JACOBS
UNIVERSITY

Accelerated Ray Tracing of Constructive Solid Geometry

by

Otmane Sabir

Bachelor Thesis in Computer Science

Submission: April 20, 2021 Supervisor: Prof. Dr. Sergey Kosov

Jacobs University Bremen | Department of Computer Science and Electrical Engineering

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder VervielfÃ¤ltigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

Date, Signature

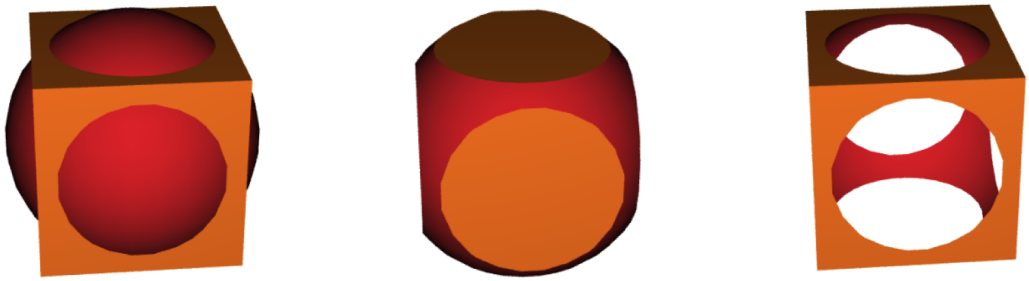
Abstract

(target size: 15-20 lines)

Contents

1 Introduction

Constructive Solid Geometry (CSG) is a method used in computer graphics, computer-aided design, generic modeling languages, and numerous other applications to construct complex geometries from simple primitives or polyhedral solids through the use of boolean operators, namely union ($\cup$), intersection ($\cap$), and difference ($-$). Figure 1 respectively shows union, intersection, and difference operations. The approach grows especially appealing when implemented in a ray tracing system as the core intricacy renders performing arithmetic logic on a pair of uni-dimensional rays. Nonetheless, most current ray tracing systems generally suffer from the detriment of the expensive object space intersection computation, and the generic CSG algorithms suffer immensely from their computational complexity, making it very difficult to integrate into operating rendering engines. Therefore, this research concentrates on constructive solid geometry and possible means of acceleration.



(a) Union. (b) Intersection. (c) Difference.

Figure 1: Examples of set operations on a mesh sphere and box.

1.1 Rendering Algorithms

Rendering digital photorealistic or non-photorealistic images has been a topic of study since the late 1960s [3]. Since then, various algorithms came forth that allow achieving different results depending on the required conditions. Inherently all these algorithms strive to solve the same underlying problem by trading-off different aspects, namely speed and realism. This problem is known as the hidden surface problem [15]. The hidden surface problem is determining the visible objects in space from a certain point of view. There are two general methods, object-space methods, which try to start from the object space and project the geometries onto the 2D raster, or the image-space ones, which perform the opposite by tracing a ray through each pixel and attempting to locate the closest intersection of that ray with the geometries in the scene. The two methods then give birth to the pair of most famous and widely adopted rendering algorithms: rasterization and ray tracing.

1.1.1 Rasterization

Rasterization has very quickly become the predominant approach for interactive applications because of its initially low computational requirements, its massive adoption in most hardware solutions, and later by the ever-increasing performance of dedicated graphics

hardware. The use of local, per-triangle computation makes it well suited for a feed-forward pipeline. However, the rasterization algorithm has many trade-offs. To name a few: handling of global effects such as reflections and realistic shading, and limitations to scenes with meshed geometries [21].

1.1.2 Ray Tracing

Ray tracing simulates the photographic process in reverse. For each pixel on the screen, we shoot a ray and identify objects that intersect the ray. A ray-tracing algorithm makes use of four essential components: the camera, the geometry, the light sources, and the shaders. These components can have different varieties, to state a few, orthographic and perspective cameras, unidirectional and area light sources, and Phong and chrome shaders. Hence, it allows achieving several outcomes depending on the necessities. The main downside has been computational time and the constraints of using such an algorithm in interactive applications. However, ray tracing parallelizes efficiently and trivially. Thus it takes advantage of the continuously rising computational power of the hardware. Many applications have successfully produced real-time ray tracing algorithms and allow for highly photorealistic results in interactive applications [1, 2].

1.2 Geometric Representations

When it comes to computer graphics, we can find numerous types of geometry descriptions [6, 8, 11, 16, 19, 24]. Many solutions exist that enable the simple conversion between these geometric formats [25]. However, there are predominantly two different representations in most geometric modeling systems [19]: boundary representations - commonly known as B-Rep or BREP - and constructive solid geometry - CSG. Each one of these representations brings forward different advantages, disadvantages, and limitations.

1.2.1 Boundary Representation

Boundary representations are indirect definitions of solids in space using their boundary or limit. This representation is usually a hierarchical composition of different dimensionally complex parts. On the very top, we have definitions of two-dimensional faces, which build on uni-dimensional edges that are subsequently built on dimensionless vertices (Figure 2). A BREP with non-curvilinear edges and planar faces is called a polygon mesh. A triangle is the simplest polygon and has the excellent property of always being co-planar. Additionally, polygons of any complexity are representable by a set of triangles. These qualities make triangular meshes a fundamental component in BREPs. The representations built on triangles are also highly optimized for fast operations. Therefore, we will mainly deal with triangular meshes in OpenRT, though it does offer descriptions for tetragon (quadrilateral) meshes.

1.2.2 Constructive Solid Geometry

Constructive solid geometry takes basis on the fundamental premise that any complex physical object is obtainable from a set of primitive geometries and the base boolean operations. CSG is radically different from BREPs as it does not collect any topological information but instead evaluates the geometries as needed by the case scenario. In other

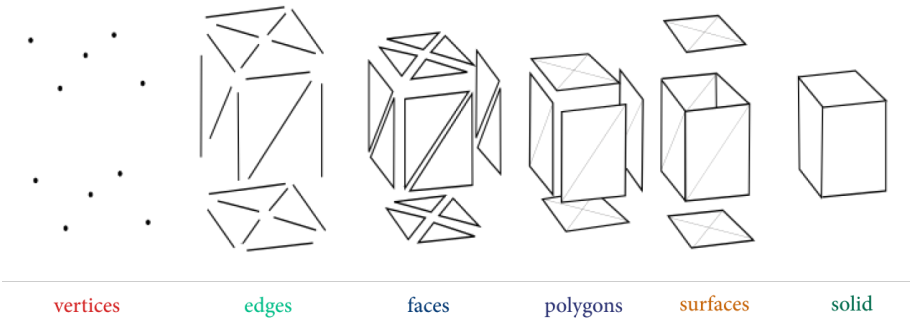


Figure 2: Sample BREP of a 3D hyper-rectangle [28]

words, there is no explicit description of the boundary of the solid. Contrary to BREPs, CSG representations are quickly modified and manipulated since incremental changes do not trigger re-computation and evaluation of the boundary of a geometry. Therefore, no topological changes occur when adjusting the geometries. The latter makes it an attractive solution as it provides a high-level specification of the objects in space and permits significantly more straightforward modification and manipulation. In the general constructive solid geometry description, the solids are put in a binary tree, referred to as the CSG tree (Figure 3). The root node is the complete composite geometry. The leaf nodes depict the base geometries (cubes, spheres, cylinders, tori, cones, and polygon meshes[1]) used in the composition. Every node in the tree, besides the leaf nodes, expresses another complete solid and contains information of the set operation of that node.

In the OpenRT library implementation, we follow a different approach to allow the use of BREPs as leaf nodes. This gives the flexibility of creating more complex geometries and allowing for nesting of combinatorial geometries inside each other. In a naive implementation, this operation can be costly; however, by using certain spatial indexing structures, each node can be represented as a binary space partitioning tree.

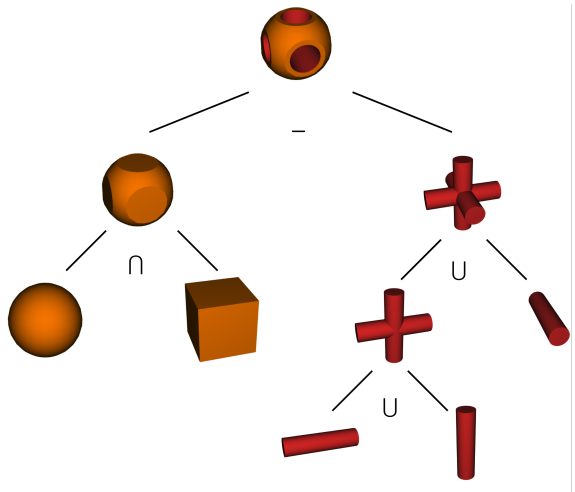


Figure 3: Sample representation of a CSG tree.

[1]Polygon meshes are usually not considered in CSG algorithms; however, the implementation discussed here allows such flexibility.

1.3 Overview

We present this CSG implementation in six sections. 1. Introduction; 2. Related Works; 3. Constructive Solid Geometry; 4. Optimization; 5. Evaluation of the results; 6. Extensions & Conclusions.

The first section was the previous introduction laying a foundation to a few topics we will be addressing.

The second section presents works already done, the limitations of the proposed implementations, and solutions to problems related to CSG.

Section 3 defines the algorithm that performs the logic in the ray-tracing framework. We first introduce the ideas behind ray intersection. We then lay a mathematical foundation to boolean algebra and membership classification. Additionally, we dive into the detail of ray classification for constructively generated geometries.

Section 4 discusses efficiency and optimization. The visible surface problem in ray tracing requires a lot of CPU time, and without any optimization, the CSG algorithm significantly increases the payload. Therefore, improvement is much needed to make this method usable and suitable for real-life applications. The speed is a function of the screen resolution and the geometry complexity (the number primitives (e.g., triangles) in the solid, and the number of nested geometries).

