JACOBS
UNIVERSITY

Accelerated Ray Tracing of Constructive Solid Geometry

by

Otmane Sabir

Bachelor Thesis in Computer Science

Submission: May 10, 2021

Supervisor: Prof. Dr. Sergey Kosov

Jacobs University Bremen | Department of Computer Science and Electrical Engineering

Family Name, Given/First Name	Sabir Otmane
Matriculation number	30002035
What kind of thesis are you submitting:	Bachelor-Thesis

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

Acknowledgements

The journey towards this thesis has been circuitous. Its completion is thanks to the special people who challenged, supported, directed, and drove me along the way.

I am tremendously fortunate for my supervisor, Professor Dr. Sergey Kosov, whose expertise was invaluable in the progress of this research. Your theoretical and practical experience remains a central part in understanding the research, architecting the implementation, and analyzing the results. Your stellar feedback pushed me to sharpen my thinking and opened up different perspectives concerning several matters. I am grateful for the inspiration you have been able to instill in all steps of this study.

Additionally, I am deeply indebted to my parents, my §sister, and my grandmother for their wise guidance, thoughtful words, and tender care. You are always there for me. Finally, I could not have completed this dissertation without the support of my friends, who continuously taught me how to embrace change and strive for the best.

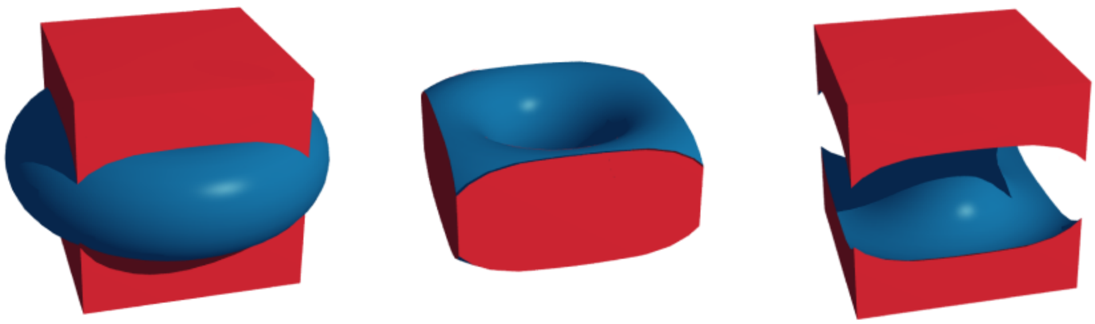
Abstract

This thesis report presents constructive solid geometry using ray tracing as a way of creating complex geometries for solid modeling. Solid objects are modeled using different implicit and explicit geometries (i.e., spheres, tori, boxes) with boolean set operators. By virtue of its simplicity, ray tracing constructive solid geometry is reliable and expandable. The most challenging issue is finding the visible geometry intersections in the fastest and most efficient way. So issues of adequacy and efficiency are addressed here, and we propose solutions providing significantly faster rendering of CSG. We validate the performance by comparing various implementations of the algorithm. The paper has two major parts. In the first, we present the generic method of describing constructively generated geometry. The second is devoted to a study of acceleration of ray tracing constructive solid geometry. We expose algorithms that are both comprehensive and efficient, and the results of the performance are shared.

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 torus 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 other features. This problem is known as the hidden surface problem [16]. 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 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 [22].

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 utilizes 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, 9, 12, 17, 20, 26]. Many solutions exist that enable the simple conversion between these geometric formats [27]. However, there are predominantly two different representations in most geometric modeling systems [20]: 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. 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 [23], 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 primitive geometries and 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 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

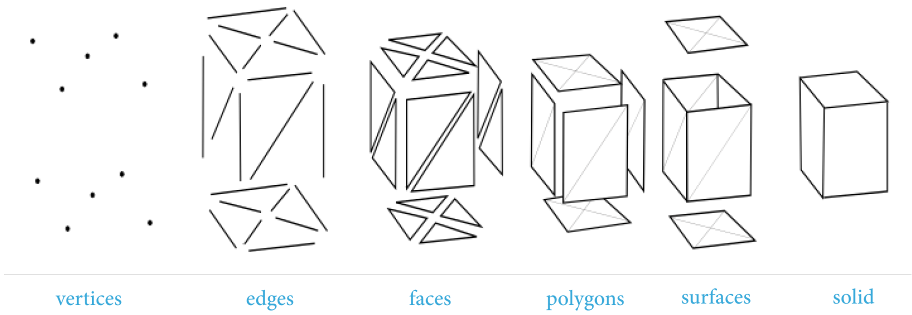


Figure 2: Sample BREP of a 3D hyper-rectangle [30]

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.

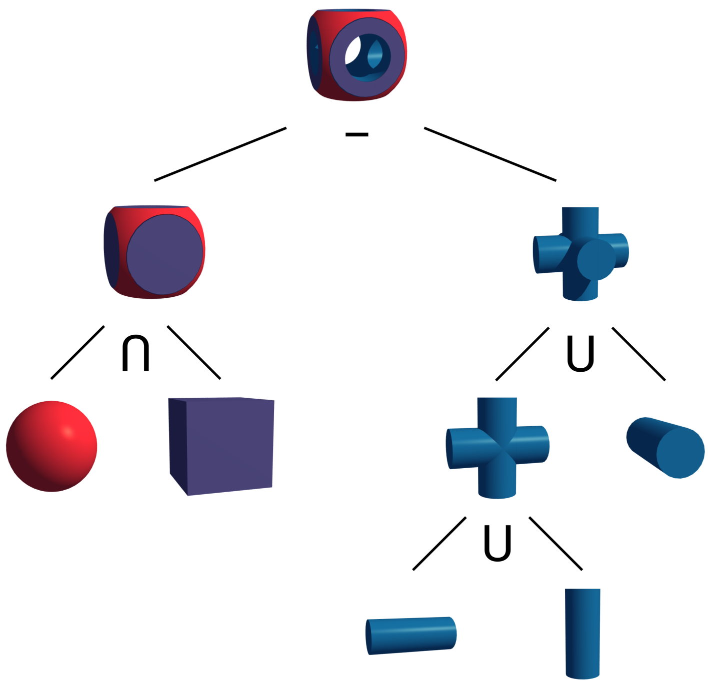


Figure 3: Sample representation of a CSG tree.

[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. Conclusions & Future Works.

The first section was the previous introduction laying a foundation for 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 for 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 of 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 uses a binary space partitioning 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 $OptimCSG$. 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 viewport. Additionally, we conduct a test to check the number of ray tests conducted per pixel per variant. The final test computes the time variations after increasing the number of nested geometries present in the composite solid while crucially maintaining a consistent viewport 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 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 [31] 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 [19]. 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 [21] 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 [10]. 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 later proved to be a solution that does not gracefully handle edge cases, especially for the difference and intersection operations [33].

A "slicing" approach is also proposed in [14]. 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 [5] 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 [11]. 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

OpenRT is used to perform our investigations. OpenRT is a C++ free open-source ray tracing library [23]. OpenRT has a fast ray-tracing engine and all of the functionality we need to describe the geometry, shaders, lights, cameras, and samplers. OpenRT also has elegant binary space portioning algorithm definitions, which we will discuss in Section 4.3.

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 all of the intersections with geometry for correct classification. With knowledge of all the information in the scene - essentially the camera model and the solids - an evaluation of these intersections 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 the intersection.
$prim$ = a pointer holding surface information of the intersected primitive.

We can distinguish two types of ray intersections [22]. 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 [18]. 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 intersection checks.

```
Result: arr array of intersections
i = 0;
for every primitive in the solid do
    solve the ray-primitive equations;
    if intersection exists then
        arr[i] = current intersection;
        i = i + 1;
end
```

