2D.

Probability estimation using weights.

In addition to the pruning variants proposed by Ferko and Ferko, we derived a modification of the probability estimation tests. As the three (or four) pruning tests consist of different numbers of operations, it may be beneficial to add weights to the probabilities according to how computationally expensive a pruning test is. For example, the above test consists of only one comparison, while the right test consists of one comparison, two multiplications, and three additions/subtractions. Thus, for a slightly higher probability for the right test, it may still be feasible to do the above test first.

We define a pruning test's weight as the sum of operations (comparisons, additions, subtractions, and multiplications) of all other pruning tests. The number of operations for 3-fold and 4-fold pruning tests, respectively, and the resulting weights can be seen in Table 4.1. During pre-processing, the calculated probabilities are multiplied by the respective weights before the order of the pruning tests is determined.

With these considerations in mind, the possible variants of pruning algorithms are:

- 3-fold pruning
- 3-fold pruning with probability estimation
- 3-fold pruning with weighted probability estimation
- 4-fold pruning
- 4-fold pruning with probability estimation
- 4-fold pruning with weighted probability estimation

Probability estimation with disregarded bounding circle test.

If executing the bounding circle test for each triangle where a 3D ray-plane intersection point has been found is slowing down the algorithm too much, there is the possibility to just skip the bounding circle test and still do probability estimation as described above. The idea behind this is to do some "quick and dirty" pruning tests where the time consumed by the bounding circle test is saved. The calculated probabilities will not reflect correct probabilities for uniformly distributed rays across the circle area, since the algorithm will also prune for rays which do not hit the circle. However, they may be sufficiently accurate for reordering the pruning tests in a favorable way.

4.9 Optimizing the algorithm

After implementing the algorithm, we try to optimize it by reordering operations, omitting tests, and adding new tests. More precisely, we evaluate the algorithm's performance using both a bounding sphere and bounding box test (as described above), using only the bounding sphere or bounding box test, respectively, or using no bounding volume test at all. Concerning the bounding sphere test, we implement the parametric approach as well as the dot product based approach (see Section 4.6). We evaluate a bounding circle test as a further possible early termination strategy. If the calculated ray-plane intersection point does not lie within the triangle's bounding circle, we reject the triangle.

During ray-plane intersection, a t-test checks whether an intersection point lies within a valid range along the ray. Other possible optimizations include delaying or omitting the t-test. Whenever possible, we delay operations to benefit from early-exit points. We consider several ways of constructing and using the transformation matrix in the unit triangle approach and choose the most efficient one. For the pruning approach, we define eight possible pruning variants. We evaluate 3-fold pruning and 4-fold pruning. We use probability estimation and reorder the pruning tests accordingly to exit early if the ray-plane intersection point does not lie inside of the triangle. We further use weights to address the fact that some pruning tests are computationally cheaper than others. We also evaluate pruning variants with by-passed bounding circle tests. We implement a hybrid algorithm that uses both the unit triangle approach and the pruning approach. Ultimately, we evaluate the optimized unit triangle algorithm and the hybrid algorithm against two other state-of-the-art ray-triangle intersection algorithms.

4.10 Language of and modifications in the original framework

The algorithm is implemented into PBRT, version 3, which has been developed by Pharr et al. [PJH16d]. The source code is written in C++. Our implementation focusses on changing the code in PBRT's *Triangle* class. More precisely, we expand its set of member variables and its constructor to implement additional pre-processing, and replace the

default ray-triangle intersection algorithms in its methods *Intersect()* and *IntersectP()*. We add our own class *BoundingSphere* to contain minimal information and a ray-sphere intersection procedure for a triangle's bounding sphere.

4.11 Analysis methods

We evaluate our results in two different ways. The first is to measure the time the new algorithm consumes for different test scenes. We compare this to the time consumed by other algorithms that achieve results with comparable image quality. We use the same scenes with the same scene descriptions for every algorithm. This also indicates how long each algorithm needs to process roughly the same number of intersections. To achieve robust results, we average the times over five runs.

The second evaluation is based on image quality measurement. We let the different algorithms process the same scenes within a given time. Then, we compare the quality of the resulting images. We do this by calculating the mean squared error between the resulting images and a high-quality ground truth image of the same scene.

CHAPTER 5

Implementation

We implemented a new ray-triangle intersection algorithm into Pharr et al.'s [PJH16d] physically based ray tracing framework PBRT. The algorithm is mostly based on Ferko and Ferko's method [FF15] and incorporates ideas from Baldwin and Weber's algorithm [BW16] (see Section 5.5). We implemented different ways of constructing the bounding sphere and the 3D-to-2D transformation matrices. In addition to Ferko and Ferko's proposed bounding volume tests, we implemented a test of a potential intersection point against the bounding circle of the triangle. This chapter presents the implementational details. More precisely, it describes the implementation of the ray-plane intersection test (Section 5.1), the bounding box test (Section 5.2), the ray-sphere intersection test (Section 5.3), the bounding circle test (Section 5.4), the 3D-to-2D transformation (Section 5.5), the pruning tests (Section 5.6), and the probability estimation for the pruning approach (Section 5.7).

5.1 Ray-plane intersection test

To find a point in the ray which fulfills the plane equation, the plane's normal (which is the triangle's normal) has to be known, so we calculate the cross product of two of the triangle's edges:

$$\vec{n} = \overrightarrow{AB} \times \overrightarrow{AC}$$

The normal $\vec{n}$ is precomputed and stored for each triangle. In the intersection routine, two ray termination possibilities take place during the ray-plane intersection test: one when evaluating the dot product of the normal and the ray direction (when the dot product is 0, the ray and plane are parallel; however, it may be negative), and one when evaluating the parametric distance t along the ray, which has to be positive. Instead of comparing these values to 0, small epsilon values may be used. For example, for t it is of importance to choose a suitable epsilon value to prevent artifacts from appearing in the rendered scene. We found that $\varepsilon = 0.00001$ is a reasonable epsilon value for the t test.

When evaluating the dot product, comparing to 0 provided satisfying results. Also, no *abs()* function has to be called if the dot product is compared to 0.

5.2 Bounding box test

PBRT uses the predefined class *Bounds3f* and the *Shape* class' methods *WorldBound()* and *ObjectBound()* to define the bounding box of primitives (shapes) in world space and object space, respectively. Pharr et al. use axis-aligned bounding boxes, which makes the computation of ray-box intersections (see Section 4.5) easier. They construct the bounding box of a triangle as the AABB spanning from the vertex with minimal triangle coordinates to the vertex with maximal triangle coordinates [PJH16d].

Ferko and Ferko's algorithm is designed to check whether the plane intersection point lies within the bounding box and terminate if it does not [FF15]. However, PBRT employs acceleration structures to avoid unnecessary intersection tests. One of them is the Bounding Volume Hierarchy (BVH) accelerator, which tests rays against the bounding boxes of the scene primitives [PJH16d]. As the intersection point can only lie within the bounding box if the bounding box is intersected by the ray, Ferko and Ferko's bounding box test need not be explicitly performed again. Nevertheless, we implemented bounding box tests into an early version of the algorithm to see how they affect performance. As they indeed slowed down the algorithm, we excluded bounding box tests from the final algorithm.

5.3 Ray-sphere intersection test

PBRT's *Bounds3f* class not only stores an AABB, but can also calculate the AABB's bounding sphere [PJH16b]. As PBRT's BVH accelerator intersects the ray with the scene primitives' AABB, there is no need to test the ray against the AABB's bounding sphere in the intersection routine. One can, however, use a tighter bounding sphere that only encompasses the triangle and not its bounding box. To achieve this, we constructed the bounding sphere as described in Section 4.4. Depending on the triangle's orientation, this bounding sphere may cover space which is not covered by the bounding box and vice versa, so there may be more early rejects.

PBRT provides classes for primitive shapes which can be used for representing a scene. One of these shapes is the sphere, as defined in the *Sphere* class. These shapes have ray-intersection routines already implemented, so to enable intersection tests for rays and a triangle's bounding sphere, a straightforward approach would be to

1. determine the bounding sphere's center and radius,
2. create a PBRT-native *Sphere* object with this center and radius and store it within the triangle during pre-processing, and

3. when testing for ray-triangle intersection, use the predefined intersection method and reject the triangle if this intersection method returns false (no intersection).

