Skoltech

MASTER'S THESIS

Adaptive Constructive Solid Geometry with Constant Evaluation Complexity for Modeling Implicitly Defined Complex Objects

Master's Educational Program: Applied Computational Mechanics

Student: ____________ Egor Chulkov
signature

Research Advisor: ____________ Oleg V. Vasilyev
signature
D.Sc., Ph.D., Professor

Moscow 2025

Skoltech

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Адаптивная блочная геометрия с константной вычислительной сложностью для моделирования неявно заданных сложных объектов

Магистерская образовательная программа: Прикладная вычислительная механика

Студент:	____________ *подпись*	Егор Чулков
Научный руководитель:	____________ *подпись*	О.В. Васильев д.ф.-м.н., Ph.D., профессор

Москва 2025

Adaptive Constructive Solid Geometry with Constant Evaluation Complexity for Modeling Implicitly Defined Complex Objects

Egor Chulkov

Submitted to the Skolkovo Institute of Science and Technology on June 2, 2025

ABSTRACT

This thesis presents a novel mathematical framework for efficient representation and manipulation of complex implicitly defined objects using Function Representation (F-rep) and Constructive Solid Geometry (CSG). Traditional F-rep approaches suffer from high computational costs during evaluation and rendering, especially for detailed models. To address these challenges, the research introduces a comprehensive framework based on adaptive Constructive Solid Geometry integrating interval analysis techniques—interval arithmetic (IA), affine arithmetic (AA), and revised affine arithmetic (RevAA)—with Reverse Polish Notation (RPN) pruning and hierarchical spatial decomposition. The developed framework demonstrates that RevAA significantly outperforms IA and AA in pruning efficiency, achieving up to 180× acceleration in evaluation times. Adaptive spatial decomposition using octrees ensures constant evaluation complexity, while a hybrid ray-casting method with interval bisection enables robust visualization. Experimental results on 2D and 3D benchmark problems validate the effectiveness of framework, showing substantial performance improvements.

Keywords: Function Representation, Interval Arithmetic, Affine Arithmetic, Constructive Solid Geometry, Ray-Casting

Research advisor:
Name: Oleg V. Vasilyev
Degree, title: D.Sc., Ph.D., Professor

Contents

Chapter 1
Introduction

1.1 Background and problem statement

Representation of three-dimensional objects has been a fundamental challenge in computer graphics and design for decades, particularly as modern technologies advance to render and manufacture increasingly complex models. The predominant approach in industry, **boundary representation (B-rep)**, represents three-dimensional objects as collections of interconnected surface primitives. This methodology, established decades ago, has become the de facto standard in **computer-aided design (CAD)**, **engineering (CAE)** and **manufacturing (CAM)** applications [33], reinforced by the development of specialized **graphics processing units (GPU)** that have revolutionized rendering capabilities.

However, B-rep faces significant limitations in scaling the model complexity. Modern manufacturing technologies, particularly 3D printing, can now achieve unprecedented precision at microscale resolutions, demanding highly detailed geometric representations [38, 56]. While B-rep can accommodate such detail by increasing polygon density, this approach leads to substantial memory requirements and computational overhead. Furthermore, modeling with set-theoretic operations, like union, intersection and difference, is not always precise in B-rep, introducing approximation errors and undefined behaviour. B-rep also faces fundamental limitations in representing complex algebraic surfaces, e.g., Kiss and Bohemian Star surfaces [11].

In recent decades, implicit modeling – defining shapes as the solutions to equations or level-sets of functions – has emerged as a powerful alternative. Implicit representations include **function representation (F-rep)** and **signed distance functions (SDF)**, which define solids by a real-valued function $F(\mathbf{x})$ of point coordinates instead of explicit vertices and faces. This method provides precise object representation, limited only by floating-point precision, making it particularly suitable for high-resolution modeling in animation [28] and advanced manufacturing applications [39]. Additionally, F-rep supports advanced geometric operations such as blending, twisting, and offsetting. Pasko et al. [48] developed the formal F-rep approach into a modeling framework. Later, the HyperFun project [8] delivered a collaborative, open-source toolkit for F-rep modeling, complete with a high-level language (HyperFun) and tools for polygonization, ray tracing, and even an experimental Web-based modeler [17].