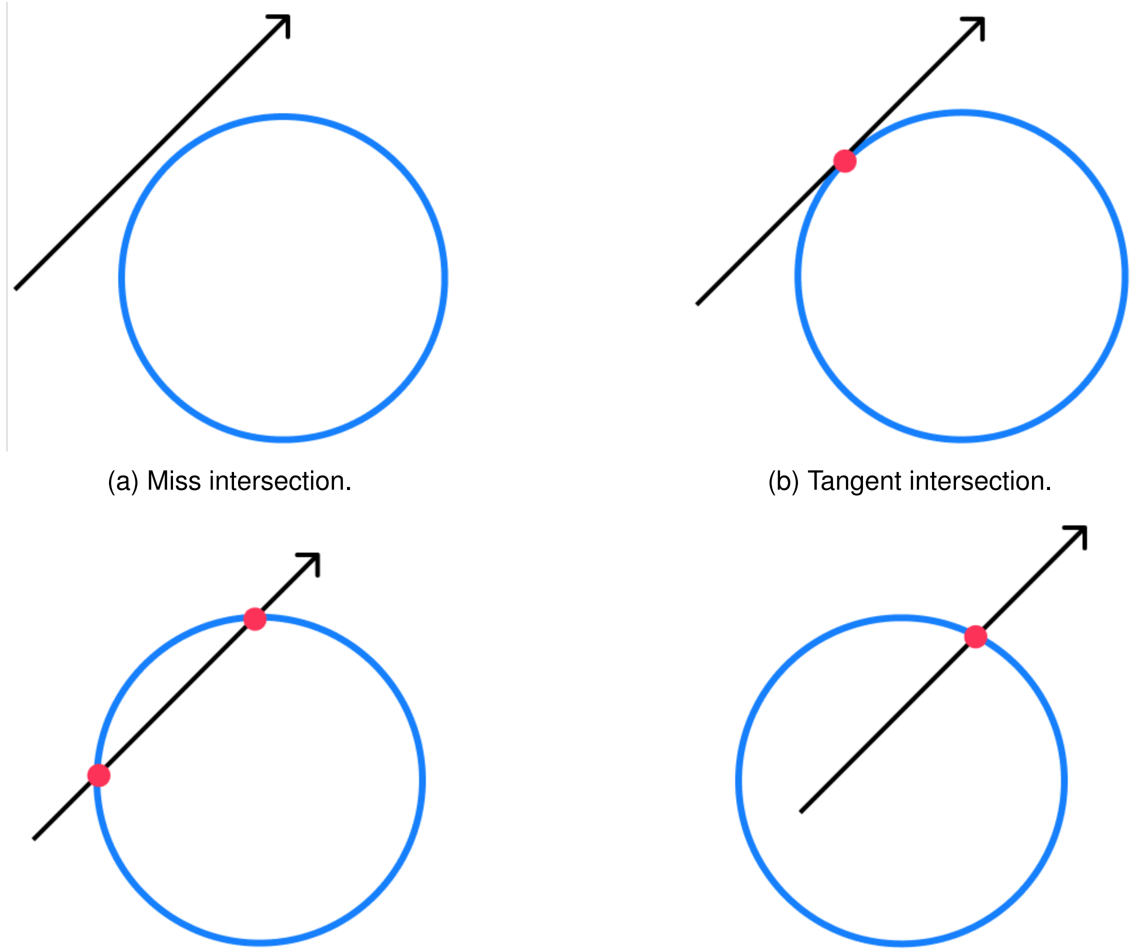
The ray-solid intersection test has four possible outcomes:

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 cases are more intricate to determine as we need to understand whether the intersection is entering or exiting the solid. We resolve 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 [19]. 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 [19], [29],[13], and many other



(a) Miss intersection.

(b) Tangent intersection.

(c) Complete intersection.

(d) Inside intersection.

Figure 4: Different ray intersection cases on a disk.

resources. Hence, I will be mainly focusing on definitions and properties that interest 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. [15]*

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

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

Property 3.2. *Union and intersection operations are distributive over themselves and each other. [15]*

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

Property 3.3. *The empty set $\emptyset$ and the universe W are identity elements for the union and intersection operators. [15]*

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

Property 3.4. *The complement, denoted c, satisfies [15]:*

$$X \cup cX = W$$

$$X \cap cX = \emptyset$$

Definition 3.2 (Boolean Algebra)**.** 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 [19].

3.2.2 Topological Spaces

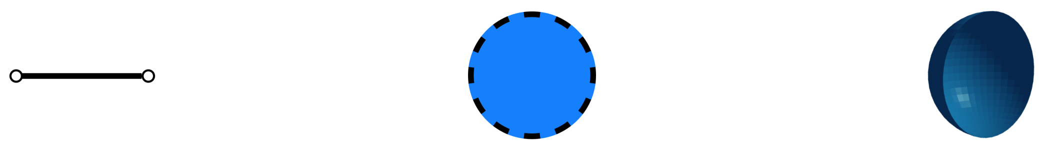
Topological spaces are a generalization of metric spaces in which the notion of "nearness" is introduced but not in any quantifiable way that requires a direct distance definition [15].

Definition 3.3. 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.[15]*

Property 3.6. *The intersection of a finite number of open sets is an open set. [15]*

Property 3.7. *The union of any collection of open sets is an open set. [15]*



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

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

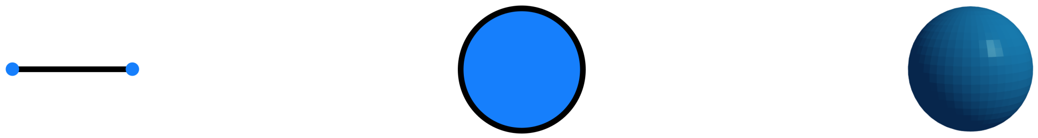
3.2.3 Closed Sets

Definition 3.4. A subset X of a topological space (W,T) is closed if its complement is open[2]. Closed sets hold the properties (3.8), (3.9), and (3.10) 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. [15]*

Property 3.9. *The intersection of a finite number of closed sets is a closed set. [15]*

Property 3.10. *The union of any collection of closed sets is a closed set. [15]*



(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. [19] (Figure 7)

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. [19] (Figure 7)

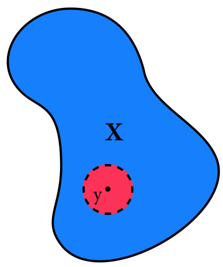


Figure 7: Interior point y on a subset X. The disc around y is the neighborhood of y.

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]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) [15].

boundary points of X. [19] (Figure 8)

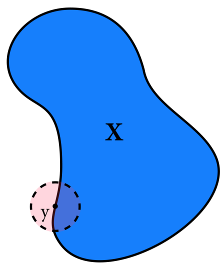


Figure 8: Boundary point y on a subset X.

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. [19] (Figure 9)

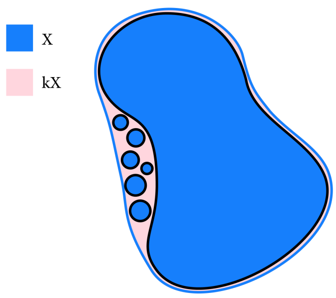


Figure 9: Closure kX of a subset X.

3.2.8 Regularity

Definition 3.9 (Regularity)**.** The regularity of a subset X of W, denoted rX, is the set of $rX = kiX$. [15]

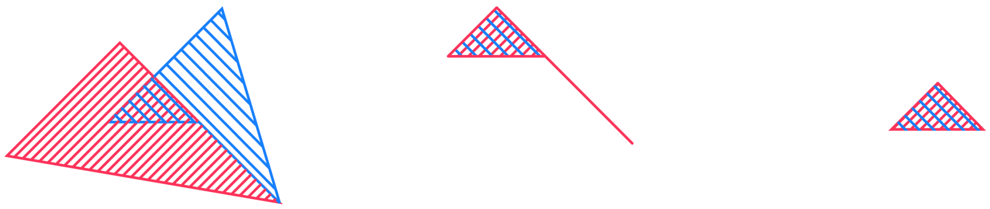
Definition 3.10 (Regular Set)**.** A set X is regular if $X = rX$, i.e. if $X = kiX$. [15] (Figure 10)

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 [29]. 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). [15]

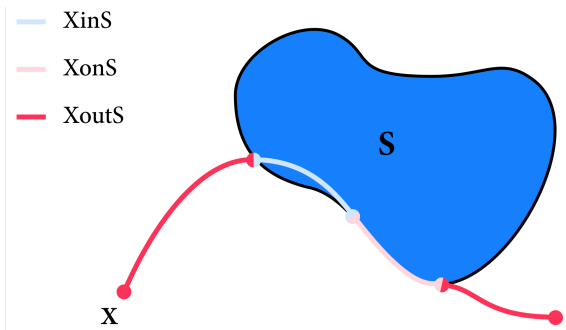


Figure 11: Membership classification function.

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. [19]

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 instances of f followed by a combine, g, step. When S is a primitive, 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.

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

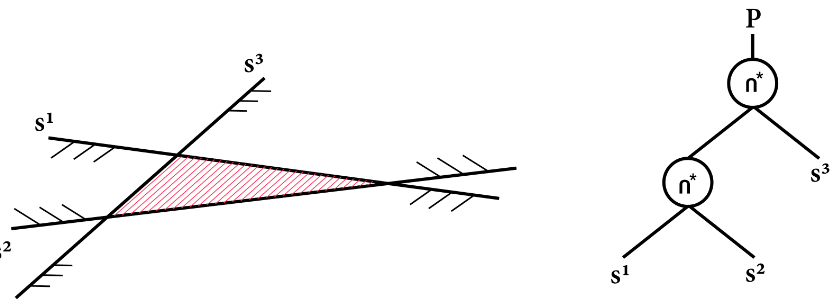


Figure 12: A constructive representation of a polygon P using half-planes. The tree on the right is the constructive geometry representation.

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.

To customize this general definition to be used in a specific domain, one must design the classification procedure, eM, and the combine procedure. The next section discusses both these procedures.

3.3 Ray classification

Given a ray and a solid composition tree, our procedure needs to classify the ray to the solid and return the classification to the caller. As previously defined in Section 3.1, the classification of a ray to a solid is the information describing all ray-solid intersections. The procedure starts at the top of the solid composition tree, recursively descends to the terminal nodes, classifies the ray to the primitives, then returns the array combining the classifications of the left and right subtrees. On the node level, this results in an array containing all possible intersections of a ray with the geometries in its left and right children. We must then sort each of these ray intersections by the distance to the ray origin and label them as entering or exiting. We finally scan through this array and apply the boolean algebra rules in Table 2. (Figure 13).