Section 5 describes the different implementations of the CSG algorithm with the various optimization techniques. The first is the naive implementation which we refer to as NaiCSG. The second is a variant that uses a Binary Space Partition tree to solve the visible surface problem but still naively finds intersections inside the combinatorial geometry, which we will refer to as BinCSG. Lastly, we'll introduce our optimized algorithm which uses a binary space partition tree on the outside (solving the visible surface problem) and also inside each composite geometry to direct the rays towards the correct geometries, which we will refer to as OptiCSG. We conduct three types of tests. The primary one is a function of time and complexity of the geometry, as we monitor the rendering time following gradual increases in the detail level of two sphere meshes. The second computes the time taken to render a scene after covering different amounts of the view port. The third computes the time variations after increasing the number of nested geometries present in the composite solid while crucially maintaining a consistent view port fill rate.

2 Related Work

I discuss below the techniques most related to ours. However, there is a tremendous body of work in this area and I cannot possibly provide an absolute overview. The goal is instead to outline similarities and differences with some of the widely adopted approaches for CSG modeling.

Constructive solid geometry has been a subject of study since the late 1970s. It was initially introduced in [29] as a digital solution to help in the design and production activities in the discrete goods industry, this marked the basis for formalizing the method.

A rigorous mathematical foundation of constructive solid geometry was later laid out in [18]. The membership classification function, a generalization of the ray clipping method,

is also thoroughly discussed and various formal properties are introduced.

A few years later it was revisited in [20] where Roth et al. (1982) introduced ray casting as a basis for CAD solid modeling systems. Challenges of adequacy and efficiency of ray casting are addressed, and fast picture methods for interactive modeling are introduced to meet the challenges.

The focus then turned towards different optimizations of CSG algorithms in the setting of ray tracing. A simplistic single hit intersection algorithm is introduced in [9]. This suggested mechanism reduces memory load and the number of computations performed for ray classification. Though limitations have to be respected since sub-objects must closed, non-self-intersecting, and have consistently oriented normals. However, this was later proven to be a solution that does not gracefully handle edge cases especially for the difference and intersection operations [31].

A "slicing" approach is also proposed in [13]. Similar to our proposed solution combinations of meshes and analytical primitives through CSG operations are permissible. Nevertheless, this approach requires one boolean per primitive and a complete evaluation of the CSG expression in each step; therefore, making it simple but limited, and much better approaches are imaginable.

Bound definitions are also a popular way of significantly reducing the time required by CSG algorithms. If the ray and the geometric entities are bound, we first perform a test to see if the ray and the bounding volume around a geometric entity overlap. Only when the boxes overlap does one continue to test whether the ray and the entity do so as well. A submitted S-bounds algorithm is brought forth in [4] as a means of acceleration in solid modeling and CSG.

Techniques that optimize various CSG rendering algorithms, namely the Goldfeather and the layered Goldfeather algorithm, and the Sequenced-Convex- Subtraction (SCS) algorithm are advanced in [10]. Although the work represents a significant improvement towards real-time image-based CSG rendering for complex models, the main focus is on hardware acceleration.

3 Constructive Solid Geometry

3.1 Ray Intersection

Ray intersection is the essence of all ray tracing systems. We supply the system a ray as input and obtain knowledge on how the ray intersects solids in the scene as an output. In ray tracing engines, one only necessitates computing the nearest intersection to assess the given scene. However, when evaluating CSG models, we require both the closest and furthest intersections to arrange the ray intersections. With knowledge of all the information in the scene - essentially the camera model and the solids - an evaluation of the closest and most distant intersection is executed with each returning the latter information:

$\vec{o}$ = the origin of the ray (e.g., camera model origin).
$\vec{d}$ = the direction of the ray (e.g., direction from camera origin to pixel in raster).
t = the distance to either the closest or furthest intersection.
$prim$ = a pointer holding surface information of the intersected primitive.

We can distinguish two types of ray intersections [21]. Firstly, ray-primitive intersection tests on convex primitives such as blocks, cylinders, cones, and spheres. Because the primitives are analytically defined, the solution is solving the analytic intersection equation. Consequently, this means that the intersection solution is primitive-specific. Many resources providing the analytical solutions are available [17]. Second, we encounter the more generic solid-ray intersection. As we have previously defined in the introduction, a solid is often a boundary representation composed of several triangles. Hence, the main intricacy in ray-solid intersection renders iterating over all primitives and reducing the problem to n ray-primitive intersection tests with n being the number of primitives (e.g., triangles) in the solid. We can consider the ray-solid intersection as a more general form of ray-primitive intersection since a primitive is always representable as a solid bearing a single surface. The interesting consequence of such an abstraction is that if we test a ray in the scene, the computation for determining ray intersection can be generalized to:

Algorithm 1: Ray-solid furthest and closest intersection.

```
Result: [in, out] intersection range
in = +inf;
out = -inf;
for every primitive in the solid do
    solve the ray-primitive equations;
    if intersection exists then
        if current intersection is closer than in then
            set in to current intersection;
        if current intersection is further than out then
            set out to current intersection;
end
```

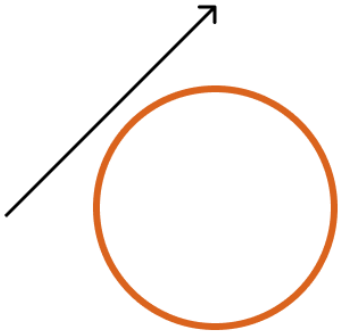
The ray-solid intersection test has four possible outcomes:

1. The ray misses the solid (Figure 4a).
2. The ray is tangent to the solid (Figure 4b).
3. The ray enters and exists the solid (Figure 4c).
4. The ray is inside/on the face of a solid and has one intersection. (Figure 4d)

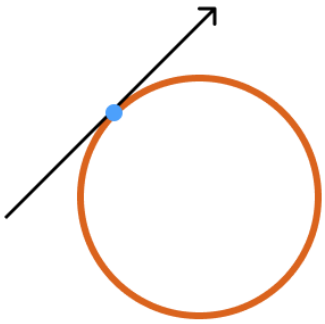
The first case is self-evident. In case 3, we compute both the entering and exiting points normally. The second and fourth case are more intricate to determine as we need to understand whether the intersection is inside or outside of the solid. We can find that by checking the orientation of the surface normal, $\vec{N}$, at the intersected point. If $\vec{N} \cdot \vec{d} < 0$, then the intersection point is outside. Otherwise, it is inside of the solid.

3.2 Mathematical Formulations

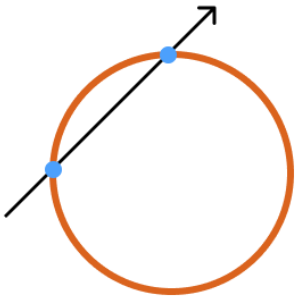
Constructive solid geometry is largely grounded in modern Euclidean geometry and the general topology of subsets of three-dimensional Euclidean space E^3 [18]. As one cannot design a reliable geometric algorithm in the absence of a clear mathematical statement of the problem to be solved, I will be treating a few mathematical formulations. Topology and set theory have been intensively discussed previously in [18], [27],[12], and many other resources. Hence, I will be mainly focusing on definitions and properties that interest



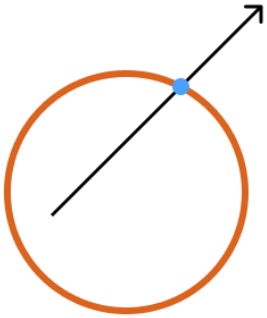
(a) Miss intersection.



(b) Tangent intersection.



(c) Complete intersection.



(d) Inside intersection.

Figure 4: Different ray intersection cases on a disk.

us. Formal proofs of the introduced properties are also available in the before-mentioned resources.

3.2.1 Set Algebra

Definition 3.1 (Set Operations)**.** Assume that X and Y are subsets of a universe W. We can use the following standard notations:

$$X \cup Y \tag{1}$$

$$X \cap Y \tag{2}$$

$$X - Y \tag{3}$$

Where (1), (2), and (3) respectively denote the union, intersection, and difference of the subsets X and Y.

Property 3.1. *Union and intersection operations are commutative. [14]*

$$X \cup Y = Y \cup X$$

$$X \cap Y = Y \cap X$$

Property 3.2. *Union and intersection operations are distributive over themselves and each other. [14]*

$$X \cup (Y \cap Z) = (X \cup Y) \cap (X \cup Z)$$
$$X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z)$$

Property 3.3. *The empty set $\emptyset$ and the universe W are identity elements for the union and intersection operators. [14]*

$$X \cup \emptyset = X$$
$$X \cap W = X$$

Property 3.4. *The complement, denoted c, satisfies [14]:*

$$X \cup cX = W$$
$$X \cap cX = \emptyset$$

Definition 3.2. Conducting the three operations $\cup$, $\cap$, and $-$ on a set of elements from the universe W while satisfying the properties (3.1) to (3.4) is called boolean algebra [18].

3.2.2 Topological Spaces

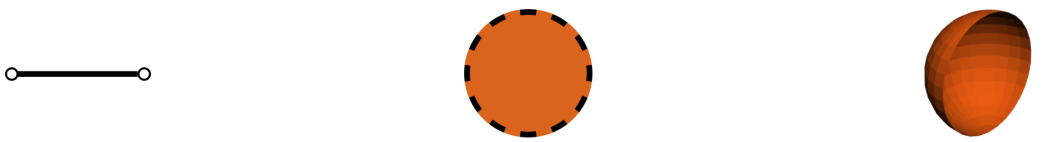
Topological spaces are a generalization of metric spaces in which the notion of "nearness" is introduced but not in any quantifiable way that requires a direct distance definition [14].