In computational mechanics, F-rep offers significant advantages for constructing complex computational domains that are challenging to define using traditional B-rep approaches [48]. However, since most numerical solvers rely on discretized geometric representations (typically finite element or finite volume meshes derived from B-rep), a transformation from F-rep to a polygonized mesh representation becomes necessary. This conversion step introduces an additional computational requirement but maintains the benefits of F-rep's precise geometric definition during the domain construction phase.

In solid modeling, F-rep inherently supports **Constructive Solid Geometry (CSG)**, a standard assembly technique for modeling complex solids or surfaces by using Boolean operators to combine simpler objects [52]. Combination of CSG and F-rep is straightforward [53]: primitives are implicit functions, and Boolean operators are R-functions [54], which extend Boolean algebra to the continuous domain.

Despite these advantages, fundamental challenge of F-rep stems from its monolithic approach, where the entire geometric domain is represented through a single expression. Local surface modifications necessitate changes to the entire function, often resulting in increased computational costs. For domains comprising thousands of objects, even seemingly simple operations can become computationally intensive [30]. This limitation has historically restricted practical adoption of F-rep in industry. Moreover, visualizing F-rep objects on the screen is an open problem, which significantly limits real-time interaction with 3D models. Lack of hardware acceleration for rendering F-rep objects is a significant drawback in contrast to B-rep, which makes difficult a direct comparison of both approaches.

Over the past decade, researchers have attempted to address these optimization challenges [36, 39, 51, 63] through various techniques, including interval [41] and affine arithmetic [59], as well as spatial acceleration structures [30, 61]. Despite these efforts, the efficient evaluation of function representation remains an open problem and constitutes the primary focus of this thesis.

1.2 Objectives and scope

The primary goal of the thesis is to develop a comprehensive mathematical framework enabling computationally efficient representation and manipulation of complex implicitly defined objects. This research aims to address the computational challenges inherent in F-rep evaluation through the development of innovative algorithms for function compression and spatial adaptation.

To achieve this goal, the following key objectives have been identified:

1. Design and implement an efficient algorithm for pruning CSG tree of F-rep primitives within localized spatial regions, enabling selective evaluation of geometric operations.
2. Develop a hierarchical data structure and associated algorithms for storage and evaluation of CSG tree across diverse spatial regions, emphasizing memory efficiency and computational performance.
3. Formulate and implement an adaptive spatial sampling algorithm utilizing sparse volume data structures.
4. Develop and implement a robust ray-tracing algorithm for visualizing F-rep objects.
5. Conduct comprehensive performance evaluation of the proposed methodology against existing F-rep implementations, establishing quantitative metrics for comparison.
6. Implement all proposed data structures and algorithms in prototype geometry and rendering kernels.

Due to time limitations, the scope of this research is deliberately constrained to the development and validation of core algorithmic approaches for efficient F-rep evaluation and storage. The following aspects are explicitly excluded from the research scope:

- Hardware acceleration, using GPU and CPU parallelism,
- Development of complete CAD software solutions,
- Development of plugins for existing commercial CAD platforms,
- Interactive modeling and rendering.

It is important to note that while the development of a production-ready geometric kernel would require substantial resources and a team of software engineers, this research concentrates on establishing the theoretical foundation and algorithmic frameworks necessary for future implementations. Real-time rendering of F-rep objects requires calculations to be performed on GPU and represents a distinct research direction that warrants separate investigation and is reserved for future work.