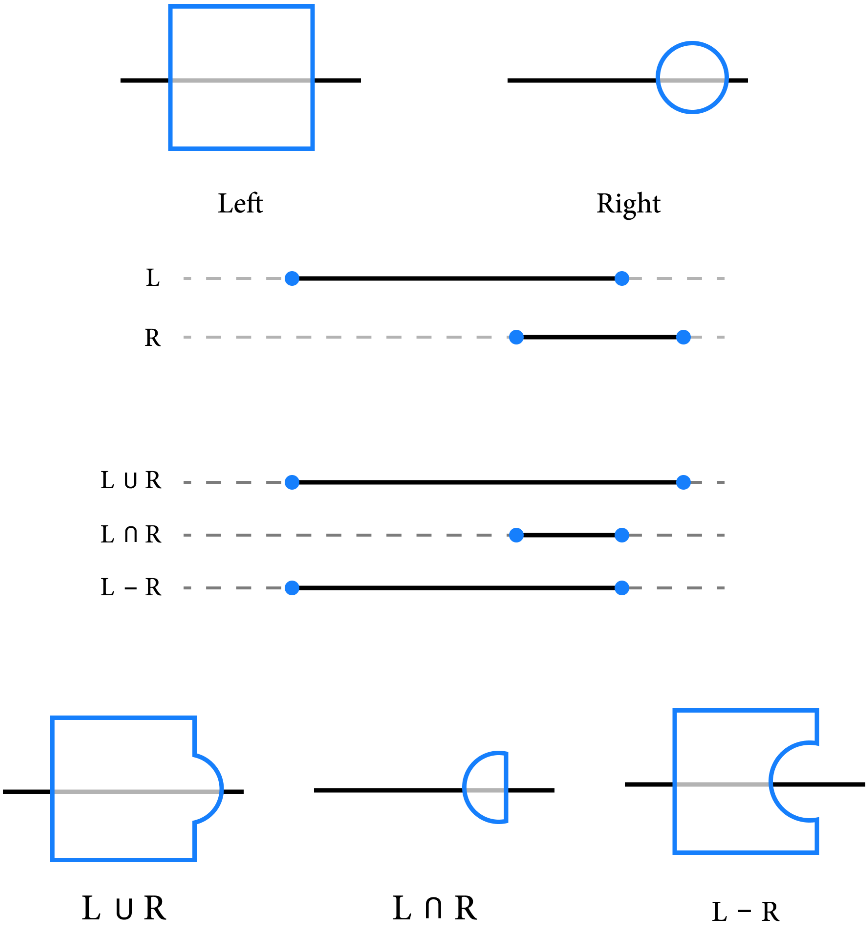


Figure 13: Example of combining ray classifications.

Table 2: Boolean operations table

Set Operator	Left Solid	Right Solid	Composite
$\cup$	*in*	*in*	*in*
	in	*out*	*in*
	out	*in*	*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 Optimization

In this section, we will introduce the state-of-the-art CSG algorithm that is implemented in the OpenRT. 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 a minimal hit classification algorithm, box enclosures, and how simple techniques such as "early-outs" can increase performance. Additionally, we discuss the binary space partitioning indexing structure for faster traversal of complex scenes and geometry. Finally, we will put it all together in our version of the CSG algorithm.

4.1 Minimal hit CSG classification

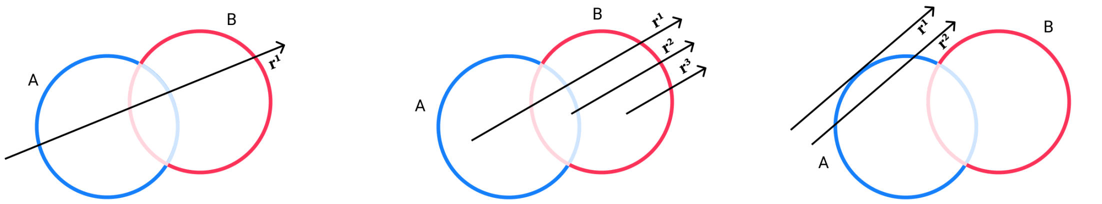
What we have introduced in Section 3.3 is the typical approach to rendering CSG. However, this method could be very costly as we nest more geometries in the tree and require lots of memory to store, classify, and combine a long chain of operations and primitives. Additionally, the algorithm could also perform additional checks when used in combination with BREPs. Therefore, we introduce a different approach which we refer to as minimal hit CSG classification. The method described here computes intersections with CSG objects using the single nearest intersections whenever possible. Though a relatively similar algorithm has been introduced in [10], it was proven in [33] to not be functional for the intersection and difference operations. The following implementation addresses those issues and adds a few optimizations to the classification code. The general idea can be conceived as a simple finite state machine. First, we investigate the closest intersections with both solids A and B. We later classify those evaluations into three potential states: enter, exit, or miss. The enter and exit cases are checked using the surface normal computation from before. The "miss" case is when the intersection distance remains to be the default value. Then depending on the operation, we evaluate the states and decide if we can return one of them. If so, then we're satisfied with the procedure. If not, we move the origin of the ray to the current viable intersection point and try the same over again. There are a few sub-functions that we must also define in this case before introducing the general algorithm (Table 3).

Table 3: Sub-procedures

ReturnClosest	Returns the closest of both.
ReturnFurthest	Returns the furthest of both.
IfXCloserReturn	Returns X if closer.
IfXFurtherReturn	Returns X if further.
IfXCloserReturnFlip	Returns X if closer and flips its normal.
AdvanceToXLoop	Sets the ray origin to intersection at X then loops.
AdvanceToClosestLoop	Sets the ray origin to the closest of both intersection then loops

4.1.1 Union Classification

Consider the case of 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.



(a) Ray goes through both spheres.

(b) Ray is inside one of the spheres.

(c) Ray misses one of the spheres.

Figure 14: Union ray classification cases.

For the case where the ray enters both spheres (Figure 14a), our procedure would first get the closest intersections with A and B. Both these intersections would be classified as enter states; therefore, we must only find $MIN(A, B)$ 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. If both A and B's states are a miss, we return a miss. Otherwise, if only one of them is a miss, we return the other regardless if it's an enter or exit.

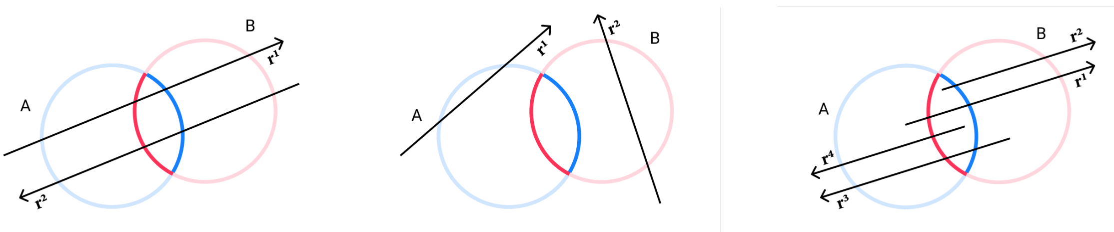
The last set of cases arise when the ray is shot from the interior of the spheres. This is more intricate since we have to teach our ray tracer to neglect the inner sides and only get the outer sides. If the first evaluation returns enter for B and exit for A, then we must check which one of them is closer. If $A < B$, then we return A. Otherwise, we move our origin to B and start the procedure again. If it is the opposite, then we perform the previous logic but we permute A and B. The final case is when the ray exits both A and B. Here, we return $MAX(A, B)$. Algorithm 2 shows the pseudocode for the union logic.

Algorithm 2: Minimal hit classification for union.

Result: Intersection Point
while *true* **do**
  min_A = intersectMin(A);
  min_B = intersectMin(B);
  $state_A$ = classify(min_A);
  $state_B$ = classify(min_B);
  if $state_A == miss$ && $state_B == miss$ **then**
    return $miss$;
  if $state_A == miss$ **then**
    return min_B;
  if $state_B == miss$ **then**
    return min_A;
  if $state_A == state_B$ **then**
    if $state_A == enter$ **then**
      ReturnClosest(min_A, min_B);
    if $state_A == exit$ **then**
      ReturnFurthest(min_A, min_B);
  if $state_A == enter$ && $state_B == exit$ **then**
    IfXCloserReturn(min_B);
    AdvanceToXLoop(min_A);
  if $state_A == exit$ && $state_B == enter$ **then**
    IfXCloserReturn(min_A);
    AdvanceToXLoop(min_B);
end

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 or B classify as misses (Figure 15a). By definition, the intersection is the shared area; therefore, if the ray misses one of the solids, we can already evaluate this as a miss.

The second case is when they both have the same classification. If both return an exit state, then we take the closest of both. However, if they both return an enter we either advance to A or B depending on which one is closest (Figure 15b).