However, PBRT's *Sphere* class allows for more complex configurations than a simple bounding sphere needs for the sole purpose of intersection testing. For example, the *Sphere* class stores additional parameters like an alpha texture, which allows parts of the sphere to be "invisible", thus reporting a miss even if being hit by the ray. This makes sense for usage of spheres in the scene which aim to represent objects with holes. A bounding sphere, however, has no need for alpha texture. PBRT also enables the computation of partial spheres, which is also not necessary for bounding spheres. Thus, the *Sphere* class stores data which is of no relevance to a bounding sphere, and although its *IntersectP()* method does only compute whether there is an intersection or not and does no shading whatsoever, it still executes lines of code irrelevant to bounding sphere intersection. Hence, we wrote an own class *BoundingSphere* which only stores the information the bounding sphere needs: the sphere's center and the radius. Depending on the type of later calculations, the class may also store the squared radius instead of the radius to avoid squaring at run-time.

Following PBRT's naming convention, the *BoundingSphere* class features its own *IntersectP()* method, which simply takes a ray as parameter and returns true, if the ray intersects the sphere, and false, if it does not. Two variants of the intersection method have been implemented: an intersection test in the sphere's object space, and a dot product based intersection test, both of which are described in Section 4.6. Both approaches can save the squaring of the sphere's radius if, instead of the radius, the squared radius is stored for each bounding sphere.

5.4 Bounding circle test

As bounding shapes should be as small as possible, we implemented a bounding circle test in addition to the bounding sphere test. This test takes advantage of the fact that circles as well as triangles are planar. The bounding circle is not tested against the ray. Instead, after the plane intersection point P_I is known, the intersection routine tests whether P_I lies inside of the bounding circle, and rejects the triangle if it does not.

We first calculate the squared distance between the plane intersection point and the bounding circle's center point. If the distance is greater than the squared radius of the bounding circle, we reject the triangle. If a bounding sphere has been constructed for the triangle, the bounding circle's center point and radius are the same as the sphere's center point and radius. By reusing these values, no additional storage is required for each triangle.

5.5 3D-to-2D transformation

We implemented both of Ferko and Ferko's [FF15] approaches (see Section 4.1), using a unit triangle transformation matrix (UTM) for the unit triangle approach and a similarity preserving transformation matrix (SPTM) for the pruning approach. Remember that for computing the UTM, the inverse of a matrix A has to be known, where A's column vectors correspond to the triangle's vertices (see Section 4.8.1). There are different ways of precomputing the inverse and, subsequently, the UTM, some of which enable optimized transformation in the intersection routine and therefore lead to small speed-ups:

1. PBRT provides classes for transformation matrices and their inverses, so one possibility is to create the target matrix A as *Matrix4x4* object and calling *Matrix4x4*'s method *Inverse()*. The resulting *Matrix4x4* object can be stored in a *Transform* object, which can be used to transform a point or vector by using its *operator()* function.

2. For 3D-to-2D transformation, only the first two rows of the inverse are needed. 1. can be optimized by not using the *operator()* method during run-time, but selectively picking only the matrix elements needed to transform points. The matrix array can be accessed via the *Transform* method *GetMatrix()*.

3. One pre-processing step can be omitted by making use of the fact that for every transformation, PBRT already calculates the inverse and stores it within the *Transform* object. Thus, it is sufficient to just create the target matrix A as a *Transform* object. The matrix can then be accessed using the *GetInverseMatrix()* method.

4. To avoid inverting the target matrix explicitly during pre-processing, the inverse can directly be formulated in code. Baldwin and Weber [BW16] note that a unit triangle can be transformed to any triangle $v_1v_2v_3$ by using the transformation matrix

$$T = \begin{pmatrix} v_{2x} - v_{1x} & v_{3x} - v_{1x} & a & v_{1x} \\ v_{2y} - v_{1y} & v_{3y} - v_{1y} & b & v_{1y} \\ v_{2z} - v_{1z} & v_{3z} - v_{1z} & c & v_{1z} \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{5.1}$$

   where (a, b, c) is a so-called free vector that does not affect the transformation. Thus, a, b, and c can be selected in a way that simplifies later multiplication of points with the inverse matrix. Baldwin and Weber set one of these free vector elements to 1 and the others to 0. They found that selecting the highest-magnitude component of the triangle's normal as the non-zero component of the free vector leads to numerically more stable results. Let $\overrightarrow{E_1}$ be $(v_2 - v_1)$ and let $\overrightarrow{E_2}$ be $(v_3 - v_1)$. $\vec{n}$ is the triangle plane's normal vector. The inverse transformation depends on

which component of the free vector was selected to be the non-zero component. For $a = 1$, the inverse (which is, in Baldwin and Weber's case, the UTM) is

$$UTM = \begin{pmatrix} 0 & \frac{\overrightarrow{E_{2z}}}{\overrightarrow{n}_x} & -\frac{\overrightarrow{E_{2y}}}{\overrightarrow{n}_x} & \frac{(v_3 \times v_1)_x}{\overrightarrow{n}_x} \\ 0 & -\frac{\overrightarrow{E_{1z}}}{\overrightarrow{n}_x} & \frac{\overrightarrow{E_{1y}}}{\overrightarrow{n}_x} & -\frac{(v_2 \times v_1)_x}{\overrightarrow{n}_x} \\ 1 & \frac{\overrightarrow{n}_y}{\overrightarrow{n}_x} & \frac{\overrightarrow{n}_z}{\overrightarrow{n}_x} & -\frac{\overrightarrow{n} \cdot v_1}{\overrightarrow{n}_x} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{5.2}$$

For $b = 1$, the inverse is

$$UTM = \begin{pmatrix} -\frac{\overrightarrow{E_{2z}}}{\overrightarrow{n}_y} & 0 & \frac{\overrightarrow{E_{2x}}}{\overrightarrow{n}_y} & \frac{(v_3 \times v_1)_y}{\overrightarrow{n}_y} \\ \frac{\overrightarrow{E_{1z}}}{\overrightarrow{n}_y} & 0 & -\frac{\overrightarrow{E_{1x}}}{\overrightarrow{n}_y} & -\frac{(v_2 \times v_1)_y}{\overrightarrow{n}_y} \\ \frac{\overrightarrow{n}_x}{\overrightarrow{n}_y} & 1 & \frac{\overrightarrow{n}_z}{\overrightarrow{n}_y} & -\frac{\overrightarrow{n} \cdot v_1}{\overrightarrow{n}_y} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{5.3}$$

For $c = 1$, the inverse is

$$UTM = \begin{pmatrix} \frac{\overrightarrow{E_{2y}}}{\overrightarrow{n}_z} & -\frac{\overrightarrow{E_{2x}}}{\overrightarrow{n}_z} & 0 & \frac{(v_3 \times v_1)_z}{\overrightarrow{n}_z} \\ -\frac{\overrightarrow{E_{1y}}}{\overrightarrow{n}_z} & \frac{\overrightarrow{E_{1x}}}{\overrightarrow{n}_z} & 0 & -\frac{(v_2 \times v_1)_z}{\overrightarrow{n}_z} \\ \frac{\overrightarrow{n}_x}{\overrightarrow{n}_z} & \frac{\overrightarrow{n}_y}{\overrightarrow{n}_z} & 1 & -\frac{\overrightarrow{n} \cdot v_1}{\overrightarrow{n}_z} \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ [BW16]}. \tag{5.4}$$

5. Baldwin and Weber further optimized their algorithm. As their inverses each contain one column $(0, 0, 1, 0)$, they only store all other columns and the index of the column which contains the $(0, 0, 1, 0)$ vector. Points can be transformed faster because it is known which values will be multiplied by 0 or 1, and these operations can be skipped or simplified [BW16]. This saves two multiplications and two additions in each call of the intersection method.

6. Another optimization reduces the memory needed for the triangles' precomputed data. Any point transformed with the UTM will have its z-component eliminated. Thus, we only need the rows of the UTM responsible for transformation of the x- and y-components. Only the first two rows of the UTM have to be stored.

The implementation of the SPTM is based on the considerations described in Section 4.8.2.1. These do not require any PBRT-specific classes or methods, but translate directly into C++ code.

5.6 Pruning tests

The 3-fold pruning described in Section 4.8.2.2 translates into code in a straight-forward way. *sptmCx* and *sptmCy* are floating point variables storing C'_x and C'_y, respectively. The pruning tests are executed in an order so that the ones with fewer calculations are executed first, i.e., pruning below, then pruning left, then pruning right (see Algorithm 5.1).