Chapter 2
Author contribution

The author made substantial contributions to the research presented in this master's thesis. Specifically, he developed and implemented an innovative mathematical framework integrating Constructive Solid Geometry (CSG) with Function Representation (F-rep), addressing significant computational challenges previously inherent in these methods.

His key contributions include:

1. Designing and implementing an adaptive algorithm for pruning the CSG tree, significantly optimizing the evaluation of complex implicitly defined geometric objects through selective evaluation and reduced computational overhead.

2. Introducing and validating the use of Reverse Polish Notation (RPN) combined with bitmask-based storage to efficiently manage and evaluate pruned CSG trees. This novel approach improved data accessibility and cache performance compared to traditional methods.

3. Developing a hierarchical spatial sampling algorithm based on Sparse Voxel Octrees (SVO), enabling adaptive, memory-efficient storage, and evaluation of complex geometric models. This advancement allowed constant evaluation complexity regardless of model detail.

4. Implementing and evaluating interval analysis techniques including Interval Arithmetic (IA), Affine Arithmetic (AA), and Revised Affine Arithmetic (RevAA). Through rigorous benchmarking, the author demonstrated the superiority of RevAA in terms of pruning efficiency and computational speed.

5. Creating a robust ray-casting visualization algorithm that leverages interval-based root-finding to provide accurate and efficient rendering of complex F-rep objects. The effectiveness of this method was validated through comprehensive experimental results across diverse geometric benchmarks.

6. Conducting detailed performance evaluations comparing the developed adaptive CSG framework with existing state-of-the-art methods, clearly illustrating substantial improvements in rendering speed, evaluation time, and memory efficiency.

Chapter 3
Literature review

Implicit surfaces entered computer graphics in the 1980s as a way to create smooth organic shapes. Notably, Blinn's 1982 work for the Cosmos TV series introduced "blobby molecules," modeling atoms by Gaussian density fields that merge when close [14]. These were later termed metaballs by Nishimura et al. [44], highlighting how multiple implicit fields meld into a single smooth object. Such skeletal implicit models were popular for organic modeling and special effects, but they were not part of mainstream CAD for engineering – early CAD in the 1970s and 80s was dominated by CSG and B-rep. The concept of constructive implicit geometry was explored by researchers like Ricci [53], who formulated union and intersection in terms of implicit equations, foreshadowing F-rep. Through the late 80s and 90s, implicit surfaces gained a firm theoretical footing [15] and started appearing in modeling systems.

In the 1990s, researchers sought to bridge the worlds of implicit surfaces and solid modeling. A milestone was the work of Pasko et al. [48], who developed the formal F-rep framework. Their paper presented the theoretical foundation and a prototype language for function-based shape modeling. Later, the HyperFun project [8] delivered a collaborative, open-source toolkit for F-rep modeling, complete with a high-level language (HyperFun) and tools for polygonization, ray tracing, and even an experimental Web-based modeler [17]. F-rep was demonstrated on various applications: product design with complex blends, collision detection via distance fields, processing of 3D scan data, and even n-dimensional modeling for animation [48]. In fact, one strength of F-rep is its natural support for multidimensional geometry: the defining function can take additional coordinates beyond x, y, z. For example, time t can be treated as a fourth coordinate, so a moving/deforming 3D shape is represented as a static 4D implicit object $F(x, y, z, t) \geq 0$ [58]. By slicing at different t, one obtains the shape at a given time. This enabled smooth metamorphosis (morphing) between shapes and other spatio-temporal operations. Pasko's team explicitly demonstrated shapes in up to 4D or higher (so-called hypervolumes), with the same functional approach applying in any dimension. Such generality was a leap beyond conventional CAD, which mostly dealt with static 3D models.