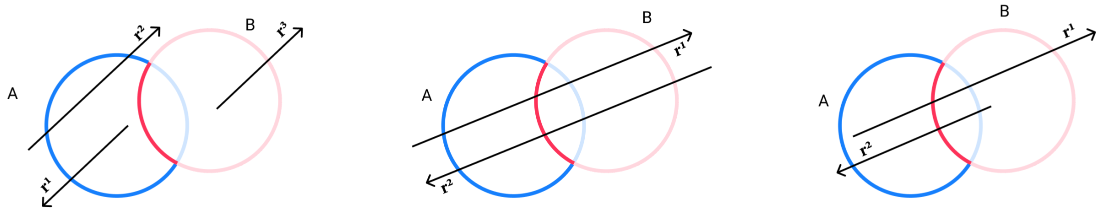
The final case is when the states are not a miss and also different from each other. If A is an enter state while B is an exit state, then we return A if it's closer or moves the ray origin to B and advance. We perform the opposite if A is exit and B is entered. Algorithm 3 shows the pseudocode for the intersection logic.

Algorithm 3: Minimal hit classification for the intersection.

```
Result: Intersection Point
while true do
    min_A = intersectMin(A);
    min_B = intersectMin(B);
    state_A = classify(min_A);
    state_B = classify(min_B);
    if state_A == miss || state_B == miss then
        return miss;
    if state_A == state_B then
        if state_A == enter then
            AdvanceToClosestLoop(min_A, min_B);
        if state_A == exit then
            ReturnClosest(min_A, min_B);
    if state_A == enter && state_B == exit then
        IfXCloserReturn(min_A);
        AdvanceToXLoop(min_B);
    if state_A == exit && state_B == enter then
        IfXCloserReturn(min_B);
        AdvanceToXLoop(min_A);
end
```

4.1.3 Difference Classification

The difference operation is not commutative; 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. If the ray only misses A or both, then we consider this a miss. If the ray only misses B, we return A regardless of entering or exit.

The second case is when they both have the same classification. If both return an exit state we return B if closer and flip it's normal, otherwise, we advance the ray origin to B. However, if they both return an entry we return A if it's closer or advances to the hit point of B.

The last case is when the classifications are different from each other. If the ray enters A and exits B, we return B if it's closer or advances to A. However, if the classifications are the opposite, we advance to whichever is closer and continue. Algorithm **??** shows the pseudocode for the different logic.

Algorithm 4: Minimal hit classification for the difference.

```
Result: Intersection Point
while true do
    min_A = intersectMin(A);
    min_B = intersectMin(B);
    state_A = classify(min_A);
    state_B = classify(min_B);
    if state_A == miss then
        return miss;
    if state_B == miss then
        return min_A;
    if state_A == enter && state_B == enter then
        IfXCloserReturn(min_A);
        AdvanceToXLoop(min_B);
    if state_A == exit && state_B == exit then
        IfXCloserReturnFlip(min_B);
        AdvanceToXLoop(min_A);
    if state_A == enter && state_B == exit then
        AdvanceToClosestLoop(min_A, min_B);
    if state_A == exit && state_B == enter then
        IfXCloserReturnFlip(min_A);
        return min_B
end
```

4.2 Bounding Boxes

Bounding boxes are the easiest way to cut down on the number of ray intersection operations and reduce overall rendering time [24]. Let us imagine 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 lessening 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 intersections on top of completing all the ray-intersection checks. However, ray-box tests are fast, and one could clear 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 [21].

We can also use many other types of enclosures; however, we choose box enclosures for their numerous benefits. 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 likelihood the ray will too intersect the geometry inside of the bounding box. Lastly, applying boolean operations on bounding boxes is straightforward [21]. Therefore, in the case of operations with less voluminous geometry, parts of the initial primitives piercing outside of the composite's bounds are also neglected.

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 and minimum points are (r, r, r) and $(-r, -r, -r)$. 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 gradually start inflating using 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 $max(2, 2, 3)$.

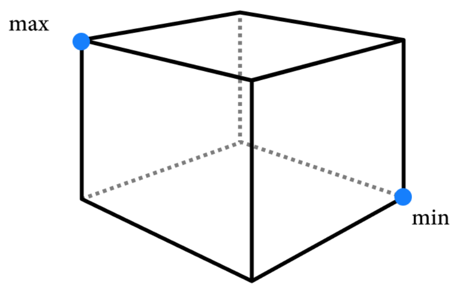
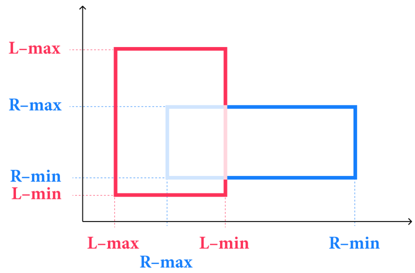
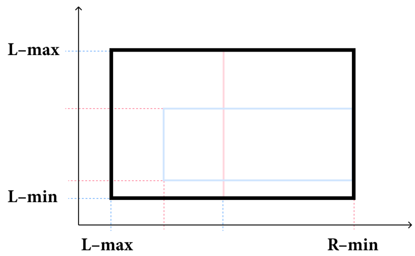


Figure 17: Bounding box.

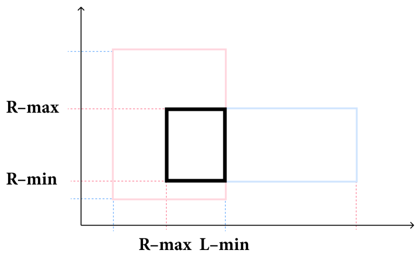
Combining the boxes on the composite level is also very important to realize. We can achieve this trivially with the usual rules of algebra defined in the previous section. Though that doesn't hold for the difference operation as its results are not easily foreseeable, and the cost of analyzing the entire composition is counter-productive in this case [21]. When dealing with the union operation, we select the smallest value from both boxes per coordinate for the minimum and vice-versa. For the intersection operation, we pick the highest value from both boxes per coordinate for the minimum - opposite to the union. 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 - if A and B are closed sets in E^n then $A - B \leq A$. 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 5 defines the procedure for composite boxes [21].



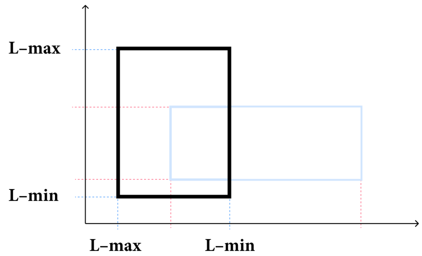
(a) Initial bounding boxes.



(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: Composite bounding box
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 spatial or hierarchical data structures built using binary space subdivision to efficiently search for objects in the scene [28]. 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 - which we refer to 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. Because the main focus of the work doesn't align with the improvement of building or traversal procedures, the BSP algorithm remains as is. However, many algorithms such as surface area heuristic, local greedy SAH, automatic termination criteria, and many more have proved to optimize KD trees [32, 25].

4.3.1 Building BSP trees

The tree is constructed recursively in a top-down manner, making a local greedy decision about the splitting planes. We use axis-aligned bounding boxes to wrap the nodes. We choose the split dimension using the current largest dimension (i.e., if the box is biggest in its x axis then we will pick that as our splitting plane). We then position the plane at the spatial median of the dimension. The subdivision is performed until either the number of primitives in a single node falls below a predefined threshold or the tree depth exceeds a maximum value. The user provides these stopping criteria. 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.

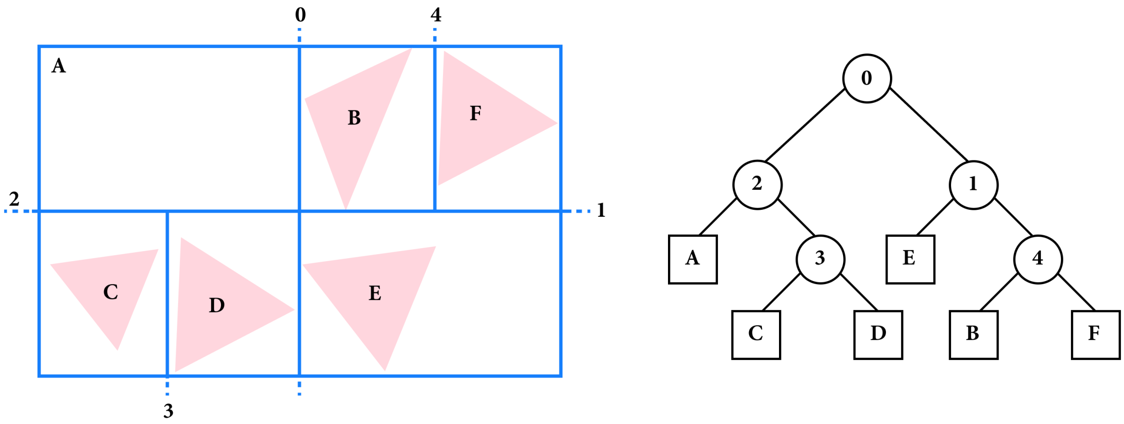


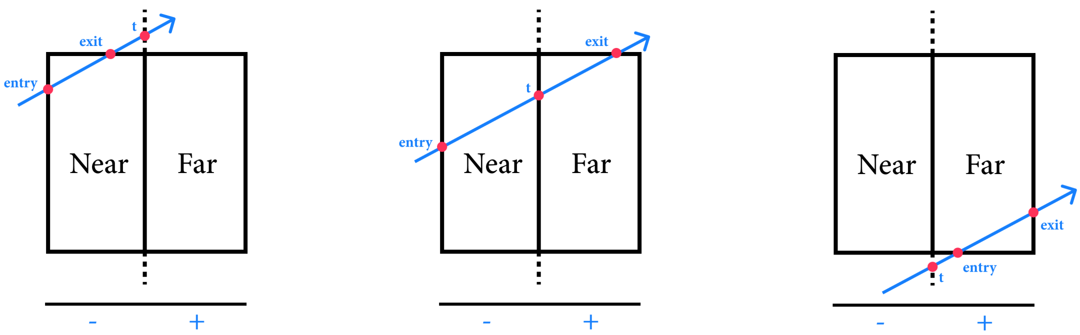
Figure 19: Simple scene with a few triangles and a corresponding tree. Leaves are boxes and inner nodes are circles.