Algorithm 5.1: 3-fold pruning tests.

```
1 if Qy < 0 then
2 |   return;
3 end
4 if -Qx*sptmCy + Qy*sptmCx > 0 then
5 |   return;
6 end
7 if sptmCy*(Qx - 1) + Qy*(1 - sptmCx) > 0 then
8 |   return;
9 end
```

4-fold pruning uses the above-test as an additional test. When using probability estimation for obtuse and right triangles, a ray is assumed to hit the pruning area below the x-axis with a probability of 50%. Thus, the below-test will also be executed first for 4-fold pruning without probabilities. As the above-test also consists of only one comparison, it will be executed second. The order of the other tests does not change (see Algorithm 5.2).

Algorithm 5.2: 4-fold pruning tests.

```
1  if Qy < 0 then
2  |   return;
3  end
4  if Qy > sptmCy then
5  |   return;
6  end
7  if -Qx*sptmCy + Qy*sptmCx > 0 then
8  |   return;
9  end
10 if sptmCy*(Qx - 1) + Qy*(1 - sptmCx) > 0 then
11 |   return;
12 end
```

5.7 Probability estimation for the pruning approach

Estimating hit probabilities in the pruning approach enables us to reorder the pruning tests in a favorable way, leading to a higher number of early terminations. We implemented

the 3-fold pruning approach as well as 4-fold pruning approach. In each approach, we distinguish between obtuse or right triangles and acute triangles. Let C' be the triangle's vertex C after transformation with the SPTM. The triangle is acute if $C'_y > 0.5$.

Translating the probability estimation formulas derived in Section 4.8.2.2 into C++ code, we get the following algorithms. Algorithm 5.3 estimates circle area hit probabilities for 3-fold pruning of obtuse and right triangles. All variables are single-precision floating point variables.

Algorithm 5.4 estimates probabilities for 3-fold pruning of acute triangles. *sptm* is a floating point array storing the SPTM, and *sptm[i][j]* addresses the element of the ith row and the jth column, with indices starting from 0. *center* is the center point of the triangle's bounding circle. *Centerx* and *Centery* represent the transformed center point.

Algorithm 5.5 estimates probabilities for 4-fold pruning of obtuse and right triangles. If we encounter an acute triangle in the 4-fold pruning approach, we resort to 3-fold pruning of acute triangles. In this case, we use Algorithm 5.4 and assume that $pAbove = 0$. No above test will be performed in the intersection routine for this triangle.

We also implemented pruning variants using weighted probabilities. In these variants, we multiply the probabilities by the weights of the respective pruning tests. We order the probabilities *pBelow*, *pLeft*, *pRight*, and (when using 4-fold pruning) *pAbove* from highest to lowest and prune in the same order. The pruning test with the highest hit probability will be executed first, the test with the second highest probability second, and so on. If two pruning tests exhibit the same probability, the computationally cheaper one is executed earlier.

Algorithm 5.3: 3-fold pruning: estimating probabilities for obtuse and right triangles.

```
1 pBelow = 0.5f;
2 pHit = 2.0f * sptmCy / Pi;
3 beta = acos(1.0f - 2.0f * sptmCx*sptmCx / (sptmCx*sptmCx +
    sptmCy*sptmCy));
4 pLeft = 0.5f * (beta - sin(beta)) / Pi;
5 pRight = 1.0f - pBelow - pHit - pLeft;
```

Note that probability calculation happens during pre-processing. The order of the pruning tests is stored within the triangle. In the intersection methods, the algorithm just has to execute the pruning tests in the given order.

Algorithm 5.4: 3-fold pruning: estimating probabilities for acute triangles.

```
1  Centerx = 0.5f;
2  Centery = sptm[1][0] * center.x + sptm[1][1] * center.y + sptm[1][2] * center.z +
     sptm[1][3];
3  r = (float) sqrt(Centerx*Centerx + Centery*Centery);
4  AO = r*r*Pi;
5  pHit = 0.5f*sptmCy / AO;
6  rMinh1 = (float) sqrt((sptmCx / 2.0f - Centerx)*(sptmCx / 2.0f - Centerx) +
     (sptmCy / 2.0f - Centery)*(sptmCy / 2.0f - Centery));
7  h1 = r - rMinh1;
8  pLeft = (float) (r*r*acos(rMinh1 / r) - rMinh1*sqrt(2.0f*r*h1 - h1*h1)) / AO;
9  rMinh2 = (float) sqrt((0.5f - Centerx)*(0.5f - Centerx) + Centery * Centery);
10 h2 = r - rMinh2;
11 pBelow = (float) (r*r*acos(rMinh2 / r) - rMinh2*sqrt(2.0f*r*h2 - h2*h2)) / AO;
12 pRight = 1.0f - pHit - pLeft - pBelow;
```

Algorithm 5.5: 4-fold pruning: estimating probabilities for obtuse and right triangles.

```
1  pBelow = 0.5f;
2  pHit = 2.0f * sptmCy / Pi;
3  pAbove = (float)(4.0f * (0.25f*acos(2.0f * sptmCy) - sptmCy*sqrt(0.25f -
     sptmCy*sptmCy)) / Pi);
4  b1 = (float)sqrt(0.25f - sptmCy*sptmCy);
5  a2 = (float)sqrt(0.25f - b1 + b1*b1 + sptmCy*sptmCy);
6  b2 = sptmCx - 0.5f + b1;
7  c2 = (float)sqrt(sptmCx*sptmCx + sptmCy*sptmCy);
8  s = (a2 + b2 + c2) / 2.0f;
9  CenterDLength = (float)sqrt(b1*b1 + sptmCy*sptmCy);
10 sinAlpha = sptmCy / CenterDLength;
11 pLeft = (float)(4.0f * (0.125f*(asin(sinAlpha) - sinAlpha) + sqrt(s*(s - a2)*(s -
     b2)*(s - c2))) / Pi);
12 pRight = 1.0f - pBelow - pHit - pAbove - pLeft;
```

CHAPTER 6

Evaluation and results

We tested different combinations of intersection tests and modifications of Ferko and Ferko's [FF15] algorithm to increase its performance. Experimenting with different tests and operation orders, we evaluated an optimized version of the algorithm and compared its results to those of two other state-of-the-art algorithms. In this chapter, we first describe the final configuration of the algorithm in Section 6.1. In Section 6.2, we discuss a hybrid algorithm incorporating both the unit triangle approach (or UTM approach, as it uses the unit triangle transformation matrix) and the pruning approach. Section 6.3 presents the results. In Section 6.4, we compare the algorithm to other algorithms proposed in the literature.

6.1 Discovered optimizations

As it became obvious in early tests that the bounding box test does not speed up the intersection algorithm (Section 5.2), we decided to skip it. Furthermore, the bounding sphere test decreased performance as well, although we computed a tight-fitting bounding sphere for obtuse triangles, as described in Section 4.4. To avoid function calls, we moved the ray-sphere intersection code from the *BoundingSphere* class directly into the ray-triangle intersection methods. We also implemented both approaches of ray-sphere intersection tests (Section 4.6), but still did not discover any improvement. Thus, we omitted the bounding sphere test as well. We see PBRT's BVH acceleration structure as a possible reason for the bounding volumes' negative impact on performance. As the BVH already tests rays against the triangles' bounding boxes, additional point-in-box tests are mostly redundant operations. The bounding boxes used in the BVH may not be the tightest fit for the triangles, so an additional test against a tighter bounding box can still lead to terminations. The bounding sphere may cover 3D space not covered by the bounding box and vice versa, providing further opportunities of early terminations.

However, both bounding volume tests do not lead to a sufficient number of terminations to compensate for the cost of the tests.

Instead of the bounding box test and the bounding sphere test, we implemented a bounding circle test, using the center point and radius of the calculated bounding sphere. The bounding circle test is mandatory for the pruning approach with probability estimation, if one desires realistic hit-area probabilities. As a replacement for the bounding box test, we implemented it for the UTM approach as well, executing it after the plane intersection point is known. However, the bounding circle test also slows down the algorithm, albeit not as much as the bounding volume tests. Our early tests showed that the best performance can be achieved by skipping bounding volume tests and the bounding circle test altogether.