Definition 3.3 (Topological Space)**.** A topological space is a pair (W, T) where W is a set and T is a class of subsets of W called the open sets and satisfying the three properties 3.5, 3.6, and 3.7. (Figure 5)

Property 3.5. *The empty set $\emptyset$ and the universe W are open.[14]*

Property 3.6. *The intersection of a finite number of open sets is an open set. [14]*

Property 3.7. *The union of any collection of open sets is an open set. [14]*



(a) Open interval. (b) Open disk. (c) Open "sphere".

Figure 5: Representation of different open sets varying in dimensional order.

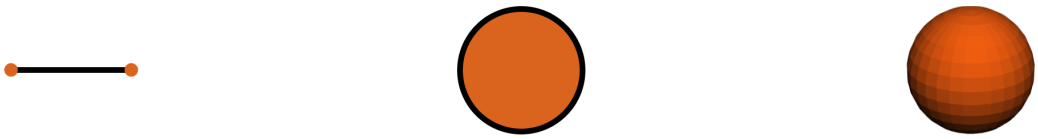
3.2.3 Closed Sets

Definition 3.4 (Closed Sets)**.** A subset X of a topological space (W, T) is closed if its complement is open[2]. Closed sets hold the following properties which are duals of properties (3.5) to (3.7). (Figure 6)

Property 3.8. *The empty set $\emptyset$ and the universe W are closed. [14]*

Property 3.9. *The union of a finite number of closed sets is a closed set. [14]*

Property 3.10. *The union of a finite number of closed sets is a closed set. [14]*



(a) Closed interval. (b) Closed disk. (c) Closed sphere.

Figure 6: Representation of different closed sets varying in dimensional order.

3.2.4 Neighborhood

Definition 3.5. The neighborhood, denoted $N(y)$, of a point y in a topological space (W, T) is any subset of W which contains an open set which contains y. If $N(y)$ is an open set, it is called an open neighborhood [18]. (Figure 7)

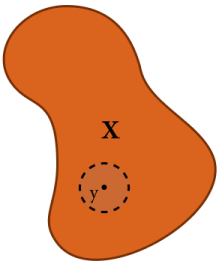


Figure 7: Interior point y on a subset X. The disc around y is the neighborhood of y.

3.2.5 Interior

Definition 3.6. A point of y of W is an interior point of a subset X of W if X is a neighborhood of y. The interior of a subset X of W, denoted iX, is the set of all the interior points of X [18].(Figure 7)

3.2.6 Boundary

Definition 3.7. A point y of W is a boundary point of a subset X of W if each neighborhood of y intersects both X and cX. The boundary of X, denoted bX, is the set of all

[2]However, this don't mean that closed sets are the opposite of open sets (e.g. the universe W and the null set $\emptyset$ are both open and closed)[14].

boundary points of X [18]. (Figure 8)

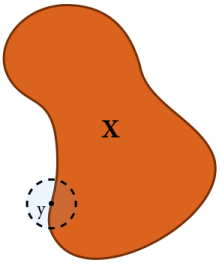


Figure 8: Boundary point y on a subset X.

3.2.7 Closure

Definition 3.8. The closure of a subset X, denoted kX, is the union of X with the set of all its limit points. A point is a limit point of a subset X of a topological space (W, T) if each neighborhood of y contains at least a point of X different from y [18]. (Figure 9)

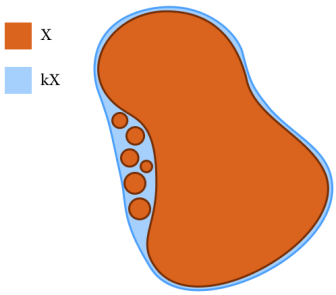


Figure 9: Closure kX of a subset X.

3.2.8 Regularity

Definition 3.9 (Regularity)**.** The regularity of a subset X of W, denoted rX, is the set of $rX = kiX$. [14]

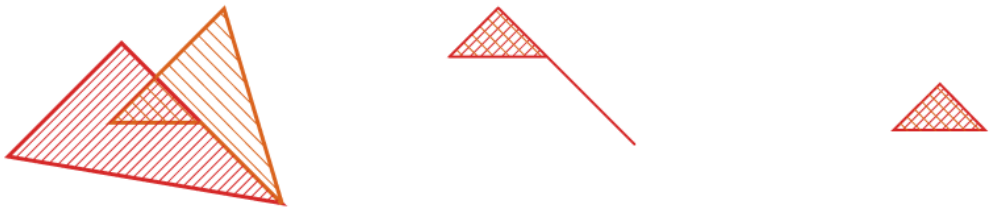
Definition 3.10 (Regular Set)**.** A set X is regular if $X = rX$, i.e. if $X = kiX$ [14]. (Figure 10)

Definition 3.11 (Regularized Set Operators)**.** The regularized union, intersection, difference and complement are defined per:

$$
\begin{aligned}
X \cup^* Y &= r(X \cup Y) \\
X \cap^* Y &= r(X \cap Y) \\
X -^* Y &= r(X - Y) \\
c^* X &= rcX
\end{aligned}
$$

3.2.9 Membership Classification Function

The membership classification function allows to segment a candidate set into three subsets which are the ”inside”, ”outside”, and ”on the” of the reference set [27]. Here, we will



(a) Initial polygons.

(b) Typical intersection with dangling edge.

(c) Regularized intersection.

Figure 10: Typical polygon intersection versus regularized intersection.

Table 1: Notation

E^n	Euclidean n-space
$\emptyset$	Empty Set
W	Reference Set Universe
W'	Candidate Set Universe
$\cup, \cap, -, c$	Set Operators
$\cup^*, \cap^*, -^*, c^*$	Regularized Set Operators in W
$\cup^{*'}, \cap^{*'}, -^{*'}, c^{*'}$	Regularized Set Operators in W'
i, b, k, r	interior, boundary, closure, and regularity in W
i', b', k', r'	interior, boundary, closure, and regularity in W'

abstractly define membership classification before moving to the practical implementations of the more specific ray classification. This theory depends heavily on the previously defined notions of interior, closure, boundary, and regularity. For a brief recapitulation, a point y is an element of the interior of a set X, denoted iX, if there exists a neighborhood of y that is contained in X; y is an element of the closure of X, kX, if every neighborhood of y contains a point of X; y is an element of the boundary of X, bX, if y is an element of both kX and $k(cX)$, where c denotes the complement. A set is said to be regular if $X = kiX$.

The membership classification function works on a pair of point sets:

S = The regular reference set in a subspace W.
X = The candidate regular set X, classified with respect to S, in a subspace W' of W.

Primed symbols will be used in order to denote operations on the subspace W' while normal symbols will be used to denote the subspace W (Table 1).

Definition 3.12. The membership classification function, M is defined as follows:

$$M[X, S] = (XinS, XonS, XoutS). \tag{4}$$

where

$$XinS = X \cap^{*'} iS$$
$$XonS = X \cap^{*'} bS$$
$$XoutS = X \cap^{*'} cS$$

The results obtained from this classification ($XinS, XonS, XoutS$) are the regular portions of the candidate set, X, in the interior, boundary, and the exterior of the reference set W (Figure 11). The produced results are a quasi-disjoint decomposition of the candidate; therefore:

$$X = XinS \cup XonS \cup XoutS \tag{5}$$

and for "almost" all points in the subset:

$$XinS \cap XonS = \emptyset$$
$$XonS \cap XoutS = \emptyset$$
$$XinS \cap XoutS = \emptyset$$

We say almost since the subsets are generally not disjoint in the conventional sense. (e.g. in Figure 11 $XinS$ and $XonS$ share a boundary point). [14]

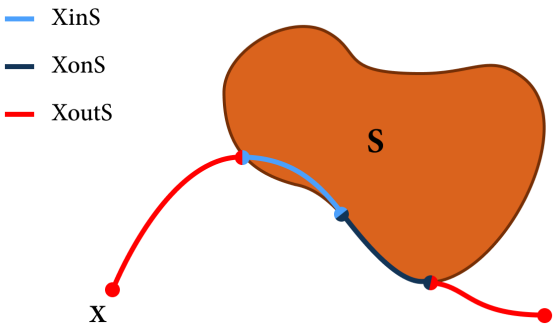


Figure 11: Membership classification function.

3.2.10 Classification by constructive geometry

Constructive geometry representations are binary trees whose nonterminal nodes designate regularized set operators and whose terminal nodes designate primitives. We refer to the specific case of constructive geometry in E^3 where regularized compositions are constructed of solid primitives as constructive *solid* geometry. Regular sets are closed under the regularized set operators thus a class of regular sets can be represented constructively as a combination of other more simple (regular) sets [18].

For example, as illustrated in Figure 12, if the universe W is in E^2 and we select the class of closed half-planes as our primitives, we could construct any regular set in E^2 given that it is bounded by a finite number of straight line segments.