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)
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 we reach the terminal nodes, 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 far child only.

Figure 20: Ray traversal cases.

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 composed of three steps. On the very top, we have our entire scene enclosed 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[3]. The trees can efficiently retrieve the closest intersections of their respective nodes and have bounding boxes that allow 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. Even when nesting constructively generated geometries, one can hold definitions of each sub-tree for each sub-object. Hence, allowing for efficient evaluation of complex and nested geometries. Such definition intrinsically means that each solid is responsible for its evaluation and can constantly feed the correct and classified intersection information to its parent nodes. Therefore, skipping the step of culling all evaluations and solely processing them on the root 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 $BinCSG$. Finally, 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 $OptimCSG$. All these variants are tested using the minimal hit CSG algorithm. We consider each algorithm for all operations and a low, medium, and high viewport fill rate. 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 demonstrates 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. We also run tests to count the number of intersection tests performed by each variant at each pixel. The last test is based on the effect of a different number of nested geometries on the various algorithms while maintaining a relevantly similar viewport fill rate. These tests are all run on the following configuration with CPU parallel processing:

- **Model:** 2020 Macbook Pro
- **Graphics Card:** Integrated Intel Iris Plus Graphics 645 1536 MB
- **RAM:** 16 GB 2133 MHz LPDDR3
- **CPU:** 1,7 GHz Quad-Core Intel Core i7

[3] In case the constructive solid only contains a single primitive element, then the result is a simple tree of depth 1.

5.1 Geometry Complexity Tests

As previously mentioned, we conduct each of these tests on varying viewport fill rates. Figure 21 shows the three main viewport ranges. Different operations give different rates (e.g., difference and intersection will generally produce a less voluminous geometry); therefore, we account for this by determining the range in which the viewport fill wavers. We will refer to low, mid, and high viewport tests respectively as LVP, MVP, and HVP.

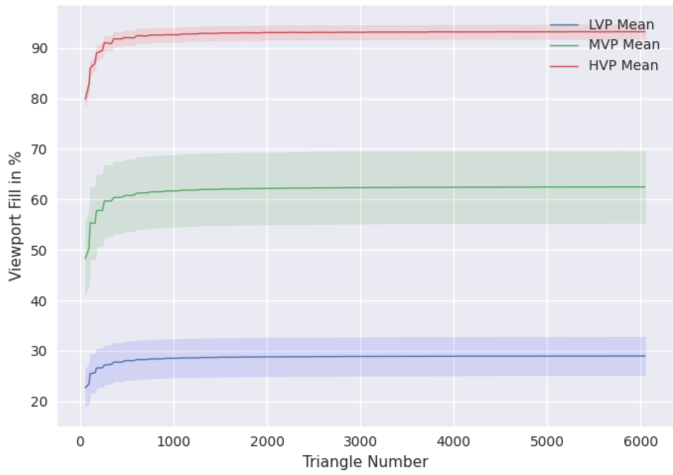


Figure 21: Range of view port rates on which the tests are conducted as the complexity of the geomtery increases. The area around the curve signifies the error by which the rate fluctuates.

First, we will start by examining the different operations for the $NaiCSG$ implementation. Figures 22 to 24 show the performance of each operation in relation to the other. We discern a difference between these operations because ray-solid checks are repeated more often in the intersection and difference operations than the union operation. The difference between these operations remains constant throughout all the other variants as well. $NaiCSG$ variant is also sensitive to changing the viewport fill rate (Figure 25). This is mainly due to the intersection and different operations since more rays have to enter deeper states of the evaluation before exiting. This is highly dependent on the spatial distribution and is the reason why the union time remains the same in all of the viewport ranges.

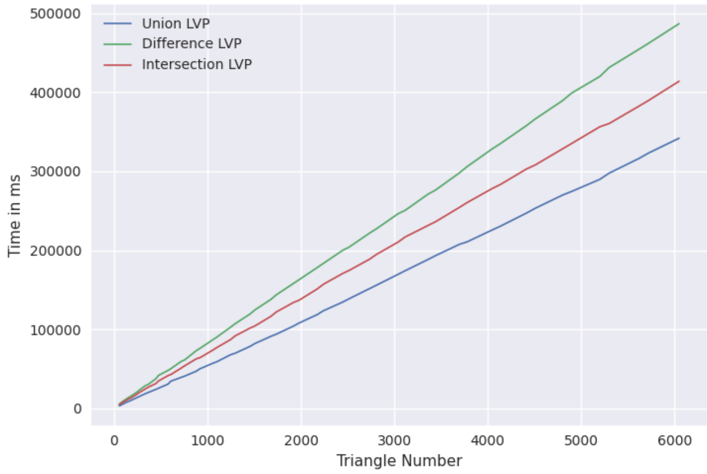


Figure 22: $NaiCSG$ rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a small rate of the view port.

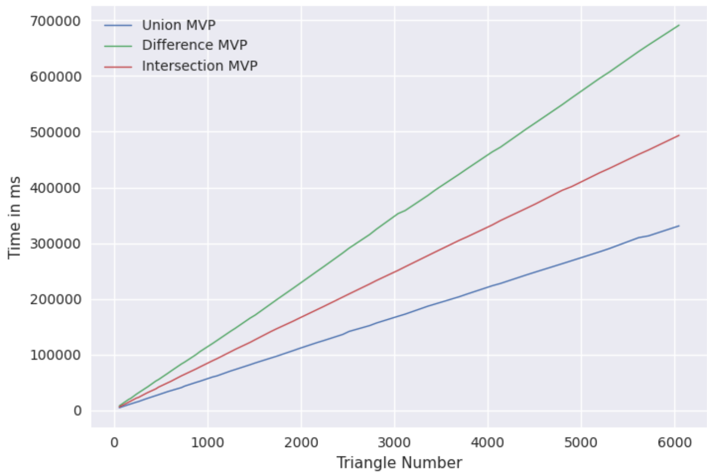


Figure 23: $NaiCSG$ rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a medium rate of the view port.

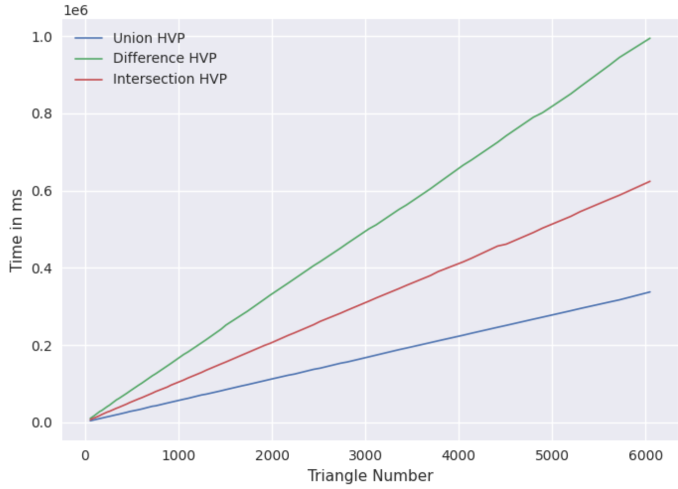


Figure 24: $NaiCSG$ rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a high rate of the view port.