The core part of the optimized intersection algorithm works as follows: First, we calculate the plane intersection point. Then, we transform it by using either the unit triangle transformation matrix (UTM) or the similarity preserving transformation matrix (SPTM). Finally, the 2D tests are conducted. For the unit triangle transformation, we used Baldwin and Weber's [BW16] transformation matrix which utilizes the free vector. Thus, we only need to store three columns of the matrix instead of four, as well as a column selector that defines the location of the free vector. This reduces the operations needed during run-time for the transformation of the plane intersection point. It requires execution of additional *if*-statements, but saves two multiplications and two additions per triangle. Furthermore, as a transformed point's z-component will always become 0, storing only the first two rows of the matrix is sufficient.

We explored further possible optimizations. When calculating the plane intersection point, we compute its parametric distance t along the ray. We then make an early termination decision. For each ray, PBRT stores a floating point value *tMax*. This can be thought of as a delimiter for the range of the ray we are interested in (for example, the t value of an already found hit point). If t exceeds *tMax*, we discard the triangle. If $t < 0$ (or $<$ some epsilon, which we found to be more robust), we discard the triangle as well. We experimented with delaying these t-tests, or omitting them altogether. The reason was that the following 2D tests may discard more triangles than the t-tests for cases in which it is rather unlikely that a calculated t exceeds *tMax*. We found that moving the $t < \varepsilon$ test is actually disadvantageous for the performance, and skipping it results in incorrect pixel values. However, delaying the *tMax* test until the 2D tests have passed slightly improves performance. PBRT provides two intersection methods, *Intersect()* and *IntersectP()*. The latter only reports whether an intersection has been found and does not calculate the intersection point. We were able to skip the *tMax* test in the *IntersectP()* method, saving the execution of one *if*-statement, without impacting the final image.

We delayed operations whenever possible to benefit from early terminations. For example, when transforming the plane intersection point with the UTM, we first transform only its x-component. If this is already out of range for a valid intersection, we discard the triangle. Only if the transformed x-component is valid, we transform the y-component as

well. We also follow Bikker's advice and use the *const* keyword whenever possible as this improves performance [Bik07].

6.2 The hybrid algorithm

As the pruning approach does not provide barycentric coordinates which PBRT often needs for further processing, we did not regard this approach as a fully functional algorithm on its own. Instead, we implemented a hybrid algorithm that uses the UTM approach when barycentric coordinates are required, and the pruning approach when they are not required. PBRT's *IntersectP()* method does not need barycentric coordinates. An exception are alpha textures. PBRT provides these textures to simulate, for example, leaves or objects with holes. These are not explicitly modelled, but the texture provides information for where a ray should pass the object, i.e., no intersection should be reported. How often each of the intersect methods is called and whether there is alpha texture present depends on the scene. The hybrid algorithm uses the UTM approach in the *Intersect()* method and the pruning approach in the *IntersectP()* method. To deliver an algorithm for all use cases, we calculate baryentric coordinates in the *IntersectP()* method using the UTM if alpha texture is present, however, we did not include this case in our tests as the hybrid algorithm generally performs slightly worse than the algorithm relying solely on the UTM.

For implementing the hybrid algorithm, we first selected the pruning variant to be used. As several variants are possible (see also Section 4.8.2.2), we tested each of them on scenes with high as well as with low ray-triangle hit-rates while disregarding barycentric coordinates. More precisely, we hardcoded the barycentric coordinates of each hitpoint to be 0.3, 0.3, and 0.4, respectively. Doing so results in incorrect images (e.g., regarding texture information), but for this evaluation we were only interested in which variant would be the fastest for the general point-in-triangle test. This way, we could also observe whether the pruning variants are faster than the UTM approach. Note that using the pruning approach in PBRT only makes sense when combining it with an approach which calculates barycentric coordinates, like we did in our hybrid algorithm.

We evaluated every possible combination of 3-fold pruning and 4-fold pruning, with and without probability estimation, and with and without weighted probabilities. For each, we counted the pruning tests performed in three scenes: the Coffee-splash scene, the Chopper-titan scene, and a low-sampled Bathroom scene. The counter was increased each time a signed-area test was conducted, i.e., before each of the pruning-*if*s. A lower counter would mean that reordering the pruning tests has led to a higher number of early terminations. Table 6.1 shows the number of pruning tests for each of the pruning variants. Note that 4-fold pruning was expected to execute more pruning tests as there are four possible tests instead of three, but the pruning above is computationally cheaper than the left and right pruning. Thus, even if the 4-fold pruning variants record more pruning tests, they may still be faster than a 3-fold pruning variant with fewer tests, so a direct comparison between 4-fold pruning and 3-fold pruning may yield misleading

Pruning variant	Bathroom	%	Coffee-splash	%	Chopper-titan	%
3x	2,489,117K	100	21,870,580K	100	131,623,897K	100
3x, p.es.	2,101,578K	84.4	17,747,579K	81.2	111,800,423K	84.9
3x, p.es., w.	2,100,807K	84.4	17,747,579K	81.2	111,800,369K	84.9
4x	2,965,974K	100	25,738,931K	100	144,438,541K	100
4x, p.es.	2,551,236K	86.0	20,837,681K	81.0	118,144,192K	81.8
4x, p.es., w.	2,554,387K	86.1	20,882,274K	81.1	118,240,240K	81.8
3x, p.es, w., bb.	2,484,093K	-	21,863,053K	-	131,645,889K	-
4x, p.es, bb.	2,945,811K	-	25,069,369K	-	138,406,541K	-

Table 6.1: Pruning tests performed for each of the variants of the pruning approach, counted for a down-sampled version of the Bathroom scene, for the Coffee-splash scene and for the Chopper-titan scene. Numbers are rounded to thousands of pruning tests. Percentages indicate the relative amount of pruning tests compared to pruning without probability estimation for 3-fold and 4-fold variants, respectively, and were calculated from the original, un-rounded values. "p.es." means probability estimation. "w." means weights. "bb." means that these pruning variants by-passed the bounding circle test. For the latter, we did not calculate percentages to avoid misunderstandings regarding their performance. Their higher number of pruning tests may be compensated by saving the bounding circle test calculations.

results. We discovered that using probability estimation reduces the number of necessary pruning tests by up to nearly 20%. Using weights slightly improves the 3-fold pruning variants' efficiency, but is counter-productive for 4-fold pruning.

As we noticed that the bounding circle test slows down the algorithm, we implemented "quick and dirty" variants of probability estimation, as described in Section 4.8.2.2. We chose both the 3-fold and the 4-fold variants with the lowest number of pruning tests, respectively, and removed the bounding circle test in each of them. The results of these variants are included in Table 6.1 as well. Note that the number of pruning tests for disregarded bounding circles is relatively high (about as high as the number of tests for pruning without probabilities, for our test scenes). The reason is that the calculated probabilities do no longer reflect the actual probabilities of rays hitting circle areas. Still, we expected the savings of the bounding circle test to compensate for the higher number of necessary pruning tests.

We tested every pruning variant of Table 6.1 on several scenes in PBRT. Results show that it is difficult to name the preferred pruning variant (see Table 6.2). Each pruning variant consumes roughly the same amount of time. The reason may be the computational overhead of the bounding circle test for the variants using probability estimation. By-passing the bounding circle test slightly improves performance for most scenes, and 3-fold pruning performs slightly better than 4-fold pruning. The pruning variants are not faster

	Buddha	Killeroo	Fractal-Buddha	Glass	Microcity
UTM	37.70	216.28	247.36	107.40	565.20
3x	38.04 100.90%	217.30 100.47%	250.98 101.46%	107.46 100.06%	568.84 100.64%
3x, p.es.	37.84 100.37%	217.46 100.55%	251.38 101.63%	107.54 100.13%	569.26 100.72%
3x, p.es., w.	37.92 100.58%	217.92 100.76%	250.88 101.42%	107.34 99.94%	573.14 101.40%
3x, p.es, w., bb.	38.32 101.64%	213.68 98.80%	249.88 101.02%	106.86 99.50%	569.48 100.76%
4x	38.14 101.17%	217.48 100.55%	250.52 101.28%	107.12 99.74%	569.92 100.84%
4x, p.es.	38.88% 103.13%	217.58 100.60%	252.18 101.95%	107.78 100.35%	575.08 101.75%
4x, p.es., w.	38.52 102.18%	218.08 100.83%	251.98 101.87%	107.08 99.70%	572.32 101.26%
4x, p.es, bb.	37.90 100.53%	213.82 98.86%	250.62 101.32%	107.76 100.34%	569.06 100.68%

Table 6.2: Time-based comparison of the UTM approach (*UTM*) and the pruning variants. Times are total rendering times on the CPU in seconds, averaged over five runs. Percentages below times indicate the time consumed by a pruning variant relative to the time consumed by the UTM algorithm. "p.es." means probability estimation. "w." means weights. "bb." means that these pruning variants by-passed the bounding circle test. Note that, aside from the UTM, no barycentric coordinates have been calculated, resulting in incorrect images. The goal of this evaluation was to find the most efficient pruning variant to be used in the hybrid algorithm.

than the UTM algorithm for most scenes. Nonetheless, some are faster for some scenes, albeit in most cases this gain is minimal and, as stated before, the pruning variants do not compute barycentric coordinates. We still decided to evaluate one of the pruning variants in our hybrid algorithm. We selected the 3-fold pruning variant with weighted probability estimation and by-passed bounding circle test. It was the fastest pruning variant for the Killeroo, Fractal-Buddha, and Glass scenes and moderately fast for the other scenes. Furthermore, it was faster than the UTM algorithm for the Killeroo and Glass scenes.