We choose to define the constructively represented regular sets using the divide-and-conquer paradigm as it is a natural approach to compute the value of such a function. Therefore, when a regular set S is not a primitive, a nonterminal node, we convert the problem of evaluating the function $f(S)$ into two simpler instances of f followed by a combine, g, step. When S is a primitive, a terminal node, the problem can no longer be divided and an evaluator, ef, is used. We can now consider the general function for evaluation M when the reference set S is represented constructively.

$$M[X, S] = \begin{cases} eM(X, S), & \text{if } S \subset A \\ g(M[X, \text{l-subtree}(S)], M[X, \text{r-subtree}(S)], \text{root}(S)), & \text{otherwise} \end{cases} \tag{6}$$

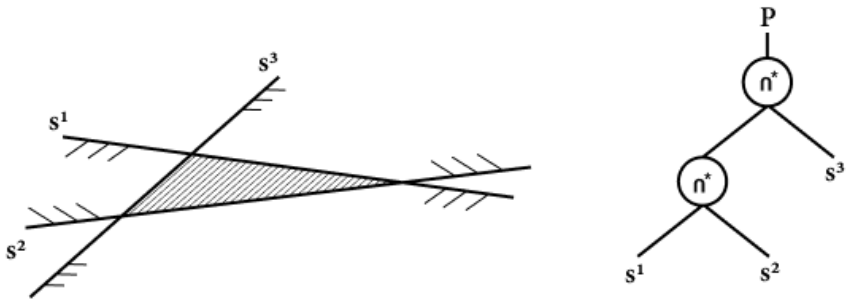


Figure 12: A constructive representation of a polygon P using half-planes. The tree on the right is the constructive geometry representation.

where

S	= The regular reference set.
X	= The candidate regular set.
eM	= The primitive evaluation function.
A	= The set of all allowed primitives.
g	= The combine function.
l-subtree	= The left subtree.
r-subtree	= The right subtree.
root	= The operation type. [3]

To customize this general definition to be used in a specific domain, one must design the classification procedure, eM, and the combine procedure. We have already defined our primitive classification procedure in Section 3.1. The combine procedure is discussed thoroughly in the next section.

3.3 Ray classification

Given a ray and a solid composition tree, our procedure classifies the ray with respect to the solid and returns the classification to the caller. As previously defined, the classification of a ray with respect to a solid is the information describing the closest and furthest ray-solid intersection, $[in, out]$. The procedure starts at the top of the solid composition tree, recursively descends to the terminal nodes, classifies the ray with respect to the primitives, then returns the tree combining the classifications of the left and right subtrees. Therefore, if our classification runs on a tree with depth 1, we would receive an $[in, out]$ for each of the solids in the leaf nodes. As illustrated in Figure 13, we can compute the combinations of the pair of intersections using boolean algebra (Table 4).

4 Optimization

In this section, we will introduce the state-of-the-art CSG algorithm that is implemented in the OpenRT framework. Here we expose all the adjustments and changes we have made to the algorithm in order to maximize its performance and results. We will discuss

[3] The current node always contains the operation.

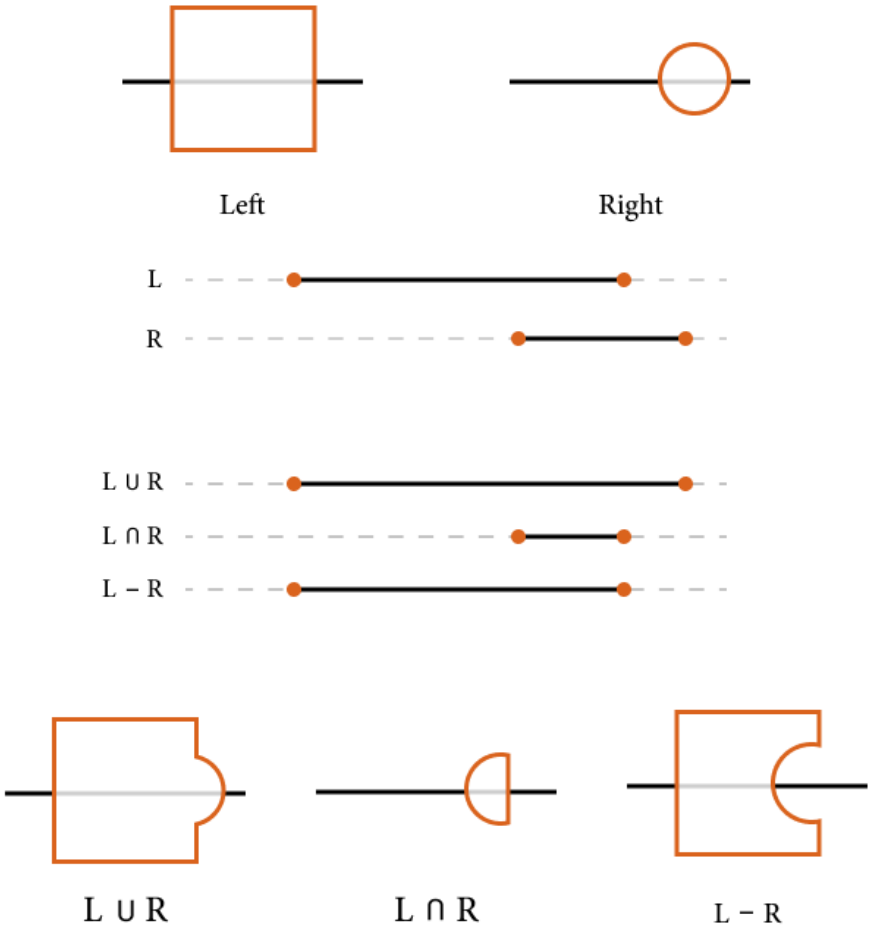


Figure 13: Example of combining ray classifications.

a minimal hit classification algorithm, box enclosures, and how simple techniques such as ”early-outs” can increase performance. Additionally, we propose the limitations of all these methods. We will discuss the binary space partition tree acceleration and spatial indexing structure, including a modified version in order to optimally reverse the tree traversal. Finally, we will put it all together in our version of the CSG algorithm.

4.1 Minimal hit CSG classification

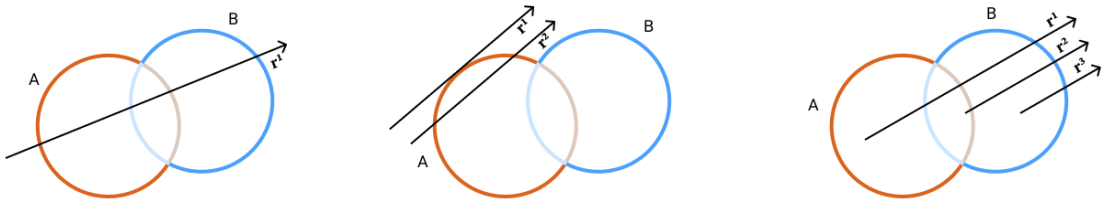
What we have introduced in the Section 3.3 is the typical approach to rendering CSG. However, this approach could be very costly as we nest more geometries in the tree and require lots of memory to store, classify, and combine a long chain of operations and primitives. Therefore, we introduce a new approach which we refer to as minimal hit CSG classifications. The approach described here computes intersections with binary CSG objects using the single nearest intersections whenever possible. Though it may need to do several of these per sub-object, the number needed is quite low and is never higher than that of the first method. To our knowledge, this approach was entirely developed in OpenRT and no resources found describe this type of ray classification. Though a relatively similar algorithm has been introduced in [9], it was proven in [31] to not be functional for the intersection and difference operations.

Table 2: Boolean operations table

Set Operator	Left Solid	Right Solid	Composite
$\cup$	in	in	in
	in	out	in
	out	int	in
	out	out	out
$\cap$	in	in	in
	in	out	out
	out	in	out
	out	out	out
$-$	in	in	out
	in	out	in
	out	in	out
	out	out	out

4.1.1 Union Classification

Consider the case of two the spheres shown in Figure 14. The union of these two solids is the boundary of each of the spheres without their interior. Therefore, to find the correct classification results we must find the closest intersection from our ray origin such that it does not belong to the interior of the sphere. Let us denote the intersection distance from the ray origin to the point as t_A. $\vec{P_A}$ is the intersection point which we can compute through the general ray equation $\vec{P_A} = \vec{o} + t_A\vec{d}$. Originally, min_A is set to $+\inf$, and max_A is set to $-\inf$. Additionally, min_A takes the value of t_A if and only if $t_A \leq min_A$. Meanwhile, max_A takes the value of t_A if and only if $t_A \geq max_A$. All the minimum and maximum values keep their default values if their corresponding intersections don't exist. (see Table 3)



(a) Ray goes through both spheres.

(b) Ray misses one of the spheres.

(c) Ray is inside one of the spheres.

Figure 14: Union ray classification cases.

For the case where the ray enters both spheres (Figure 14a), one would only have to find $MIN(min_A, min_B)$ in order to conclude which one of the boundaries of the sphere is closest.

Let us now examine the case where no intersection is found with one or all of the solids as shown in Figure 14b. In case the ray only hits one of the entries, we need to check if the ray exits the boundary of the missed sphere. If the ray doesn't exit the second sphere, we are sure the ray misses the second sphere and can return min_A. Otherwise, this leads

Table 3: Notation

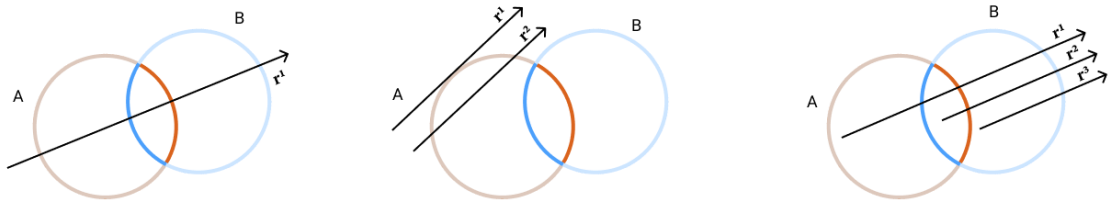
A, B	left sphere, right sphere.
t_A, t_B	Current ray intersection distance with A, B.
min_A, min_B	Closest ray intersection distance with A, B.
max_A, max_B	Furthest ray intersection distance with A, B.
P_A, P_B	Intersection point for A, B.

to the r^1 case of Figure 14c.