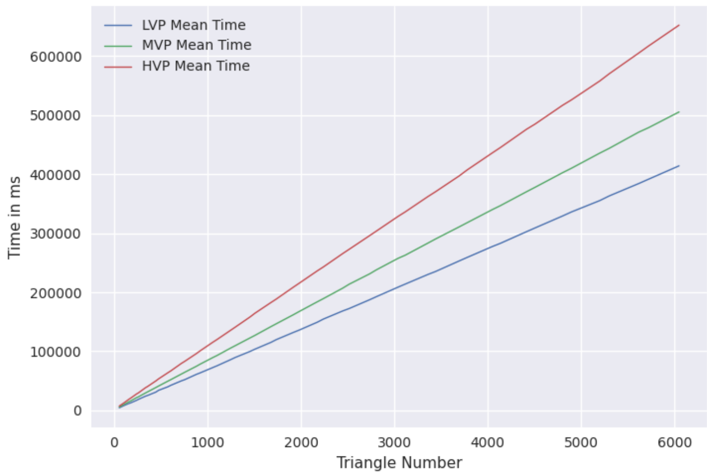


Figure 25: The $NaiCSG$ mean rendering time of the operations for the different view port fill rates.

Second, we compare the rendering time of the $BinCSG$ variant. Figure 26 to 28 show the performance of each operation in relation to the other. In Figure 26, we notice the same discrepancy between the union operation and the two others. A simple check proves that the factor by which the time increases per operation confirms that it is indeed a constant. Figure 33 demonstrates how the computational time of $BinCSG$ worsens depending on the spatial distribution in the scene as the time gained from the ray-box tests performed in $BinCSG$ become less useful.

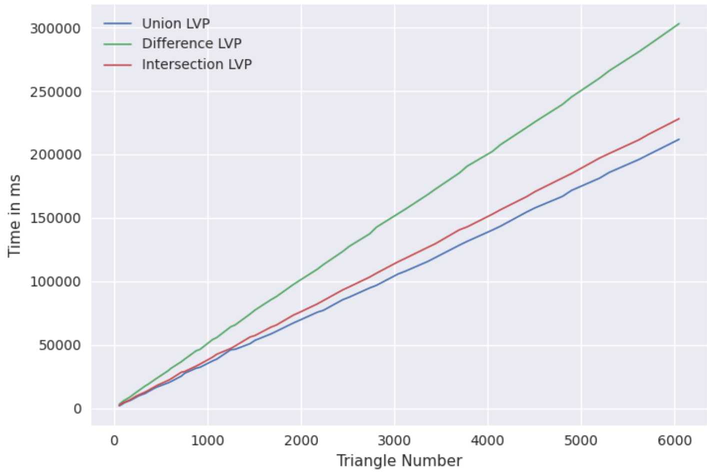


Figure 26: $BinCSG$ rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a small rate of the view port.

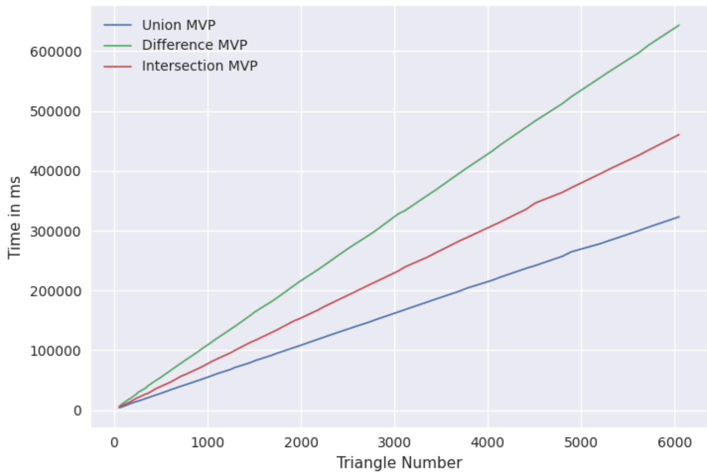


Figure 27: $BinCSG$ rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a medium rate of the view port.

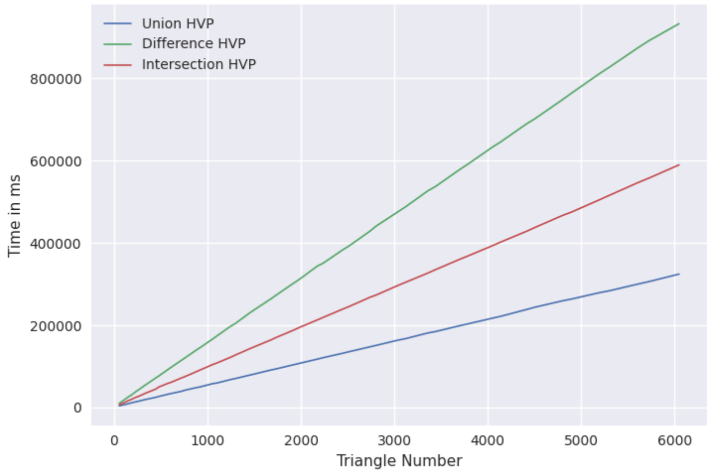


Figure 28: $BinCSG$ rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a high rate of the view port.

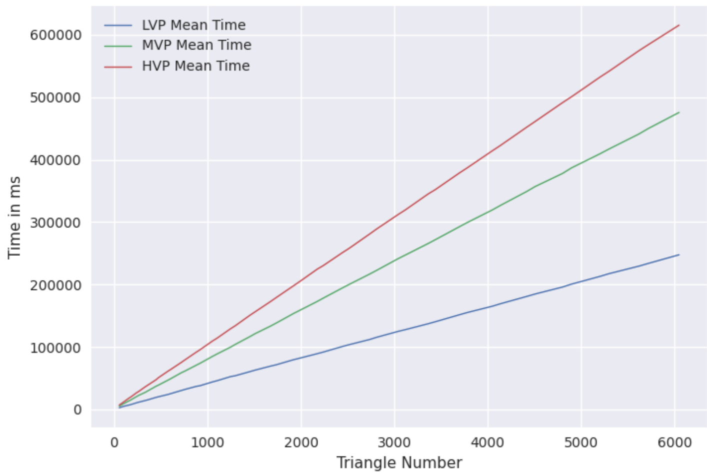


Figure 29: The $BinCSG$ mean rendering time of the operations for the different view port fill rates.

We must also analyze the performance of $OptimCSG$ with the different operations. As we can see in Figures 30 to 32, $OptimCSG$ shows different performance in terms of the operations. The general curve is also not linear but follows a rather logarithmic trend. This can be closely tied to the cost of traversing a binary tree. The procedure also increases in time after an increase in the fill rate.

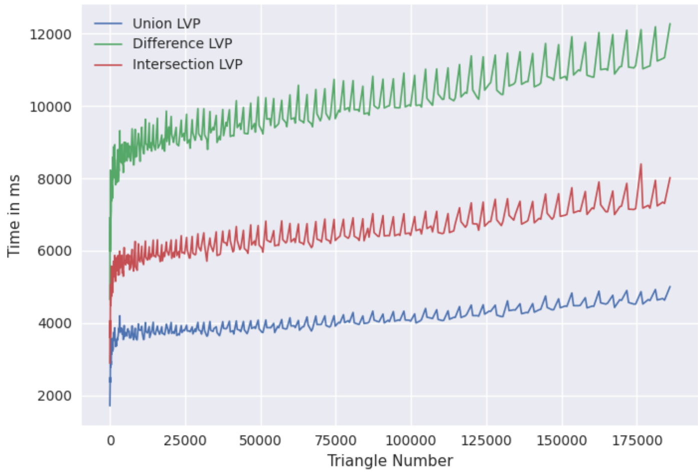


Figure 30: $OptimCSG$ rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a small rate of the view port.

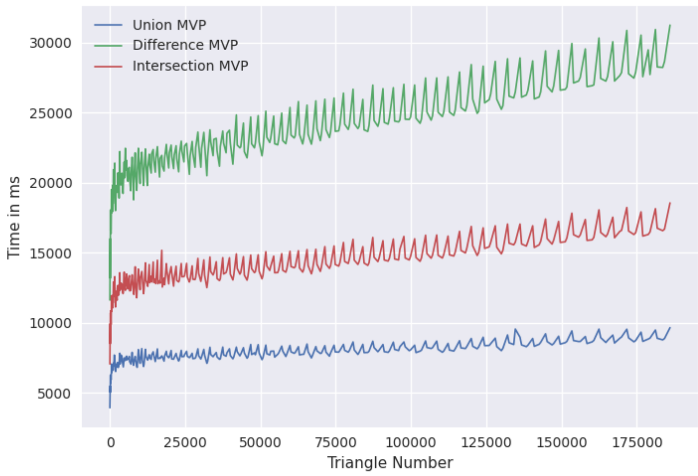


Figure 31: $OptimCSG$ rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a medium rate of the view port.

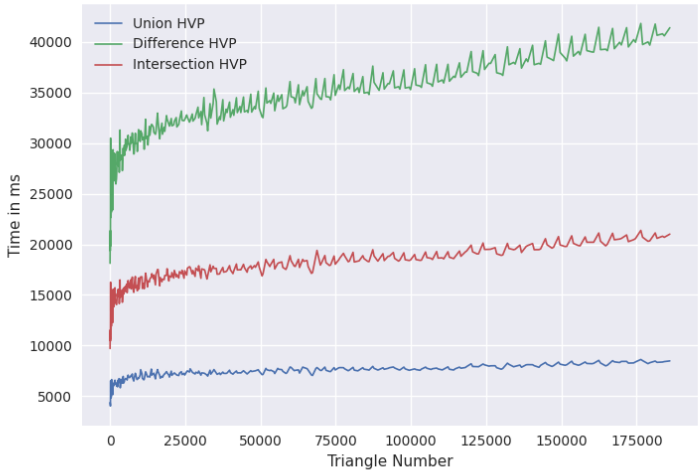


Figure 32: $OptimCSG$ rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a high rate of the view port.