6.3 Results

We evaluated the UTM and hybrid algorithms against PBRT-V3's standard ray-triangle intersection algorithm [PJH16c] and against Baldwin and Weber's [BW16] algorithm, which we re-implemented into PBRT-V3. The algorithms have been tested on the test scenes provided by Pharr et al. [PJH16d] so that their viability for realistic applications could be evaluated instead of testing in a laboratory environment. Scenes with high as well as scenes with low hit-rate have been selected to assess how the hit-rate influences the algorithms' performance. All algorithms have been built using CMake 3.10.0[1] and Microsoft Visual C++ 2015, and have been tested on a PC equipped with 8 GB RAM and an Intel Core i7-4770K processor with 3.5 GHz.

We let PBRT measure the rendering time. Time needed for pre-processing is not included. We evaluated the algorithms in terms of:

- the consumed time with (roughly) the same amount of intersections for achieving results of comparable quality, and
- the quality of the rendered image after a given rendering time.

6.3.1 Time-based comparison

In this subsection, we compare the time each algorithm consumed for achieving results of comparable quality. We rendered our scenes in PBRT using the same scenes and scene descriptions for each algorithm. Thus, for each algorithm, the same number of primary rays has been used and the intersection routines have been called roughly the same number of times. To meet the requirement of "comparable quality", we did not calculate squared errors over the generated images for the time-based comparison. As the realistic images generated through ray tracing are meant to present scenes for human observers, we did not follow a machine-based approach when comparing them. Instead, we qualitatively evaluated the images by comparing them in image viewing applications. We also quickly alternated between the images of the same scene generated by different algorithms as differences would become apparent as color jumps or movements. This way of qualitatively evaluating generated images was also employed by Baldwin and Weber [BW16].

We noticed that, to the naked eye, Baldwin and Weber's and Pharr et al.'s algorithms provide qualitatively the same results if we let PBRT render the scenes with exactly the same configurations. Ferko and Ferko's algorithms (the UTM algorithm and the hybrid algorithm) do so as well, the only difference in quality could be seen in the Coffee-splash scene, where these algorithms generated a small, white spot below the cup's edge on the left side, where the other algorithms rendered a continuous, dark-brown area. This may result from rounding differences since the UTM and hybrid algorithms operate differently. As we found no other apparent differences in the scene and the white spot does not

[1] https://cmake.org

look like an artifact or other erroneous pixel values, we still consider the generated Coffee-splash image to be valid.

For our tests, we selected scenes with different complexity and different hit-rates. This was done to identify scene configurations for which the UTM and hybrid algorithms would perfom rather good or rather bad. We included detailed scenes (like the Bathroom scene, depicted in Figure 6.1) as well as simpler ones (like the Teapot scene, depicted in Figure 6.2), resulting in intersection times from 37.7 to 9,949.1 seconds across all algorithms. Hit-rates ranged from 9.20% (Chopper-titan scene) to 45.07% (Cloud scene). We define the term "hit-rate" as the number of passed ray-triangle intersection tests in relation to the total number of ray-triangle intersection tests that an algorithm reports for a certain scene. Note that the hit-rates have been measured using Pharr et al.'s default algorithm. We noticed that the other algorithms report slightly different hit-rates. These originate from algorithms dealing differently with certain triangles. An algorithm may reject triangles which are accepted by another algorithm. We do not see these as false rejects or false accepts as we would have to define which algorithm produces the "correct" results, and the generated images exhibit comparable quality with almost no visual distinction, as stated above.

See Table 6.3 for the results of the time-based comparison. All times have been averaged over five runs per scene for each algorithm. Note that pre-processing time is not included in the displayed times. Baldwin and Weber tested their algorithm against the Möller-Trumbore algorithm, which is the standard algorithm in PBRT, version 2 [PH10b]. They reported that their algorithm outperformed the Möller-Trumbore algorithm, except at high hit-rates [BW16]. Pharr et al. use an algorithm like the one proposed by Woop et al. [WBW13] in PBRT-V3 [PJH16c]. We tested scenes with hit-rates up to 45.07%, but Baldwin and Weber's outperforms the PBRT-V3 algorithm for every scene. The UTM algorithm is faster than Pharr et al.'s for every scene as well, and it is comparable to or slightly slower than Baldwin and Weber's for most scenes. For the Teapot, Buddha, and Killeroo scenes, it is faster, however, the difference is marginal.

We only included results for the hybrid algorithm where the pruning approach has actually been called, i.e., where it would make a difference whether the UTM algorithm or the hybrid algorithm is used. More precisely, the Bathroom, Coffee-Splash, Teapot, and Cloud scenes do not trigger calls of PBRT's *IntersectP()* method for triangle intersection. It can be seen that the hybrid algorithm is slightly slower than the UTM algorithm for most scenes. This reflects the fact that when selecting the pruning approach to be used in the hybrid algorithm, for most scenes, our tests also showed no significant improvements in the pruning variants' performance, compared to the UTM algorithm's (see Table 6.2). Another reason for preferring the UTM algorithm over the hybrid algorithm in a ray tracer may be the latter's longer pre-processing time and higher memory consumption. These drawbacks originate from constructing the SPTM, calculating triangle areas and probabilities when probability estimation is used, and from storing both the UTM and the SPTM for each triangle.

We found that the UTM and hybrid algorithms are rather robust with regard to scene

Scene	Hit-rate	Pharr et al.	Baldwin-Weber	UTM	Hybrid
Bathroom	20.89%	2,456.60	2,367.20	2,388.56	-
		100%	96.36%	97.23%	-
Coffee-splash	15.61%	2,794.84	2,690.68	2,707.58	-
		100%	96.27%	96.88%	-
Chopper-titan	9.20%	9,949.10	9,351.90	9,562.70	9,598.92
		100%	94.00%	96.12%	96.48%
Teapot	28.00%	551.98	549.26	548.56	-
		100%	99.51%	99.38%	-
Buddha	22.65%	39.00	37.78	37.70	38.42
		100%	96.87%	96.67%	98.51%
Killeroo	22.89%	220.58	216.42	216.28	213.32
		100%	98.11%	98.05%	96.71%
Fractal-Buddha	13.15%	258.14	240.84	247.36	250.82
		100%	93.30%	95.82%	97.16%
Glass	31.52%	109.50	106.36	107.40	106.76
		100%	97.13%	98.08%	97.50%
Microcity	38.57%	588.98	558.76	565.20	565.28
		100%	94.87%	95.96%	95.98%
Cloud	45.07%	1,947.22	1,862.82	1,873.38	-
		100%	95.67%	96.21%	-

Table 6.3: Time-based comparison of Pharr et al.'s, Baldwin and Weber's, and the modified algorithms of Ferko and Ferko, one using solely the UTM approach (*UTM*) and the other using the hybrid approach (*Hybrid*). Times are total rendering times on the CPU in seconds, averaged over five runs. Percentages below times indicate the time consumed by an algorithm relative to the time consumed by PBRT's default algorithm (*Pharr et al.*). Hit-rates have been measured using the default algorithm as the relation of passed ray-triangle intersection tests to total ray-triangle intersection tests. For the hybrid algorithm, values are only included for scenes which could benefit from the pruning approach, i.e., scenes which trigger calls of PBRT's *IntersectP()* method.

complexity and hit-rate. For all scenes, they are faster than PBRT's default algorithm. They consume about 96% to 99% of the default algorithm's computation time. Although the Baldwin-Weber algorithm is generally faster, the UTM and hybrid algorithms perform comparably for some scenes.