The last set of cases arise when the ray is shot from the interior of the spheres. This is more intricate since we have to teach our ray tracer to neglect the inner sides and only get the outer sides. The first scenario can occur when the ray only misses one of the entries but exits both solids. Here, we must ensure that the order of the evaluation in terms of distance is $min < max < max$. If so, then we can render the furthest max as our intersection (r^1 from Figure 14c). If not, this means the spheres are disjoint from each other, leading to $max < min < max$, and we return the closest max. If the ray is shot from the shared interior of the two spheres (r^2 from Figure 14c), this means the ray has missed both entries and one has to return the furthest of the two exit intersections. This solution is also valid for the last case (r^3 from Figure 14c) since the value of max is set to $-\inf$; therefore, the greatest of the two would be the current intersection. Algorithm 2 describes this function.

4.1.2 Intersection Classification

We will stick to the same general example; however, we will be performing the intersection of two spheres. (Figure 15)



(a) Ray goes through both spheres.

(b) Ray misses one of the spheres.

(c) Ray is inside one or both spheres.

Figure 15: Intersection ray classification cases.

The intersection of two spheres is their interior without the boundaries. We will apply the same previously defined notations shown in Table 3. First, we will begin with the obvious case where a ray goes through both A and B, as shown in Figure 15a. We first only compute min_A and min_B. In case min_A and min_B have values deviating from $\pm\inf$, then we are guaranteed our ray is going through both the spheres. But we still need to verify if the spheres have a shared interior (the intersection of two disjoint sets is the empty set $\emptyset$). Therefore, we must analyze the exit point of the closest entry. If that exit point doesn't lie between the two entry points, then we must return this as a miss. However, if the out point lies between the two spheres. Then we return the furthest min as the intersection.

Algorithm 2: Minimal hit classification for union.

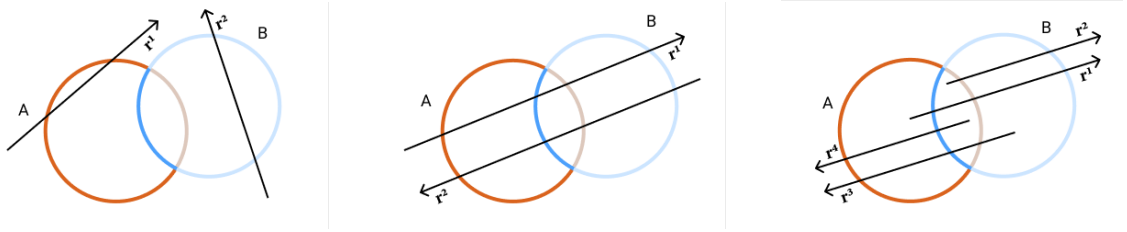
Result: t intersection distance
min_A = intersectMin(A);
min_B = intersectMin(B);
if $min_A != \inf$ *and* $min_B != \inf$ **then**
  return $MIN(min_A, min_B)$;
if $min_A != \inf$ *and* $min_B == \inf$ **then**
  max_B = intersectMax(B);
  if $max_B = -\inf$ **then**
    return min_A;
  if $max_B < min_A$ **then**
    return max_B;
  max_A = intersectMax(A);
  return max_A;
if $min_A == \inf$ *and* $min_B != \inf$ **then**
  max_A = intersectMax(A);
  if $max_A == -\inf$ **then**
    return min_B;
  if $max_A < min_B$ **then**
    return max_A;
  max_B = intersectMax(B);
  return max_B;
if $min_A == \inf$ *and* $min_B == \inf$ **then**
  max_A = intersectMax(A);
  max_B = intersectMax(B);
  if $max_A == -\inf$ *and* $max_B == -\inf$ **then**
    return $miss$;
  return $MAX(max_A, max_B)$;

The second case is when a ray only intersects with one of the spheres, as depicted in Figure 15b. In case one of the previously computed entries has a default value, we must check if the sphere with no min has an exit point. If it doesn't, then this means the ray completely misses one of the spheres and we return a miss (r^1 and r^2 in Figure 15b, and r^3 in Figure 15c).

The last cases are when the sphere is inside one or both of the solids. The first case arises when the ray doesn't enter one of the spheres but continues to have all other exits. Here we must check if the values of all the intersections are in the following order $min < max < max$. If that's the case, then we return the first min. Otherwise, this hints that the spheres are disjoint; therefore, we classify them as a miss. If the spheres lie within the interior area shared by the two spheres (r^2 in Figure 15c), we return the closest of both the exit points.

4.1.3 Difference Classification

The difference operation remains the most complex and least optimizable operation out of all three due to several factors. First, the minimal knowledge of the scene is higher than both previously mentioned operations. Second, the difference operations are not commutative nor distributive; therefore, the direction of the ray renders completely different results. We shall stick to the same example as the previous two cases and with similar notations (Figure 16).



(a) Ray misses one of the spheres.

(b) Ray goes through both spheres.

(c) Ray is inside one or both spheres.

Figure 16: Difference ray classification cases.

We will first consider the case where a ray misses one of the two spheres, as shown in Figure 16a. Contrary to union and intersection, we always get both the closest and furthest intersection points of A. If the ray does not enter and exit A, we can already consider this a miss (r^2 in Figure 16a). However, if it either enters or exits A, we must already review if it enters B, thus we compute min_B. If the ray doesn't enter B, we return the shortest of the two distances min_A and max_A. Otherwise, we move to the next case.

As illustrated in 16b, this case heavily depends on the direction of the ray and only stands if all enter and exit points of both spheres are valid. Starting with r^1, if we have completed the first check of entering and exit of A we can move to check B. We now need both the entrance and exit points of B. Assuming Bs points are both not the default values ($\pm\inf$), we must first check if $min_A < min_B < max_A$. If this holds, then we can already return the entry point of A as our intersection. If it doesn't (r^2 in Fig 16b) we must then check if $min_B < min_A < max_B < max_A$, then we can return the exit of B as the intersection. Otherwise, this case is considered a miss.

The last and most complicated case is when the ray lies somewhere inside one of these

Algorithm 3: Minimal hit classification for the intersection.

```
Result: t intersection distance
min_A = intersectMin(A);
min_B = intersectMin(B);
if min_A != inf and min_B != inf then
    if min_A < min_B then
        max_A = intersectMax(A);
        if min_B < max_A then
            return min_B;
    if min_B < min_A then
        max_B = intersectMax(B);
        if min_A < max_B then
            return min_A;
    return miss;
if min_A != inf and min_B == inf then
    max_B = intersectMax(B);
    if max_B == -inf then
        return miss;
    max_A = intersectMax(A);
    if min_A < max_B < max_A then
        return min_A;
    return miss;
if min_A == inf and min_B != inf then
    max_A = intersectMax(A);
    if max_A == -inf then
        return miss;
    max_B = intersectMax(B);
    if min_B < max_A < max_B then
        return min_B;
    return miss;
if min_A == inf and min_B != inf then
    max_A = intersectMax(A);
    max_B = intersectMax(B);
    if max_A == -inf or max_B == -inf then
        return miss;
    return MN(max_A, max_B);
```

two spheres. Starting with r^1, we already understand that the ray doesn't enter but only exists A, thanks to the calculation from the first case. We verify if the ray enters or exits B. If the ray enters and exists B, then we can return $MIN(max_A, max_B)$. If it only exits B, then we necessitate checking if $max_A < max_B$. If that holds, it is a miss. If it doesn't, we return max_B.

Algorithm 4: Minimal hit classification for the intersection.

```
Result: t intersection distance
min_A = intersectMin(A);
max_A = intersectMax(A);
if min_A == inf and max_A == -inf then
    return miss;
min_B = intersectMin(B);
max_B = intersectMax(B);
if min_B == inf and max_A == -inf then
    return MIN(min_A, max_A);
if max_A != -inf and max_B != -inf then
    if min_A != inf then
        if min_B != inf then
            if min_A < min_B and min_B < max_A then
                return min_A;
            if min_B < min_A and max_B < max_A then
                return max_B;
        if min_B == inf then
            if min_A < max_B and max_B < max_A then
                return max_B;
    if min_A == inf then
        if min_B != inf then
            if min_B < max_A then
                return min_B;
        if min_B == inf then
            if max_B < max_A then
                return max_B;
return miss;
```

4.2 Bounding Boxes

Bounding boxes are the simplest way to cut down on the number of ray intersection operations and reduce overall rendering time [22]. Let's visualize the situation where a union of two spheres composed of 100 triangles lies in the middle of a 500x500px view of which the composite covers 100x100 pixels. In the former approach, we would examine every single ray with the complete composite. Resulting in a staggering $25.000.000$ intersection checks; though, we solely necessitate a fifth of that. We introduce a box enclosure to do a preliminary examination before testing the rest of the composite. Hence, with a tight enough box (covering 110x110), the ray tracer would only need to check for $1.460.000$ intersections such that 250.000 tests are box enclosure ones and the rest $1.210.000$ are ray-solid tests - a decrease of roughly 80%. In the worst case, when an enclosure stretches across the entire view, the box enclosure will add additional operations of ray-box in-

tersections on top of completing all the ray-intersection checks. Nevertheless, ray-box tests are fast, and one could dismiss the additional costs of those operations. When this method is used in the context of CSG, this solution essentially turns into an efficient binary tree traversal [20].

We can also use many other types of enclosures; however, we choose box enclosures for their numerous advantages. First, one can define an abstract box by only two points (a minimum and maximum point). Because the enclosure definition lies inside every node in the CSG tree, we must ensure that we do not excessively increase the required memory per node. Second, boxes are arguably the tightest types of bounding volumes. Implying that if a ray-box intersection test is positive, there is a high probability the ray will too intersect the geometry inside of the bounding box. Lastly, applying boolean operations on bounding boxes is straightforward. Therefore, if an object is part of a CSG tree, but the precomputed bounding box is smaller than the object (e.g., the case of a subtraction operation), the rest of the geometry piercing outward will also be neglected when performing ray-composite intersection tests [20].