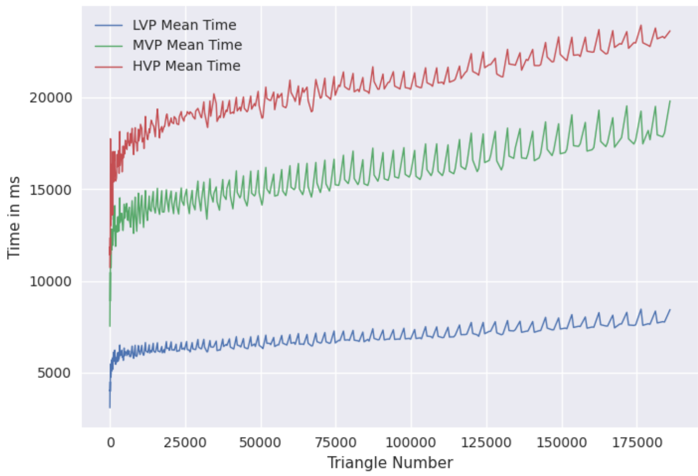


Figure 33: The $OptimCSG$ mean rendering time of the operations for the different view port fill rates.

Lastly, we compare the performance of these variants to each other. Figures 34 to 36 show the comparison of the different implementations with low, mid, and high viewport fills. As we can see, $OptimCSG$ outperforms both variants in all cases. $BinCSG$ does outperform $NaiCSG$ in smaller scenes; however, it scales to the same computational time in more complex ones.

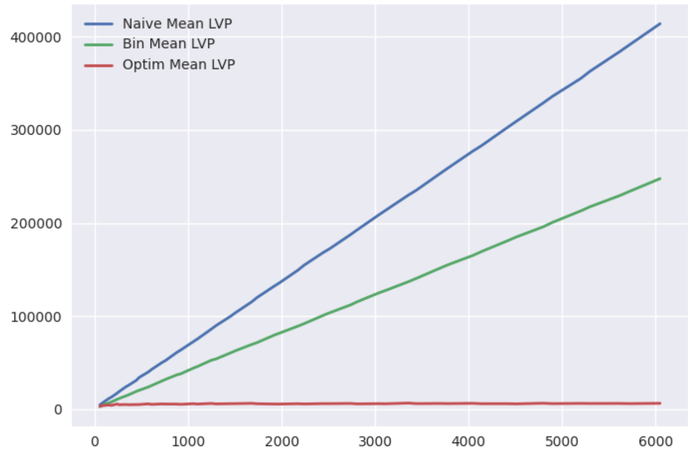


Figure 34: Rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a small rate of the view port.

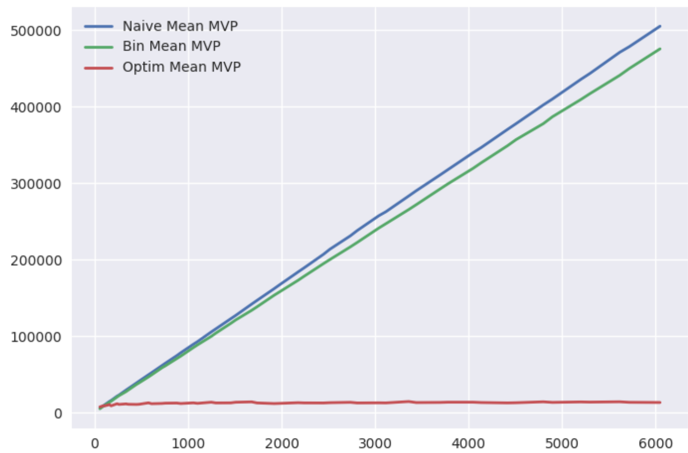


Figure 35: Rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a medium rate of the view port.

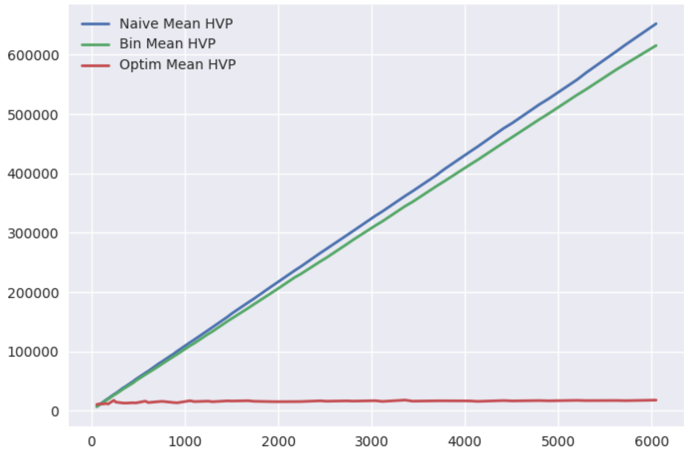


Figure 36: Rendering time of different operations with respect to gradual increases in geometry complexity in a scene filling a high rate of the view port.

We can explain these variations by the number of ray-primitive intersection tests performed by each variant. In the naive implementation, we check for all primitives for all the rays in the scene, which explains why it is not so case-dependent. In $BinCSG$, if the ray-box test is positive, it naively makes the ray-primitive intersections. $OptimCSG$ has a more directed approach as both the left and right geometries in the composite are also split up into smaller boxes and traversed efficiently. Figures 38 to 40 represent the number of intersections performed in a 1000×1000 pixel image with two spheres in the scene (Figure 37). In Figure 38 we can see that we continually make the same number of ray-primitive tests for each pixel. However, Figure 39 reveals how $BinCSG$ is capable of avoiding intersections toward the areas where the bounding box is not defined but still performs all tests in pixels overlapping it. Ultimately, $OptimCSG$ (Figure 40) exhibits a much more efficient approach where the number of ray primitive intersections solely grows in areas dense with primitives or overlapping bounding boxes.

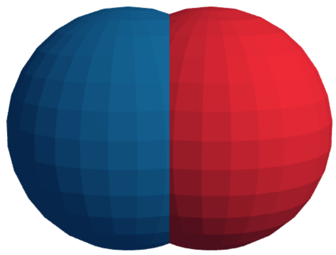


Figure 37: The scene on which the number of intersections is counted.

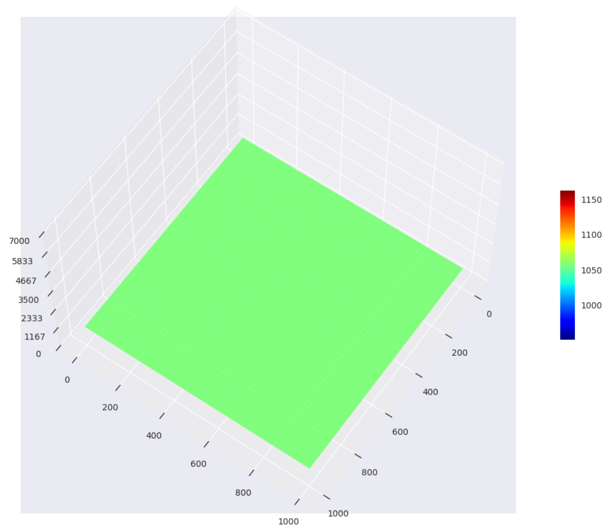


Figure 38: Surface plot showing the number of ray-primitive tests on each pixel with $NaiCSG$.

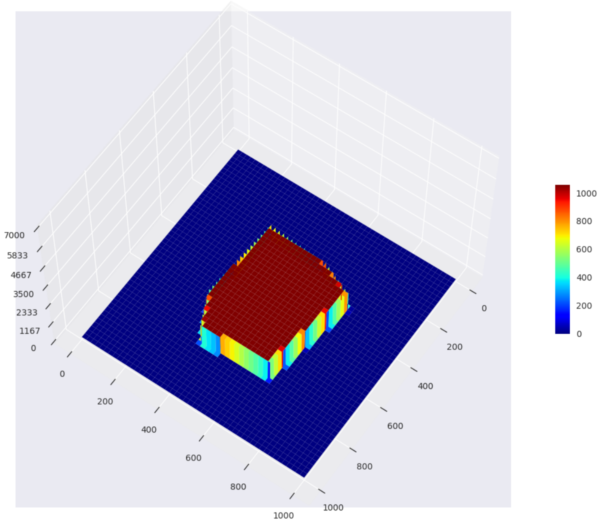


Figure 39: Surface plot showing the number of ray-primitive tests on each pixel with $BinCSG$.