In addition to scenes with different complexity and hit-rates, we also included the Fractal-Buddha scene to evaluate the UTM and hybrid algorithms on special modelling cases as well. The Fractal-Buddha scene uses PBRT's method of instancing and is constructed from tiny copies of itself (see Figure 6.3). We found that even for this scene, the performance increase is comparable to that for other scenes when using the UTM and hybrid algorithms instead of the default algorithm.

Figure 6.1: The Bathroom scene.

6.3.2 Quality-based comparison

For this evaluation, we compared the quality of images generated within a fixed rendering time. To achieve this in PBRT, which does not support interrupting the rendering process, we experimented with the sampling rate in the scene descriptions, using the *Stratified* sampler. Lower sampling rates reduce the quality of the rendered image as well as the consumed time. Once we found sampling rates for which PBRT's default algorithm, the Baldwin-Weber algorithm, and the UTM algorithm each consume (roughly) the same time for the same scene, we compared the generated images to a high-quality ground truth image of this scene. The latter has been generated by PBRT's default algorithm using a very high sampling rate (10,000 primary rays per pixel). We tolerated time differences of 1.5 seconds for each triple of test runs. The quality has been evaluated using the mean squared error between the generated image of an algorithm and the respective ground truth image. We selected scenes with relatively high as well as scenes with relatively low hit-rates. Again, we used the Fractal-Buddha scene as well. We did

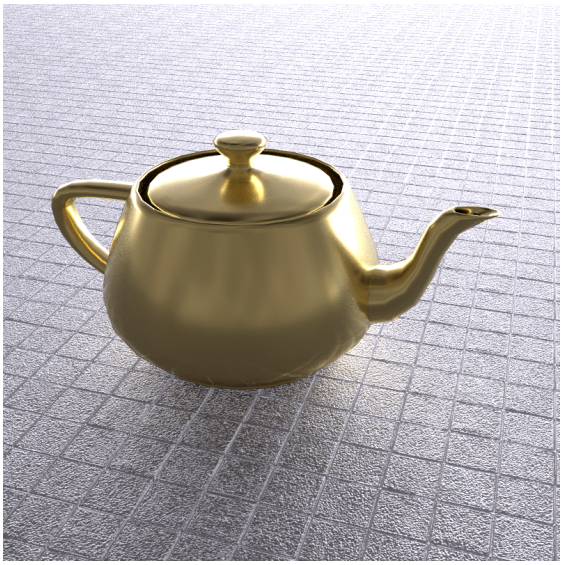


Figure 6.2: The Teapot scene.

Scene	Microcity	Killeroo	Fractal-Buddha
Consumed time	104.5–104.7s	193.6–194.9s	682.5–683.3s
Pharr et al.	0.0011146	0.1015484	0.00033312954
Baldwin-Weber	0.0010415	0.1278213	0.00031241373
UTM	0.0010349	0.1196676	0.00039471788

Table 6.4: Quality-based comparison of Pharr et al.'s, Baldwin and Weber's, and the UTM algorithms. Values are mean-squared errors to a high-quality ground truth image generated for the respective scenes. Times are in seconds.

not include the hybrid algorithm in this evaluation as the UTM algorithm performs better for most scenes.

See Table 6.4 for the results. For the images which consumed less computation time (Microcity and Killeroo), the images generated by the UTM algorithm show a lower mean squared error than the ones generated by the Baldwin-Weber algorithm. The default algorithm, however, proves to be the most accurate in the Killeroo scene and the least accurate in the Microcity scene. For the Fractal-Buddha scene, we used a relatively high sampling rate for each algorithm, resulting in lower error values. Here, the UTM algorithm performs the worst and the Baldwin-Weber algorithm the best. The Killeroo and Fractal-Buddha images do not exhibit visually noticeable differences. However, for each of the generated Microcity images, a different distribution of noise becomes apparent (see Figure 6.4).

Figure 6.3: The Fractal-Buddha scene. It uses the Stanford Buddha model to generate a Stanford Buddha model from 25,250 copies of itself.

The Bathroom scene was made available by "nacimus"[2] under the CC-BY license[3]. The Fractal-Buddha scene is based on the Stanford Buddha model, courtesy of the Stanford Computer Graphics Laboratory[4], and has been varied by Guillermo M. Leal Llaguno[5]. All scenes have been downloaded from PBRT's scene repository[6].

6.4 Comparison with related work

The algorithms proposed by Kensler and Shirley [KS06], Havel and Herout [HH10], Shevtsov et al. [SSK07], and Amanatides and Choi [AC97] directly test for ray-triangle

[2]https://www.blendswap.com/blends/view/73937, last accessed: 31.07.2018
[3]https://creativecommons.org/licenses/by/2.0/
[4]http://graphics.stanford.edu/data/3Dscanrep/, last accessed: 31.07.2018
[5]http://www.evvisual.com, last accessed: 31.07.2018
[6]http://pbrt.org/scenes-v3.html, last accessed: 31.07.2018

intersection in 3D. This is not the case for the new algorithm. Following Ferko and Ferko's outline, it uses 2D transformations to simplify further calculations, like is also done by Held [Hel97], Woop et al. [WBW13], Badouel [Bad90], Möller and Trumbore [MT97], and Baldwin and Weber [BW16]. The unit triangle transformation is most similar to the transformations used in these approaches.

The new algorithm first uses a ray-plane intersection test to calculate a potential intersection point, then verifies this point by additional tests. This approach is also discussed by Badouel [Bad90], Amanatides and Choi [AC97], and Segura and Feito [SF01].

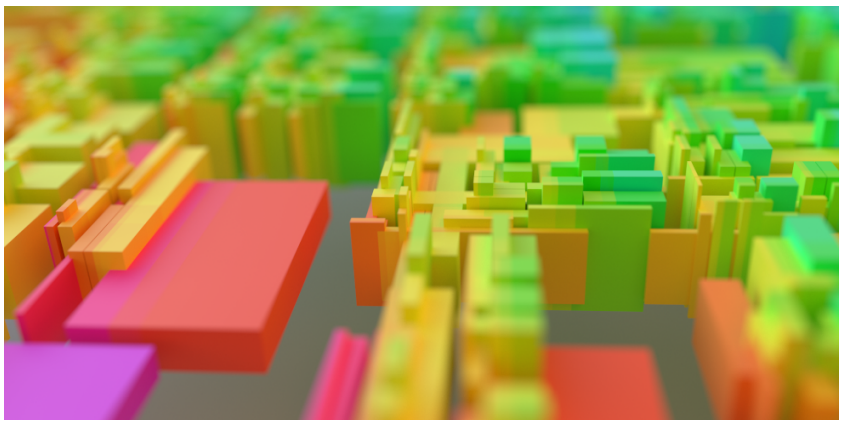
Kensler and Shirley [KS06], Held [Hel97], Amanatides and Choi [AC97], and Shevtsov et al. [SSK07] use Plücker coordinates for intersection tests. The new algorithm uses Plücker tests in the pruning approach. After the plane intersection point is known, in the UTM approach, it uses barycentric coordinates to do the 2D tests. The pruning approach does not primarily compute the intersection point, but only detects whether there is an intersection. Similar to Segura and Feito's algorithm [SF01], the intersection point can be calculated later using an additional unit triangle transformation.

Badouel [Bad90] and Held [Hel97] choose the plane the intersection problem is projected onto according to the triangle normal's highest-magnitude component. This avoids transformed triangles being degenerate in 2D [Bad90][O'R98]. Woop et al. [WBW13] and Pharr et al. [PJH16c] choose the transformation with regard to the ray's highest-magnitude component. The new algorithm projects every triangle onto the xy-plane, as is also done by Baldwin and Weber [BW16] and Amanatides and Choi [AC97]. Like Baldwin and Weber's algorithm [BW16], it incorporates the normal's highest-magnitude component when defining the UTM's free vector. In the pruning approach, the SPTM is constructed in such a way that the 2D triangle cannot be degenerate.