A bounding box is a rectangular parallelepiped defined by exactly two points (Figure 17). Each primitive, solid, and composite must be able to define its bounding box. For primitive cases, the bounding box is case-specific. For example, the bounding box of a primitive sphere of radius $r = 1$ located at center point $\vec{o} = (0, 0, 0)$ has a bounding box whose maximum point is (r, r, r) and minimum point $(-r, -r, -r)$. Solids are more complicated as they are composed of many primitives. Hence, one has to create a collapsed bounding box (a bounding box whose min and max coordinates are respectively $+\inf$ and $-\inf$) and slowly start inflating by the primitive's predefined boxes. The inflation step is as simple as checking if the value of a coordinate of the current bounding box is smaller or bigger than that of the primitive's bounding box and either picking the smallest or the greatest value depending on the point being checked. For instance, if our current bounding box has $min(0, 0, 0)$ and $max(1, 1, 3)$ and the current primitives bounding box has $min2(-1, -1, 1)$ and $max2(2, 2, 2)$ then the current values of the points of the inflated bounding box become $min(-1, -1, 0)$ and $max(2, 2, 3)$.

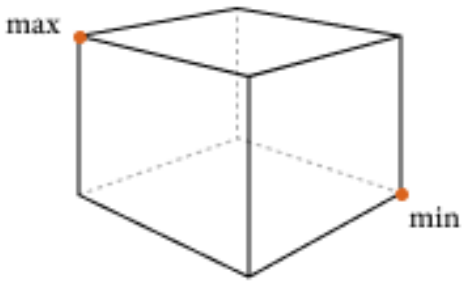
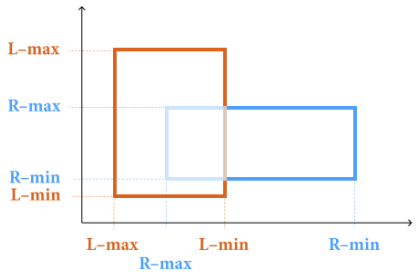


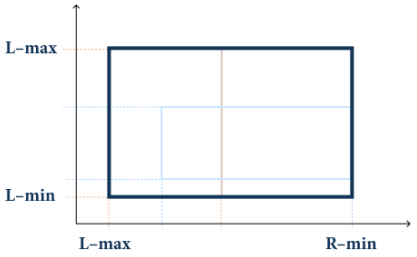
Figure 17: Bounding box.

Combining the boxes on the composite level is also very important to realize. We can achieve this trivially with the usual rules of algebra defined in the previous section. Though that doesn't hold for the difference operation as its results are not easily foreseeable, and the cost of analyzing the entire composition is counter-productive in this use case [20].

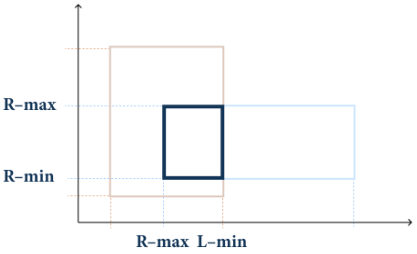
When dealing with the union operation, we select the smallest value from both boxes per coordinate for the minimum and vice-versa. In an intersection case, we pick the highest value from both boxes per coordinate for the minimum (opposite to the union) and the dual for the maximum. For the difference, we have previously mentioned that it's not possible to generalize using boolean algebra; therefore, we keep the minimum and maximum of the left box as we are sure that the result of the subtraction operation will never be bigger than the left geometry, $A - B \leq A$. Figure 18 shows the different operations on rectangles. The same logic holds for the three dimensional solids as we only check for an additional coordinate. Algorithm 18 defines the procedure for composite boxes [5].



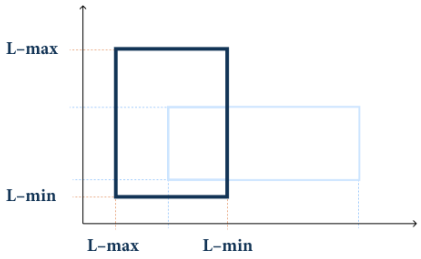
(a) Initial bounding boxes.



(b) Union of bounding boxes.



(c) Intersection of bounding boxes.



(d) Difference of bounding boxes.

Figure 18: Composite bounding boxes.

Algorithm 5: Composite solid box enclosure estimation algorithm.

Result: t intersection distance
initialization;
for i = *1, 2, 3* **do**
  if *Operator is* $\cup$ **then**
    min[i] = MIN(leftMin[i], rightMin[i]);
    max[i] = MAX(leftMax[i], rightMax[i]);
  if *Operator is* $\cap$ **then**
    min[i] = MAX(leftMin[i], rightMin[i]);
    max[i] = MIN(leftMax[i], rightMax[i]);
  if *Operator is* $-$ **then**
    min[i] = leftMin[i];
    max[i] = leftMax[i];
end

4.3 Binary Space Partitioning Trees

One of the most fundamental concepts in ray tracing is the use of spatial or hierarchical data structures built using binary space subdivision to efficiently search for objects in the scene [26]. A predominant concept in these data structures is binary space partitioning which refers to the successive subdivision of a scene's bounding box with planes until we reach termination criteria. The resulting data structure is called a binary space partition tree or a BSP tree. BSP trees offer the flexibility of using arbitrarily oriented planes to accommodate complex scenes and uneven spatial distributions. Therefore, in theory, BSP trees are a simple, elegant, and efficient solution to our visible-surface problems. In our implementation, we use a variant called KD-trees (we refer to KD-tree as BSP here). These are a more "restricted" type of BSP trees in which only axis-aligned splitting planes are allowed. These trees conform much better with computational advantages and memory needs but do not adapt very well to scene complexities. It is relatively easy to generate an inefficient binary tree with non-axis-aligned geometry (e.g., a long skinny cylinder oriented diagonally) [7]. All variations of the algorithms are generally composed of two fundamental parts, building and traversing the tree. How we choose these two core procedures tremendously affects the amount of acceleration achievable. We will discuss our building and traversing procedures and a custom traversal to find the furthest intersection. Because the main focus of the work doesn't align with the improvement of building or traversal procedures, the BSP algorithm is unoptimized. However, many algorithms such as surface area heuristic, local greedy SAH, automatic termination criteria, and many more have proved to optimize KD trees [30, 23].

4.3.1 Building BSP trees

The tree in our case is constructed recursively in a top-down manner, making local greedy decision about the choice of the splitting planes. We use axis-aligned bounding boxes in order to wrap the nodes. The split dimension is chosen using the current largest dimension (i.e., if the box is biggest in its x direction then we will pick that as our splitting plane.). The plane is then positioned at the spatial median of the dimension. Usually, this subdivision is performed until either the number of primitives in a single node falls below a predefined threshold or until the subdivision depth exceeds a certain maximum value. Both these stopping criteria could be provided by the user. To better illustrate the algorithm, we will utilize the simple two-dimensional kd-tree and the triangles in Figure 19. Each node in the tree represents a triangle, and each internal node represents an axis-aligned rectangular region with an axis-aligned plane that separates the regions of its two children.

4.3.2 Traversing BSP trees

A ray traverses a BSP tree by intersecting the ray with the split plane; therefore, giving a ray distance to the plane, allowing us to divide it into segments. The initial ray segment is computed by clipping the ray with the axis-aligned bounding box. We traverse a node if the ray segment overlays the node. Since the two-child nodes do not overlap, we can trivially classify which node is closer to the ray direction and traverse that node first. For the traversal algorithm, the children should be labeled as *near* and *far* child nodes, giving us three possible cases of traversal:

1. Ray goes through near child only. (Figure 20a)

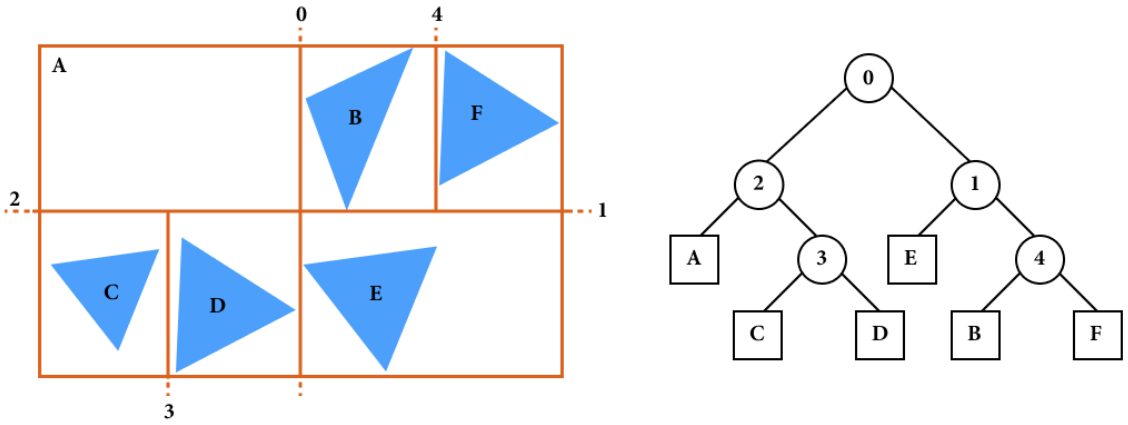
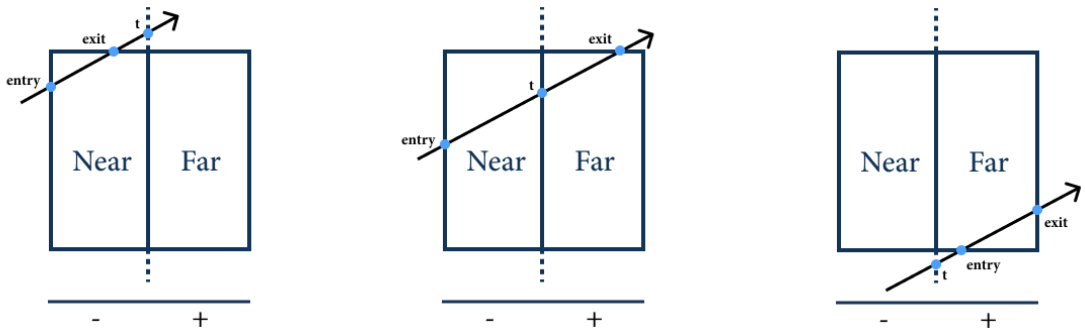


Figure 19: Simple scene with a few triangles and a corresponding tree. Leaves are boxes and inner nodes are circles.