In the 2000s, implicit representations gained traction in simulation and volume modeling. The level set method [47] became a popular technique to deform implicit shapes by evolving an SDF over time, used for simulating fluids or optimizing shapes. Though originating in computer graphics and computational physics, these methods influenced CAD in areas like shape optimization and metamorphic design. Researchers also developed adaptive spatial data structures for implicit objects, such as **adaptively sampled distance fields (ADF)** [22], which store SDF values on an octree grid, refining where more detail is needed. This made it feasible to represent detailed implicit geometry with less memory, an idea later revisited by neural implicit models [31]. Commercial modeling and animation tools incorporated implicit surface features too (e.g., Blender's metaballs, volumetric effects in game engines), but adoption in mechanical CAD remained limited.

In the 2010s, two trends sparked renewed interest in implicit modeling for CAD: additive manufacturing and generative design. 3D printing enabled fabrication of complex lattice structures and organic forms that traditional CAD struggled to model explicitly. Implicit modeling – especially field-based representations – can handle enormous complexity (like thousands of lattice struts or graded microstructures) without managing individual faces for each tiny feature [49]. This led to startups and research projects focusing on implicit CAD modeling. For example, nTopology introduced a commercial CAD tool built on implicit modeling around 2018 [45]. By 2019, their software demonstrated the ability to generate highly complex lattices and perform boolean and offset operations on them robustly. Similarly, software like Autodesk Within and other lattice generation tools [1] have internally used distance-field representations to make lightweight structures. Nowadays, it is clear that implicit modeling is transitioning from a research topic into practical CAD and CAE workflows.

Traditionally, F-rep have employed a monolithic approach to model representation, where all geometric modifications are contained within a single expression. This approach leads to increased evaluation costs as model complexity grows, even for localized changes. Consequently, a significant focus of F-rep research has been on reducing computational costs for evaluation and visualization [40, 36, 63, 65]. Interval analysis has emerged as a key technique for minimizing computation costs in geometric function evaluation [30, 61]. Duff [20] demonstrated application of **interval arithmetic (IA)** in creating hierarchical spatial partitioning and local function simplification for CSG tree of implicit primitives. His approach allows for the derivation of a *pruned* evaluation tree that preserves local geometry for any spatial subdivision. By omitting evaluations of non-overlapping geometric primitives, the overall function evaluation count is reduced. Similar approach in B-rep was proposed by Tilove [60] with similar pruning algorithm.

Regarding visualization, Keeter [30] implemented an F-rep geometric kernel and a GPU interpreter, which allows for real-time modeling and rendering. In contrast to Duff's approach [20], Keeter prunes specific arithmetic operations, such as $\min / \max$ operators, in implicit functions. As a result, Keeter's pruning algorithm guarantees, that value of F-rep defining function is preserved inside local region, where pruning is performed. In Duff's approach, pruning preserves only the sign of F-rep defining function, enabling better pruning capabilities, and still preserving the geometry. Nevertheless, Keeter's advancement showed that real-time rendering of F-rep on GPU is feasible. This advancement further optimizes the evaluation process by utilizing parallel computing capabilities, potentially offering significant performance improvements in F-Rep model processing and visualization. However, Keeter's approach lacks spatial adaptivity and efficient storage of pruned F-rep, which results in higher computational cost of pruning algorithm and increased memory consumption.

Continuing Keeter's work, Uchytil and Storti [61] developed a geometry kernel using a sparse volume data structure for F-rep data management and implemented it for CAD applications and visualization. As volume data structure they used N^3-tree, which is a generalization of binary tree from 2 to N^3 children . Their approach combines IA, spatial adaptivity, **just-in-time (JIT)** compilation of user-defined functions, and GPU-based parallel evaluation for efficient F-rep processing. Also, they implemented more efficient method of storage pruned versions of expression trees using bit masks. As a result, in most of the cases they have much better performance than Keeter, which demonstrates feasibility of the real-time rendering.