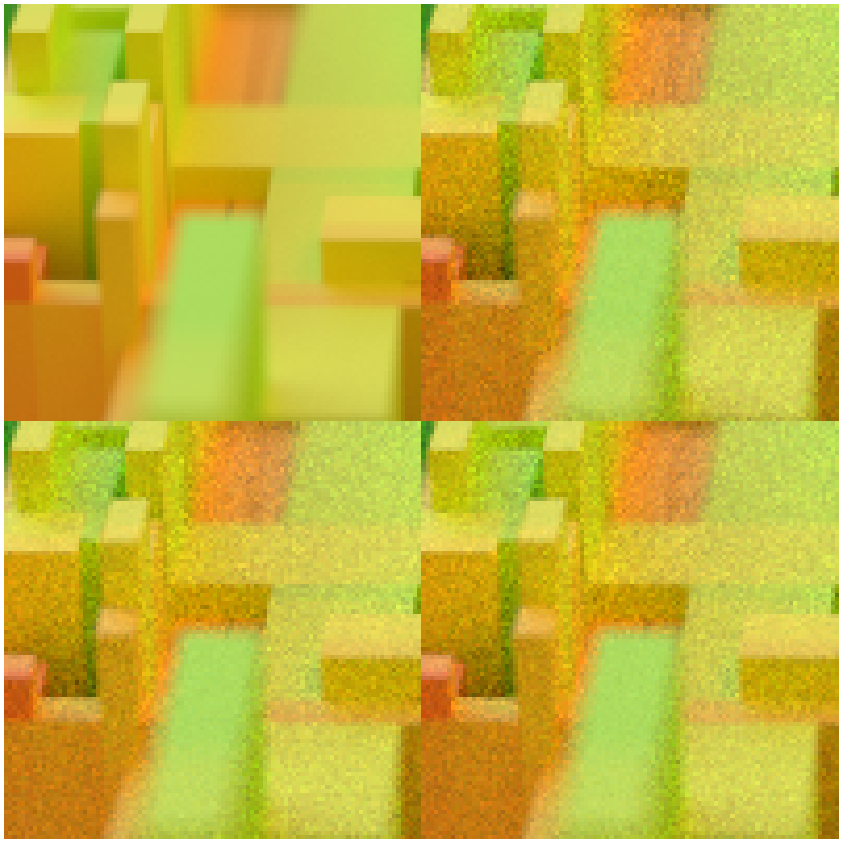
Held [Hel97], Segura and Feito [SF98][SF01], and Jiménez et al. [JSF10] represent rays as segments defined by two points. Shevtsov et al. represent triangles via one vertex and two edges [SSK07]. The new algorithm follows the approach of Pharr et al.'s PBRT [PJH16d] and represents rays as an origin and a direction, and triangles as a set of three vertices.

Wald [Wal04], Bikker [Bik07], Havel and Herout [HH10], Shevtsov et al. [SSK07], and Noguera et al. [NUH09] aim to achieve high-performance ray tracing using SIMD instructions and ray packeting. Jiménez et al. also implement several parallelized state-of-the-art algorithms [JONP14]. The new algorithm does not use SIMD instructions or ray packets as the purpose of its implementation is to compare and enhance performance with respect to the default ray-triangle intersection algorithm implemented in PBRT. Furthermore, the Baldwin-Weber algorithm [BW16], which it has been tested against, also does not use SIMD instructions or ray packets. Frameworks like Arauna [Bik07] or Manta [BSP06] are designed for real-time ray tracing. The new algorithm is designed for flexible, physically correct image rendering in PBRT and does not aim at real-time rendering, however, effort has been made to make it as fast as possible while maintaining Ferko and Ferko's [FF15] approach. The usage within PBRT is also the reason for the new algorithm being implemented for the CPU and not for the GPU.

Woop et al. [WBW13] address the watertightness problem of ray tracing results. The UTM approach of the new algorithm does not raise the claim of producing watertight results, however, it generates visually satisfying images. The pruning approach does not employ the simplified edge test which Woop et al. state as a reason of an algorithm not providing watertightness [WBW13]. Instead, the pruning operations are based on the same edge tests used by Woop et al. [WBW13] and Pharr et al. [PJH16c], who claim to produce watertight results. Thus, the pruning approach's results may be watertight as well, however, no effort has been made to evaluate the generated images in terms of watertightness.



(a)



(b)

Figure 6.4: Visible differences in noise distribution in the generated Microcity images. (a) The high-quality ground truth image, as a reference. (b) Close-ups of the images generated by the different algorithms (top left: ground truth, top right: Pharr et al., bottom left: Baldwin and Weber, bottom right: UTM).

CHAPTER 7

Summary and future work

This Chapter summarizes this thesis as well as the results and evaluation described in Chapter 6 (see Section 7.1). Section 7.2 addresses open issues of the implemented algorithm and proposes objectives for future research.

7.1 Summary

Ray tracing is a technique capable of synthesizing highly realistic images. Its ability of simulating the physical distribution of light throughout a scene provides advantages in a wide field of applications, like artistic image generation, movie production, visualization, or game design. It can help evaluating critical design decisions and minimizes the need of costly, physical prototypes in the automotive industry. The distribution of light in a scene is simulated using "rays". Ray tracing is, in general, seen as a rather slow method of image generation [GS08]. One of its core concepts is the calculation of intersections between rays and objects. Triangles are often-used, simple primitives. Furthermore, complex objects can be tesselated into triangles. Thus, a substantial amount of a ray tracer's computation time is consumed by ray-triangle intersections [HH10][Ben06][BW16][KS06], creating the need of fast ray-triangle intersection algorithms.

In the scope of this thesis, a novel, fast ray-triangle intersection algorithm for CPU-based ray tracing has been implemented into the physically based ray tracing framework PBRT, version 3 [PJH16d]. The algorithm has been proposed by Ferko and Ferko (2015) in an unpublished paper [FF15]. It is designed to produce fast results while allowing for pre-processing and higher memory consumption. It consists of two approaches: a unit triangle approach and a pruning approach.

We the algorithm as well as several possible optimizations. Other ray-triangle intersection algorithms have been studied and compared to the proposed algorithm in order to further optimize it. We found the most efficient configuration of Ferko and Ferko's algorithm

to be a ray-plane test, followed by a unit triangle transformation, followed by 2D tests. Bounding volume tests negatively affected the performance. We also implemented several variants of the pruning approach, identifying the most promising one in terms of efficiency, and combined it with the unit triangle approach. However, this hybrid approach was, in general, not as efficient as the unit triangle approach.

We compared Ferko and Ferko's algorithm to PBRT's default algorithm [PJH16c] as well as Baldwin and Weber's [BW16] algorithm, which we adapted for PBRT-V3. We did quality-based tests where we let the algorithms operate for the same time on the same scenes and evaluated the resulting image quality. We also did a time-based test, generating images of comparable quality and measuring the time each algorithm consumed. Scenes with different hit-rates, different complexity, and also special modelling concepts have been evaluated. We found that Ferko and Ferko's algorithm is robust with respect to scene configuration and performs better than PBRT's default algorithm for every scene. It reduces overall rendering time by up to 4%. Still, the Baldwin-Weber algorithm outperforms both algorithms for almost every scene. However, for some scenes, Ferko and Ferko's algorithm performs comparably.

7.2 Future work

Future research can explore how to use the SSE paradigm propagated for ray tracing by Wald [Wal04] to further improve performance. The watertightness problem as described by Woop et al. [WBW13] can be addressed for the UTM approach. Furthermore, watertightness tests can be conducted for the pruning approach. Like suggested by Löfstedt and Akenine-Möller [LAM05], more extensive evaluation can be done by comparing the new algorithm to other state-of-the-art algorithms, using several different machines and multiple compilers.

Tests with the different pruning variants can be expanded in future work. In particular, the weighted probability estimation may be improved by finding more efficient weighting principles. This may lead to even more efficient ordering of pruning tests and could improve the performance of the pruning approach and, ultimately, the performance of the hybrid algorithm.

All algorithms have been implemented into the PBRT framework using BVH as acceleration structure. This may be the main reason why the bounding volume tests lead to a decrease in performance. Using other acceleration structures may show different results and legitimate the use of bounding volume tests for fast ray-triangle intersection in Ferko and Ferko's algorithm.

PBRT is designed as a flexible, physically based ray tracing framework that aims for realism rather than performance [GS08]. Thus, there may be even more room for performance improvements.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AC97] John Amanatides and Kin Choi. Ray tracing triangular meshes. In *Proceedings of the Eighth Western Computer Graphics Symposium*, pages 43–52, 1997.

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Third Edition*. CRC Press, July 2008.