2. Ray goes through far child only. (Figure 20c)
3. Ray goes through the near child first followed by the far child. (Figure 20c)

The near and far classification uses the direction of the ray and the position of the splitting plane. Therefore, it classifies the left node as near and the right node as far if the sign of the ray direction in the splitting axis is positive and vice versa if negative. Once the terminal nodes are reached, we can then search for the intersection of all the primitives in the node, if any.



(a) Ray goes through near child only.

(b) Ray goes through both children.

(c) Ray goes through near child only.

Figure 20: Ray traversal cases.

An important thing we must also achieve is to traverse the tree but search for the furthest intersection instead. In this case, the traversal remains mostly the same, except that in case where a ray hits both the near and far node, we start with the far node as that's what we're actually interested in. The rest of the modifications are all implemented in the leaf nodes where we instead look for out exit intersections.

4.4 Optimized CSG

With all three optimizations from Sections 4.1, 4.2, 4.3, we now have the building blocks for the optimized CSG algorithm. We can deconstruct this algorithm to a hierarchical

pipeline of 4 elements. On the very top, we have our entire scene enclosed with all the geometries in the scene in a BSP tree. We then have constructively constructed geometries inside of this scene represented using two other BSP trees for their respective left and right geometries[4]. These trees are then capable of efficiently retrieving the $[in, out]$ of their respective nodes and have bounding boxes which allow for quick tests of ray-solid intersections. Finally upon retrieval of the ray intersections, the minimal hit algorithm allows for efficient and robust classification of these intersections. This means that even when nesting constructively generated geometries, we are still able to hold definitions of each sub-tree for each sub-object therefore allowing for very fast evaluation of complex and nested geometries such that each solid is responsible for its own evaluation and can always feed the correct intersection information to its parents nodes. Therefore, skipping the step of gathering all evaluations and only processing them on the leaf nodes.

5 Evaluation of the results

There are three variants of the CSG method implemented in OpenRT. The first is the naive and brute force implementation which we refer to as $NaiCSG$. The second uses a binary space partition tree to solve the visible surface problem but still naively finds intersections inside the combinatorial geometry, which we will refer to as $BinCSG$. Lastly, we'll introduce our optimized algorithm, which uses a binary space partition tree on the outside (solving the visible surface problem) and also inside each composite geometry to direct the rays towards the correct geometries, which we will refer to as $OptiCSG$. Every algorithm is checked for each operation - meaning we carry a total of 9 simulations per test type. We conduct three main tests. First, we assess how the rendering time develops to the complexity of the geometry. In this case, the complexity of the geometry is the number of polygons in the sphere meshes. The second test is based on the reaction of the different algorithms to a different number of nested geometries while maintaining a relevantly similar viewport fill rate (how much of the image contains geometries). The third test explains how the spatial distribution of the scene affects the times for each of the algorithms. Therefore, helping us grasp how significant the viewport fill percentage changes each of these algorithms individually.

5.1 Geometry Complexity Tests

Starting with naive implementation, we can see that the increase in rendering in time is linear (Figure 21). This is expected because as we increase the complexity of the geometries, the only part that is affected is the loop in our ray intersection procedure inside of the composite solid.

[4] In case the constructive solid only contains single primitive elements, then the tree is a simple tree of depth 1

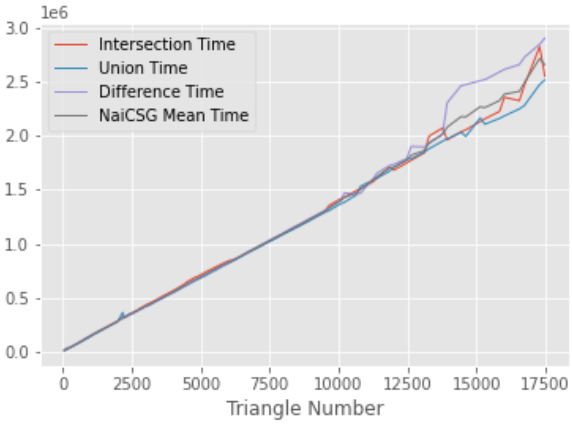


Figure 21: Rendering time of the naive algorithm after gradual increases in the number of triangles in the meshes for all three operations.

The second implementation which uses a binary tree in order to efficiently find objects in the scene (BinCSG) is also linear (Figure 22) except that here we save iterations over the classification loop since the algorithm will opt out of the solid-intersection computation in case the preliminary ray-box test is negative. Though, this is a big improvement over the linear algorithm, the time taken heavily depends on the amount filled by the bounding box of the geometry. Meaning, that if the bounding box were the take the majority of the viewport, we will notice that both variants perform similarly. Figure **XX** demonstrates how the rendering time of an identical scene using this implementation exponentially grows and becomes the same as the naive implementation. Figure **YY** compares the rendering time of NaiCSG and BinCSG when an object takes up the entirety of the viewport.

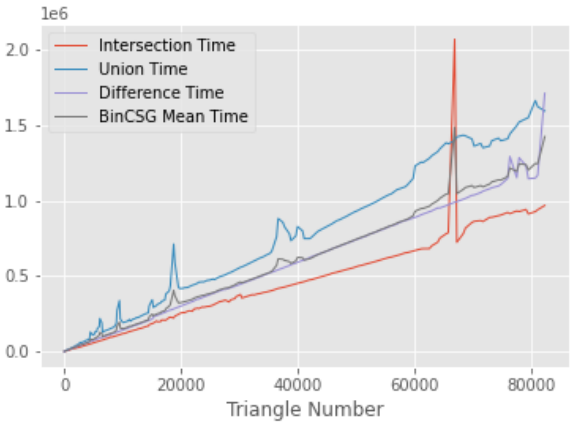


Figure 22: Rendering time of the BSP algorithm after gradual increases in the number of triangles in the meshes for all three operations.

The final implementation, OptimCSG, uses a BSP tree to find objects in the scene and represents each mesh in its separate binary tree. Therefore, even when a ray enters the first check through the solid, we do not naively iterate over all primitives to find the intersection but rather try to traverse their respective trees. This also means that when a ray only intersects one of the meshes, we don't check for the other as its ray-box test

is negative. The time complexity here is that of traversing the binary tree ($n \log n$) with an additional constant which we can omit. Consequently, we can explain the logarithmic nature of the Figure 23. We can also notice here the optimization of the minimal hit classification algorithm. As previously mentioned in Section 4.1.3, the difference operation is the most difficult to enhance; therefore, explaining the slight increase in rendering time. The union and intersection cases are somewhat similar. The generated geometry and bounding box from the intersection operation is also smaller than that of the union, which reveals why the intersection operation demands the least cost of time.

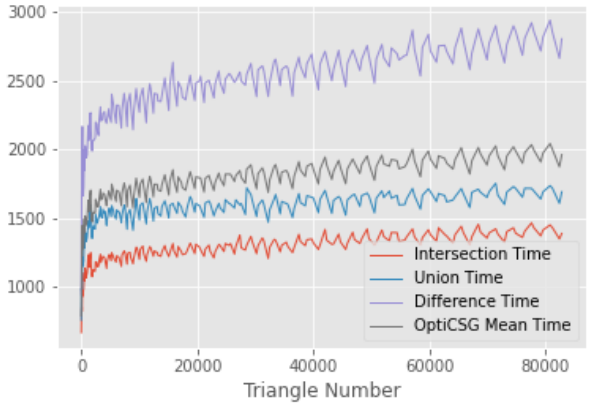


Figure 23: Rendering time of the optimized algorithm after gradual increases in the number of triangles in the meshes for all three operations.

Figure **??** compares the variants to the same payload with a relatively low viewport fill rate. We can see how box enclosures can help bring down the rendering time significantly. We also notice how the OptiCSG algorithm outperforms all of the other variants. Figure **XX** shows how the curves of NaiCSG and BinCSG are seemingly identical after entirely filling the view port. Table **XX** shows the mean time for conducting these tests per operation and the achieved speedup.

Table 4: Boolean operations table

Variant	View Port Fill	Mean Time
NaiCSG	33.33%	895725.64ms
NaiCSG	66.66%	TBD
NaiCSG	99.99%	TBD
BinCSG	33.33%	97903.80ms
BinCSG	66.66%	TBD
BinCSG	99.99%	TBD
OptiCSG	33.33%	1603.03ms
OptiCSG	66.66%	TBD
OptiCSG	99.99%	TBD

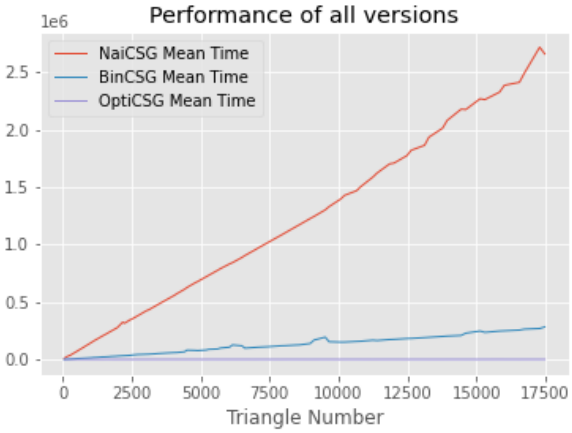


Figure 24: Average rendering time of each of the algorithms after gradual increases in the number of triangles in the meshes.