Currently, most of research is done on SDFs, which are a subclass of functions of F-rep. SDFs are implicitly defined functions, whose magnitude is equal to distance to zero-level surface and sign divides interior and exterior of object. SDF find applications in various domains: CAD packages [3, 6], demoscene [16, 7], modeling applications [21] and games [4]. The distance property significantly facilitates rendering procedure. Based on distance property, Hart [26] proposed a rendering technique called Sphere Tracing, which guaranteed not to penetrate the implicit surface. In recent years, several research papers on pruning of SDF appeared [12, 65], but they cover pruning only for rendering purposes, without changing the representation of SDF itself.

Despite the fact that standard criteria for space partitioning remains interval bounds evaluated by IA, there is still place for improvements. IA has been known to predict wider bounds of values of function, than it is, which sometimes results in effect called *error explosion*. It happens when evaluated bounds are extremely big and bounds evaluation becomes unnecessary. Alternative approaches, that are not explored in previous research, are **affine arithmetic (AA)** [59], **reduced**

affine arithmetic (RAA) [32] and **revised affine arithmetic (RevAA)** [23]. These arithmetics are particularly useful, when objects are expressed through algebraic functions, improving accuracy in bounds evaluation. Thus, implementing pruning algorithm with newest interval analysis techniques can result in higher pruning performance and better spatial adaptivity.

The remainder of the thesis is organized as follows. Section 4.1 reviews the F-rep geometric approach, examining core modeling operations and techniques. A detailed analysis of CSG follows in Section 4.2 with emphasis on its algorithmic foundations and practical applications. The discussion subsequently transitions to interval-based range evaluation techniques in Section 4.3, offering a concise comparison of IA, AA, and RevAA. Section 4.4 evaluates state-of-the-art sparse volume data structures, highlighting their role in adaptive geometric modeling. Methodological and algorithmic contributions are outlined in Section 5, culminating in Section 6, which presents comprehensive benchmarking results for the prototype geometry kernel integrating all proposed algorithms. Future work and concluding remarks appear in Sections 7 and 8, respectively.

Chapter 4
Key supporting technologies

4.1 Implicit modeling (Function representation)

Function representation constitutes a fundamental approach to geometric modeling through the utilization of implicit function $F(x, y, z)$, referred to as the *defining* function [54]. The classification of spatial points (x, y, z) is determined through evaluation of the function's sign: $F(x, y, z) > 0$ indicates interior points, $F(x, y, z) = 0$ corresponds to surface points, and $F(x, y, z) < 0$ denotes exterior points (see Figure 4.1). While the selection of sign convention remains arbitrary, this investigation adheres to the aforementioned convention. A canonical example of F-rep implementation is the mathematical representation of a sphere:

$$F(x, y, z) = R - \sqrt{x^2 + y^2 + z^2}. \tag{4.1}$$

This formulation demonstrates the inherent elegance of **point membership classification (PMC)** - the problem for spatial point classification relative to object boundaries. The problem can be conceptualized as a transformation from continuous space to ternary logic space, encompassing three distinct states: interior, exterior, and boundary. Significantly, when considering a singular function, only the sign evaluation is requisite for point classification, as magnitude becomes irrelevant. This property facilitates efficient pruning of F-rep, which will be elaborated upon subsequently.

Set-theoretic operations within F-rep approach were extensively developed by Rvachev [54], who established a rigorous mathematical foundation for implicit function manipulation. His research addressed the inverse geometric problem, where B-rep serves as output and F-rep as input. Rvachev introduced *R-functions*, characterized by their distinctive property: their signs depend exclusively on the signs of their arguments. This framework introduces a logical function (operating in ternary logic) designated as the *companion* function of the R-function. Before the definition of