[AR13] Howard Anton and Chris Rorres. *Elementary Linear Algebra: Applications Version, 11th Edition*. Wiley Global Education, October 2013.

[Bad90] Didier Badouel. An Efficient Ray-Polygon Intersection. In *Graphics Gems*, pages 390–393. Academic Press Professional, Inc., San Diego, USA, 1990.

[Ben06] Carsten Benthin. *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006.

[Bik07] Jacco Bikker. Real-time Ray Tracing Through the Eyes of a Game Developer. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 1–10, Washington, D.C., USA, September 2007. IEEE.

[Bog88] Rod Bogart. Re: Ray/triangle intersection with barycentric coordinates. *Ray Tracing News*, 1(11), November 1988.

[BSP06] James Bigler, Abe Stephens, and Steven G. Parker. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 187–196, Salt Lake City, USA, September 2006. IEEE.

[BW09] Carsten Benthin and Ingo Wald. Efficient Ray Traced Soft Shadows Using Multi-frusta Tracing. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 135–144, New Orleans, USA, 2009. ACM.

[BW16] Doug Baldwin and Michael Weber. Fast Ray-Triangle Intersections by Coordinate Transformation. *Journal of Computer Graphics Techniques (JCGT)*, 5(3):39–49, September 2016.

[DEG+12] Tomáš Davidovič, Thomas Engelhardt, Iliyan Georgiev, Philipp Slusallek, and Carsten Dachsbacher. 3d Rasterization: A Bridge Between Rasterization and Ray Casting. In *Proceedings of Graphics Interface 2012*, pages 201–208, Toronto, Canada, May 2012. Canadian Information Processing Society.

[DK06] Holger Dammertz and Alexander Keller. Improving Ray Tracing Precision by Object Space Intersection Computation. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 25–31, Salt Lake City, USA, September 2006. IEEE.

[DWWS04] Andreas Dietrich, Ingo Wald, Markus Wagner, and Philipp Slusallek. VRML scene graphs on an interactive ray tracing engine. In *Proceedings of IEEE Virtual Reality 2004*, pages 109–282, Chicago, USA, March 2004. IEEE.

[Eri97] Jeff Erickson. Plücker coordinates. *Ray Tracing News*, 10(3), December 1997.

[Eri07] Christer Ericson. realtimecollisiondetection.net – the blog » Minimum bounding circle (sphere) for a triangle (tetrahedron). `http://realtimecollisiondetection.net/blog/?p=20`, July 2007. Accessed: 2018-07-31.

[FF15] Andrej Ferko and Michal Ferko. Precomputed Fast Rejection CPU Ray-Triangle Intersection. Unpublished manuscript, 2015.

[FGD+06] Heiko Friedrich, Johannes Günther, Andreas Dietrich, Michael Scherbaum, Hans-Peter Seidel, and Philipp Slusallek. Exploring the Use of Ray Tracing for Future Games. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, pages 41–50, Boston, USA, July 2006. ACM.

[GRHS08] Iliyan Georgiev, Dmitri Rubinstein, Hilko Hoffmann, and Philipp Slusallek. Real time ray tracing on many-core-hardware. In *Proceedings of the 5th INTUITION Conference on Virtual Reality*, 2008.

[GS08] Iliyan Georgiev and Philipp Slusallek. RTfact: Generic concepts for flexible and high performance ray tracing. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 115–122, Los Angeles, USA, August 2008. IEEE.

[Hel97] Martin Held. ERIT - A Collection of Efficient and Reliable Intersection Tests. *Journal of Graphics Tools*, 2(4):25–44, 1997.

[HH10] Jiří Havel and Adam Herout. Yet Faster Ray-Triangle Intersection (Using SSE4). *IEEE Transactions on Visualization and Computer Graphics*, 16(3):434–438, May 2010.

[JONP14] Juan J. Jiménez, Carlos J. Ogáyar, José M. Noguera, and Félix Paulano. Performance analysis for GPU-based ray-triangle algorithms. In *Proceedings of the 9th International Conference on Computer Graphics Theory and Applications (GRAPP 2014)*, pages 1–8, Lisbon, Portugal, January 2014. IEEE.

[JSF10] Juan J. Jiménez, Rafael J. Segura, and Francisco R. Feito. A robust segment/triangle intersection algorithm for interference tests. Efficiency study. *Computational Geometry*, 43(5):474–492, July 2010.

[KS06] Andrew Kensler and Peter Shirley. Optimizing ray-triangle intersection via automated search. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 33–38, Salt Lake City, USA, September 2006. IEEE.

[LAM05] Marta Löfstedt and Tomas Akenine-Möller. An evaluation framework for ray-triangle intersection algorithms. *Journal of Graphics Tools*, 10(2):13–26, 2005.

[MT97] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, January 1997.

[MW06] J. Mahovsky and B. Wyvill. Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. *Computer Graphics Forum*, 25(2):173–182, June 2006.

[NUH09] José Noguera, Carlos Ureña, and Rubén J. G. Hernandez. A Vectorized Traversal Algorithm for Ray Tracing. In *Proceedings of the 4th International Conference on Computer Graphics Theory and Applications (GRAPP 2009)*, pages 58–63, Lisbon, Portugal, February 2009.

[O'R98] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, USA, 2nd edition, 1998.

[PH04] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, August 2004.

[PH10a] Matt Pharr and Greg Humphreys. Chapter 3 - Shapes. In *Physically Based Rendering*, pages 106–181. Morgan Kaufmann, Boston, USA, 2nd edition, June 2010.

[PH10b] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, Boston, USA, 2nd edition, June 2010.

[PJH16a] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Chapter 1 - Introduction. In *Physically Based Rendering: From Theory to Implementation*, pages 1–55. Morgan Kaufmann, 3rd edition, November 2016.

[PJH16b] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Chapter 2 - Geometry and Transformations. In *Physically Based Rendering: From Theory to Implementation*, pages 57–121. Morgan Kaufmann, 3rd edition, November 2016.

[PJH16c] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Chapter 3 - Shapes. In *Physically Based Rendering: From Theory to Implementation*, pages 123–244. Morgan Kaufmann, 3rd edition, November 2016.

[PJH16d] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann, 3rd edition, November 2016.

[PRS+10] Steven G. Parker, Austin Robison, Martin Stich, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, and Keith Morley. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics*, 29(4):66:1–66:13, July 2010.

[RW80] Steven M. Rubin and Turner Whitted. A 3-dimensional Representation for Fast Rendering of Complex Scenes. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '80)*, pages 110–116, Seattle, USA, July 1980. ACM.

[SB87] John M. Snyder and Alan H. Barr. Ray Tracing Complex Models Containing Surface Tessellations. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87)*, pages 119–128, Anaheim, USA, July 1987. ACM.

[SF98] Rafael J. Segura and Francisco R. Feito. An algorithm for determining intersection segment-polygon in 3d. *Computers & Graphics*, 22(5):587–592, October 1998.

[SF01] Rafael J. Segura and Francisco R. Feito. Algorithms to Test Ray-Triangle Intersection. Comparative Study. *Journal of WSCG*, 9(3):76–81, 2001.

[Sho07] Thomas S. Shores. *Applied Linear Algebra and Matrix Analysis.* Undergraduate Texts in Mathematics. Springer, New York, USA, 2007.

[Shu13] Vladimir Shumskiy. GPU Ray Tracing – Comparative Study on Ray-Triangle Intersection Algorithms. In *Transactions on Computational Science XIX*, Lecture Notes in Computer Science, pages 78–91. Springer, Heidelberg, Germany, 2013.

[SSK07] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Ray-Triangle Intersection Algorithm for Modern CPU Architectures. In *Proceedings of GraphiCon 2007*, Moscow, Russia, June 2007. Moscow State University.

[SWW+04] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 95–106, Grenoble, France, August 2004. ACM.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

[WBW13] Sven Woop, Carsten Benthin, and Ingo Wald. Watertight ray/triangle intersection. *Journal of Computer Graphics Techniques (JCGT)*, 2(1):65–82, June 2013.

[Whi80] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, June 1980.

[WPS+03] Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime ray tracing and its use for interactive global illumination. *Eurographics State of the Art Reports*, 1(3):5, 2003.

[WSS05] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *ACM SIGGRAPH 2005 Papers*, pages 434–444, Los Angeles, USA, July 2005. ACM.

[WWB+14] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics*, 33(4):1–8, July 2014.