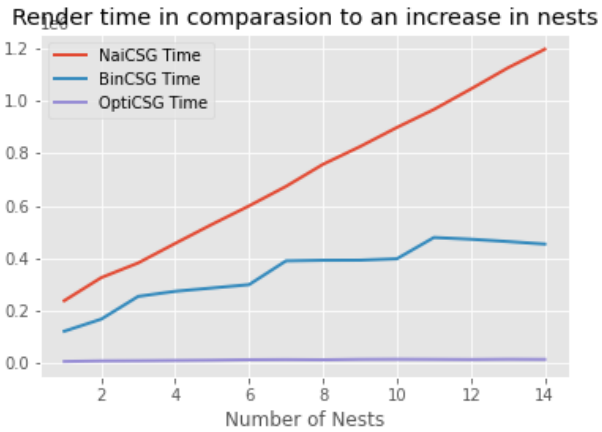


Figure 25: Rendering time of all 3 algorithms following increases nesting multiple constructive geometries.

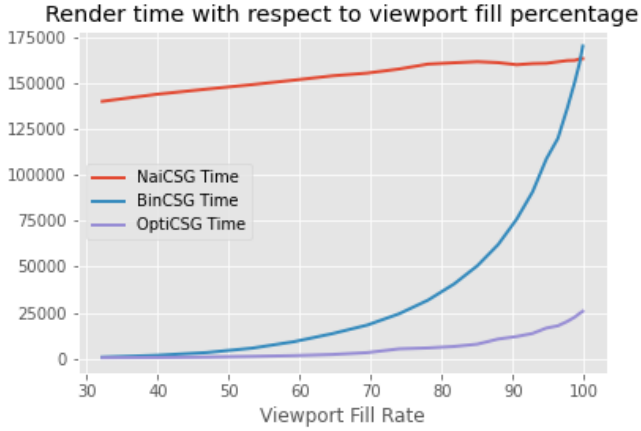


Figure 26: Closure kX of a subset X.

Summarize the main aspects and results of the research project. Provide an answer to the research questions stated earlier.

(target size: 1/2 page)

6 Conclusion

References

[1] Timo Aila and Samuli Laine. “Understanding the Efficiency of Ray Traversal on GPUs”. In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG ’09. New Orleans, Louisiana: Association for Computing Machinery, 2009, 145–149. ISBN: 9781605586038. DOI: 10.1145/1572769.1572792. URL: https://doi.org/10.1145/1572769.1572792.

[2] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. 4th. USA: A. K. Peters, Ltd., 2018. ISBN: 0134997832.

[3] Arthur Appel. “Some Techniques for Shading Machine Renderings of Solids”. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS ’68 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1968, 37–45. ISBN: 9781450378970. DOI: 10.1145/1468075.1468082. URL: https://doi.org/10.1145/1468075.1468082.

[4] S. Cameron. “Efficient bounds in constructive solid geometry”. In: *IEEE Computer Graphics and Applications* 11.3 (1991), pp. 68–74. DOI: 10.1109/38.79455.

[5] S. Cameron. “Efficient Bounds in Constructive Solid Geometry”. In: *IEEE Computer Graphics and Applications* 11.03 (1991), pp. 68–74. ISSN: 1558-1756. DOI: 10.1109/38.79455.

[6] Kuang-Hua Chang. *Chapter 3 - Solid Modeling*. Ed. by Kuang-Hua Chang. Boston, 2015. DOI: https://doi.org/10.1016/B978-0-12-382038-9.00003-X. URL: https://www.sciencedirect.com/science/article/pii/B978012382038900003X.

[7] Thiago Ize, Ingo Wald, and Steven Parker. “Ray tracing with the BSP tree”. In: Sept. 2008, pp. 159–166. ISBN: 978-1-4244-2741-3. DOI: 10.1109/RT.2008.4634637.

[8] Peter Keenan. *Geographic Information Systems*. Ed. by Hossein Bidgoli. New York, 2003. DOI: https://doi.org/10.1016/B0-12-227240-4/00077-0. URL: https://www.sciencedirect.com/science/article/pii/B0122272404000770.

[9] Andrew Kensler. *Ray Tracing CSG Objects Using Single Hit Intersections*. English. Oct. 2006. URL: http://xrt.wdfiles.com/local--files/doc\%3Acsg/CSG.pdf.

[10] Florian Kirsch and Jürgen Döllner. “Rendering Techniques for Hardware-Accelerated Image-Based CSG.” In: Jan. 2004, pp. 221–228.

[11] Reinhard Klette and Azriel Rosenfeld. *CHAPTER 7 - Curves and Surfaces: Topology*. Ed. by Reinhard Klette and Azriel Rosenfeld. San Francisco, 2004. DOI: https://doi.org/10.1016/B978-155860861-0/50009-2. URL: https://www.sciencedirect.com/science/article/pii/B9781558608610500092.

[12] Alistair H. Lachlan, Marian Srebrny, and Andrzej Zarach. *Set theory and hierarchy theory V: Bierutowice, Poland, 1976*. Springer-Verlag, 1977.

[13] Sylvain Lefebvre. “IceSL: A GPU Accelerated CSG Modeler and Slicer”. In: *AEFA’13, 18th European Forum on Additive Manufacturing*. Paris, France, June 2013. URL: https://hal.inria.fr/hal-00926861.

[14] Maynard J. Mansfield. *Introduction to topology*. R.E. Krieger, 1987.

[15] M. E. Newell, R. G. Newell, and T. L. Sancha. “A Solution to the Hidden Surface Problem”. In: *Proceedings of the ACM Annual Conference - Volume 1*. ACM ’72. Boston, Massachusetts, USA: Association for Computing Machinery, 1972, 443–450. ISBN: 9781450374910. DOI: 10.1145/800193.569954. URL: https://doi.org/10.1145/800193.569954.

[16] Stephen M. Pizer et al. *6 - Object shape representation via skeletal models (s-reps) and statistical analysis*. Ed. by Xavier Pennec, Stefan Sommer, and Tom Fletcher. 2020. DOI: https://doi.org/10.1016/B978-0-12-814725-2.00014-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128147252000145.

[17] *Ray tracing primitives*. URL: https://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node2.html (visited on 04/05/2021).

[18] A. Requicha and R. Tilove. "Mathematical Foundations of Constructive Solid Geometry: General Topology of Closed Regular Sets". In: 1978.

[19] Aristides G. Requicha. "Representations for Rigid Solids: Theory, Methods, and Systems". In: *ACM Comput. Surv.* 12.4 (Dec. 1980), 437–464. ISSN: 0360-0300. DOI: 10.1145/356827.356833. URL: https://doi.org/10.1145/356827.356833.

[20] Scott D Roth. "Ray casting for modeling solids". In: *Computer Graphics and Image Processing* 18.2 (1982), pp. 109–144. ISSN: 0146-664X. DOI: https://doi.org/10.1016/0146-664X(82)90169-1. URL: https://www.sciencedirect.com/science/article/pii/0146664X82901691.

[21] Scratchapixel. *Rasterization: a Practical Implementation*. Jan. 2015. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation.

[22] Bin Sheng et al. "Efficient non-incremental constructive solid geometry evaluation for triangular meshes". In: *Graphical Models* 97 (2018), pp. 1–16. ISSN: 1524-0703. DOI: https://doi.org/10.1016/j.gmod.2018.03.001. URL: https://www.sciencedirect.com/science/article/pii/S1524070318300067.

[23] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. "Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes". In: *Computer Graphics Forum* 26.3 (2007), pp. 395–404. DOI: https://doi.org/10.1111/j.1467-8659.2007.01062.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01062.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01062.x.

[24] Allan D. Spence and Yusuf Altintas. *Modeling Techniques and Control Architectures for Machining Intelligence*. Ed. by C.T. Leondes. 1995. DOI: https://doi.org/10.1016/S0090-5267(06)80031-9. URL: https://www.sciencedirect.com/science/article/pii/S0090526706800319.

[25] Sebastian Steuer. *Methods for Polygonalization of a Constructive Solid Geometry Description in Web-based Rendering Environments*. Dec. 2012. URL: https://www.en.pms.ifi.lmu.de/publications/diplomarbeiten/Sebastian.Steuer/DA_Sebastian.Steuer.pdf.

[26] Kelvin Sung and Peter Shirley. "VI.1 - RAY TRACING WITH THE BSP TREE". In: *Graphics Gems III (IBM Version)*. Ed. by DAVID KIRK. San Francisco: Morgan Kaufmann, 1992, pp. 271–274. ISBN: 978-0-12-409673-8. DOI: https://doi.org/10.1016/B978-0-08-050755-2.50061-0. URL: https://www.sciencedirect.com/science/article/pii/B9780080507552500610.

[27] R.B. Tilove. "A study of GEOMETRIC SET-MEMBERSHIP CLASSIFICATION". en. (Master's thesis, University of Rochester, 1977.

[28] *Visualization of a polygon mesh*. Jan. 2021. URL: https://en.wikipedia.org/wiki/Polygon_mesh.

[29] H. B. Voelcker and A. A. G. Requicha. “Geometric Modeling of Mechanical Parts and Processes”. In: *Computer* 10.12 (1977), pp. 48–57. DOI: 10.1109/C-M.1977.217601.

[30] Ingo Wald and Vlastimil Havran. “On Building Fast kd-trees for Ray Tracing, and on Doing that in O(N log N)”. In: *Symposium on Interactive Ray Tracing* 0 (Sept. 2006), pp. 61–69. DOI: 10.1109/RT.2006.280216.

[31] *XRT Renderer*. URL: http://xrt.wikidot.com/doc:csg.