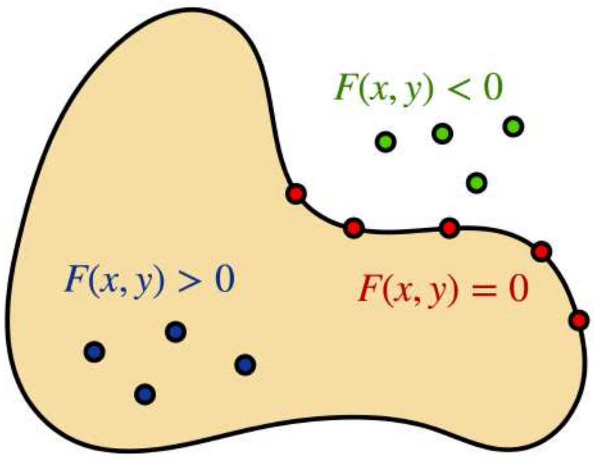


Figure 4.1: Function representation of arbitrary object.

companion function, the characteristic function should be defined as:

$$S(x) = \begin{cases} 2 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \\ 0 & \text{if } x < 0 \end{cases}. \tag{4.2}$$

The companion function correspondence can be formally expressed as:

$$S[F(x, y, z)] = K[S(x), S(y), S(z)], \tag{4.3}$$

where $F : R^3 \to R$ - defining function, $K : [0, 1, 2]^3 \to [0, 1, 2]$ - companion function of $F(x, y, z)$. This correspondence enables the mapping of set-theoretic operations from ternary logic to the continuum space. The fundamental set-theoretic operations can be expressed through R-functions as follows:

$$\begin{aligned} \text{Union } (\vee)\text{:} \quad & F(f_1, f_2) = f_1 + f_2 + \sqrt{f_1^2 + f_2^2}, \\ \text{Intersection } (\wedge)\text{:} \quad & F(f_1, f_2) = f_1 + f_2 - \sqrt{f_1^2 + f_2^2}, \\ \text{Negation } (\neg)\text{:} \quad & F(f) = -f, \end{aligned} \tag{4.4}$$

where $f_1(x, y, z)$ and $f_2(x, y, z)$ are the defining functions of two solid objects. While these functions are widely adopted due to their differentiability and analytical properties, they represent one particular class of R-functions. Alternative formulations exist, including the computationally efficient but non-differentiable max and min functions for union and intersection operations, respectively.

The application of F-rep extends to complex geometric modeling tasks. For instance, a unit square can be formally defined as:

$$F(x, y) = (x \wedge (1 - x)) \wedge (y \wedge (1 - y)). \tag{4.5}$$

A particularly significant application of F-rep lies in the modeling of periodic microstructures [49]. A three-dimensional periodic lattice can be efficiently represented as:

$$F(x, y, z) = \sin(ax + b) \vee \sin(ay + b) \vee \sin(az + b), \tag{4.6}$$

where a controls the periodicity and b determines the phase shift. This representation has been successfully employed in [49] for the fabrication of periodic microstructures through additive manufacturing, utilizing surface polygonization techniques.

The F-rep approach encompasses additional modeling operations, e.g., blending, twisting, offsetting, and bijective mapping. Of particular interest is the blending operation, which introduces smoothing at the intersection of set-theoretic operations where functions approach zero. The mathematical formulation in F-rep is:

$$F(f_1, f_2) = R(f_1, f_2) + d(f_1, f_2), \tag{4.7}$$

where $R(f_1, f_2)$ represents the set-theoretic operation and $d(f_1, f_2)$ denotes the displacement function, which exhibits asymptotic convergence to zero as f_1 and f_2 approach zero. This decomposition is significant as it enables the implementation of displacement functions with diverse blending characteristics. A frequently implemented displacement function takes the form:

$$d(f_1, f_2) = \frac{a_0}{1 + \frac{f_1^2}{a_1^2} + \frac{f_2^2}{a_2^2}}, \tag{4.8}$$

where a_0, a_1, a_2 represent scaling parameters. The sign of a_0 determines the blending morphology: positive values produce expansion relative to the original object, while negative values result in contraction. This formulation offers computational efficiency; however, it lacks spatial localization. The scaling parameters a_0, a_1, a_2 cannot constrain blending to regions where functions exhibit zero values. Blending extent primarily depends on the morphological characteristics of f_1 and f_2 functions. Furthermore, as blending primarily serves to smooth transitions between objects, localization becomes a requisite property.