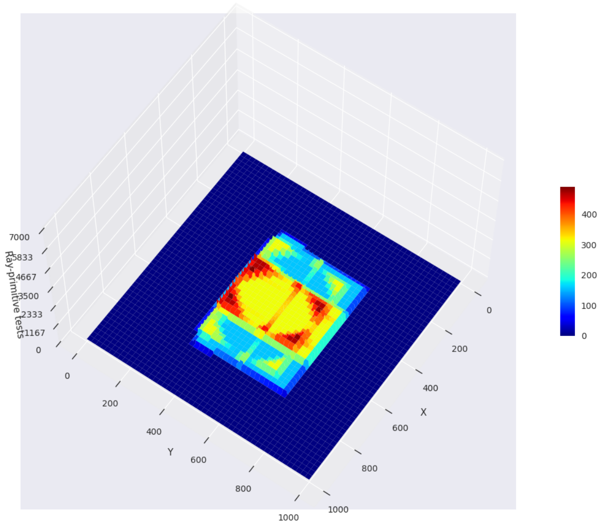
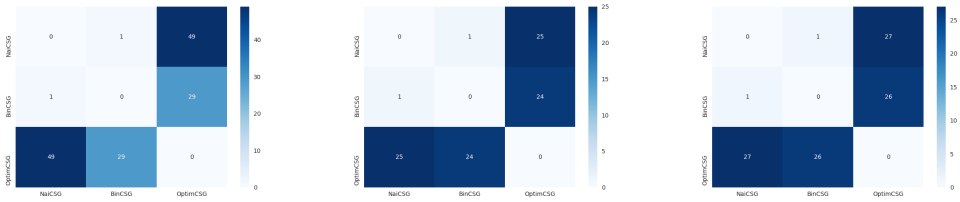


Figure 40: Surface plot showing the number of ray-primitive tests on each pixel with $OptimCSG$.

Figure 41 demonstrates the achieved speedup between all three variants in the various viewport fill rates with 6048 triangles. We can see that $OptimCSG$ can achieve up to 49x faster times in a small scene and up to 26x faster on a dense scene. Additionally, since both $NaiCSG$ and $BinCSG$ grow linearly and $OptimCSG$ grows logarithmically, then the speedup should increase as the spatial distributions gets more complex.



(a) LVP speedup. (b) MVP speedup. (c) HVP speedup.

Figure 41: Achieved speedup across variants.

5.2 Nesting Tests

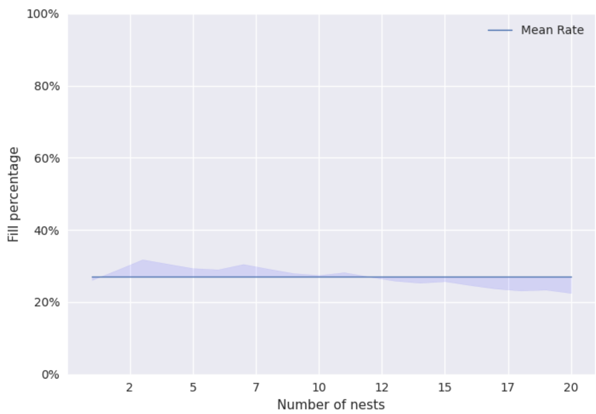


Figure 42: Range of view port rates on which the tests are conducted as the number of nests increases.

Figure 43 portrays the different performances of each variant. As expected, the naive time grows linearly as we use more geometries. $BinCSG$ is also somewhat linear but is expected to scale to the same time as the naive if we increase the viewport fill rate. On the other hand, $OptimCSG$ still follows a logarithmic trend when nesting.

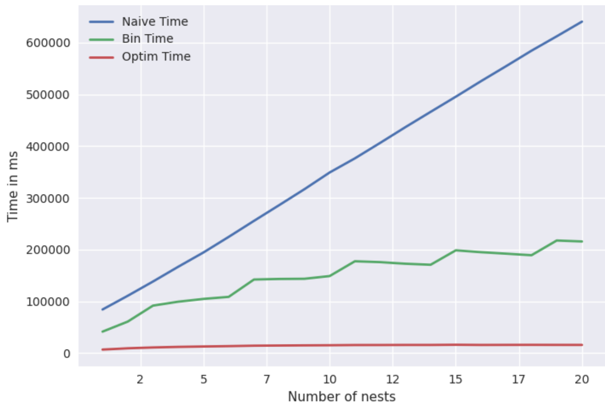


Figure 43: Performance of the different variants in comparison to each other when nesting more geometries together.

Similar to the geomtric complexity tests, $OptimCSG$ provides a much higher speedup. (Figure 44)

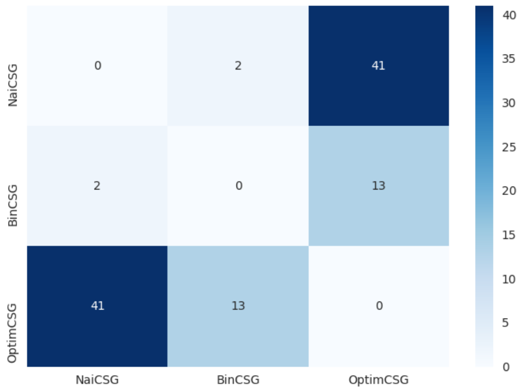


Figure 44: Achieved speedup for nested geometries.

6 Conclusion & Future Work

The observed results in the previous section lay a foundation for accelerating the performance of complex geometrical modeling using CSG in real applications. We have briefly covered the general mathematical foundation of constructively generated geometry. We then introduced the generic idea behind constructive solid geometry and the algebraic rules applied to it. Later, we introduced different optimizations used to push the performance of each operation. Last, we compared each variant and understood its time complexity. There are still various means of possibly improving the performance of the proposed solution while still maintaining a balance of generality and robustness. I outline a few issues with the proposed algorithm and possible future research directions in accelerated ray tracing of constructive solid geometry in this section.

The first encountered issue is artifacts created by the numeric stability issues in classification. When classifying using the surface normals, we will run into the issue where the dot product of the ray direction and the normal is near zero (vectors are orthogonal). Hence, leading to certain artifacts emerging near the boundaries of the geometries of edges. The described issue can be solved using a sampler which would increase the number of rays shot in the neighborhood of a pixel and estimate a better shading result. This solution is already plausible in OpenRT.

The second limitation is that the geometries must have consistently oriented normals to understand intersections. While all primitives and solids constructed inside OpenRT guarantee this property, meshes imported from the outside could potentially lead to issues. However, one can solve this by implementing an extra scan when constructing solids or passing them to a composite to verify and modify the surface normals when needed. Algorithms that allow for fast checks of consistent normal orientation are readily available [4].

Many improvements are also possible in alignment with the work established in this paper. The first is extending the binary space partitioning tree algorithm to be more efficient in building and traversing. Such a change can bring drastic improvements to the performance and handle much more complex scenes. Automated stopping criteria are a way to let each solid deterministically choose which stopping criteria work best, particularly in BREPs.

Second, this research heavily focused on acceleration with CPU-based ray tracing. Many solutions to extend the system to a GPU exist, and such benefits could make the algorithm gain from the ever-increasing performance of the graphics hardware.

Conversion algorithms from constructively defined solids to BREPs are detrimental as well. Conversion allows for faster computations and ease of use. If we can translate a constructively generated geometry to a BREP, we can increase performance, and final geometries would no longer rely on a recursive evaluation. One can then also use the optimized triangle operations for faster computations and rendering. The solution is especially appealing since we can divide complex geometries into smaller models to unify later.

Applying textures to constructive solid geometry is also an area of interest. We could achieve the latter with a few different flavors. Automatic texture mapping is one of them. The goal is of automatic texture mapping is to produce texture coordinates for geometries that don't possess any. One could use a sphere, cube, or any other map to generate these texture coordinates. Therefore, enabling the use of different textures on each of the constructively generated geometries. The texture coordinate generation function can also be specified separately for each geometry from the user.

Topics such as the reconstruction of constructive solid geometry trees using unsupervised neural networks are also interesting. [8] produced models that allow for fast and accurate reconstruction of CSG trees from images. Such an advancement would make the possibility of combining a multitude of geometrical resources simpler. Additionally, combining this with the conversion to a boundary representation makes such a possibility more appealing.

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] P. Borodin, Gabriel Zachmann, and Reinhard Klein. "Consistent normal orientation for polygonal meshes". In: July 2004, pp. 18 –25. ISBN: 0-7695-2171-1. DOI: 10.1109/CGI.2004.1309188.

[5] S. Cameron. "Efficient Bounds in Constructive Solid Geometry". In: *IEEE Computer Graphics and Applications* 11.03 (May 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] Kacper Kania, Maciej Zięba, and Tomasz Kajdanowicz. *UCSG-Net – Unsupervised Discovering of Constructive Solid Geometry Tree*. 2020. arXiv: 2006.09102 [cs.CV].

[9] 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.

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

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

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

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

[23] Sergey Kosov. *OpenRT*. May 8, 2021. URL: https://github.com/Project-10/OpenRT.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] R.B. Tilove. “A study of GEOMETRIC SET-MEMBERSHIP CLASSIFICATION”. en. (Master’s thesis, University of Rochester, 1977.

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

[31] 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.

[32] 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.

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