To address these limitations, localized displacement functions have been proposed. Pasko [50] introduced bounded blending operations and presented several construction methodologies. It should be started with the definition of a displacement function of single variable $d_b(r)$ satisfying the following conditions:

1. $d_b(r) > 0$ and $d_b(r)$ reaches maximum value at $r = 0$,
2. $d_b(r) = 0$ for $r \geq 1$,
3. $\left.\dfrac{\partial d_b}{\partial r}\right|_{r=1} = 0$.

Here, r represents a generalized distance constructed from defining functions of initial solids. Among the various functions $d_b(r)$ satisfying the necessary conditions, the following piecewise-defined function could be employed:

$$d_b(r) = \begin{cases} (1-r)^2, & \text{if } |r| \leq 1 \\ 0, & \text{if } |r| > 1 \end{cases}. \tag{4.9}$$

This selection is justified by its constant curvature properties, which provide enhanced visual quality in rendering applications. The generalized distance r is defined according to the following formulation [50]:

$$r^2(f_1, f_2) = \frac{f_1^2}{a_1^2} + \frac{f_2^2}{a_2^2}, \tag{4.10}$$

where $f_1(x, y, z)$ and $f_2(x, y, z)$ represent defining functions of solid objects in the blending operation; $a_1 = f_1(P_2)$, with P_2 designated as a control point on surface $f_2 = 0$; $a_2 = f_2(P_1)$, with P_1 designated as a control point on surface $f_1 = 0$. This function exhibits the following properties:

1. $r^2(0, 0) = 0$,
2. at point P_1: $f_1(P_1) = 0$, $f_2(P_1) = a_2$, and $r^2(f_1, f_2) = 1$,
3. at point P_2: $f_1(P_2) = a_1$, $f_2(P_2) = 0$, and $r^2(f_1, f_2) = 1$.

These properties ensure that the displacement function $d_b(r)$ achieves maximum value at surface intersection points where $f_1 = 0$ and $f_2 = 0$, while vanishing at control points P_1 and P_2. The complete formulation for bounded blending operations with control points is thus expressed as:

$$F_b(f_1, f_2) = R(f_1, f_2) + a_0 d_b(r), \tag{4.11}$$

Localized blending operation allows for additional simplification of defining function, which result in a more efficient function evaluation.

4.2 Constructive Solid Geometry

Constructive Solid Geometry is a modeling paradigm in which complex 3D shapes are built by applying set-theoretic Boolean operations (union, intersection, difference) to simpler solid primitives [52]. Instead of explicitly storing surfaces or meshes, a CSG model is defined procedurally as a binary tree: the leaves are primitive solids (e.g. cuboids, cylinders, spheres), and the internal nodes are Boolean operations that combine these volumes (see Figure 4.2). By successively merging and cutting primitives, a modeler can obtain very intricate geometries from a relatively compact specification. Importantly, every intermediate result of CSG operations is itself a valid solid – a closed, bounded volume – so the final model is guaranteed to be a well-defined solid. This robust solid-by-construction property is a key advantage of CSG modeling in professional CAD and CAM applications.

To visualize or utilize a CSG model, the abstract constructive description must be evaluated into a geometric form. This process, known as *boundary evaluation*, computes the actual surfaces or mesh of the solid by carrying out the Boolean operations on the primitive geometries [46]. In principle, evaluating a CSG tree involves computing all pairwise intersections of primitive surfaces through the tree hierarchy, which can be complex for many primitives. Early algorithms for CSG evaluation converted the tree into an explicit boundary representation by finding intersection curves and splitting faces, often using robust geometric predicates to handle the set operations. An alternative is direct rendering: rather than converting to a mesh, the CSG model can be drawn by algorithms that implicitly compute the shape on-the-fly. One common approach is ray-casting, wherein rays are shot into the scene and each ray is recursively classified against the CSG tree (e.g., using IA or **binary space partitioning (BSP)** trees) to find the first intersection. Ray-casting is naturally suited to CSG because one can compute ray–primitive intersections and then combine the distance intervals per the Boolean operations. In practice, many CAD systems use CSG internally for modeling convenience, but convert the final result to a boundary mesh for display, finite-element analysis, or manufacturing.

The evaluation cost of CSG tends to scale with the number of nodes: each additional Boolean operation introduces a new combination step, so naive evaluation time grows roughly

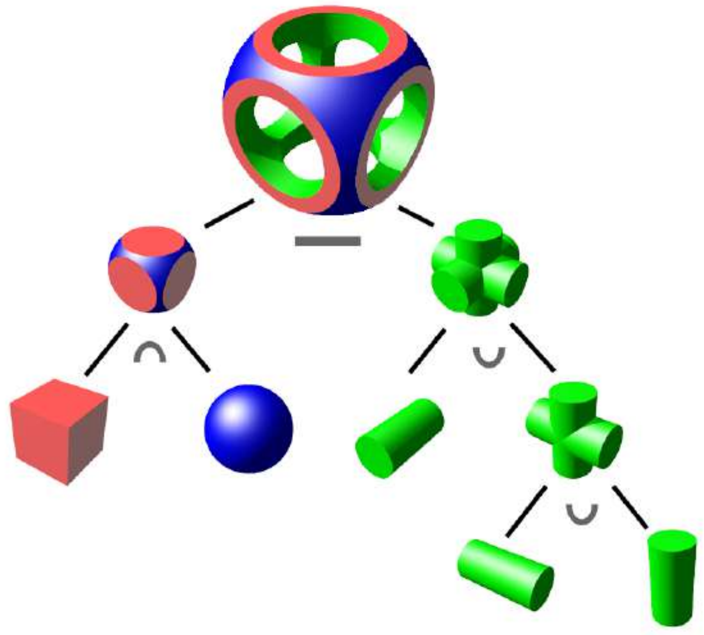


Figure 4.2: Illustration of a CSG modeling tree.

linearly with the number of primitives and operations. This linear complexity can become a performance bottleneck for very large CSG trees, motivating ongoing research into more efficient evaluation algorithms and data structures. Recent work has explored GPU parallelization and deferred evaluation techniques to mitigate the cost, but the dependence on tree size remains primary performance limitation of CSG in practice.

A natural extension of CSG is an F-rep approach to modeling, substituting Boolean combinations with R-functions [54]. They ensure that the zero-level set of the resulting F-rep corresponds exactly to the desired CSG combination. In recent years, the CSG extension to F-rep gained popularity [30, 61, 12], enabling real-time rendering and modeling, yet sacrificing pruning capabilities. The proposed research targets to achieve constant evaluation complexity of CSG tree, leveraging Boolean operations, defined in F-rep, and range evaluation techniques. This approach extends Duff's work [20] to more aggressive pruning, resulting in faster CSG tree evaluation and rendering.

4.3 Interval analysis techniques

Interval arithmetic

Interval arithmetic provides a foundational approach by representing each uncertain quantity as a closed interval $[a, b]$, explicitly capturing the range of possible values. Arithmetic operations are then defined on these intervals such that the resulting interval is guaranteed to contain the true result of the operation. For instance, addition and subtraction are defined as follows:

$$[a_1, b_1] + [a_2, b_2] = [a_1 + a_2, b_1 + b_2],$$
$$[a_1, b_1] - [a_2, b_2] = [a_1 - b_2, b_1 - a_2].$$

A key property of IA is its inclusion monotonicity: if $X \subseteq X'$ and $Y \subseteq Y'$, then for any operation '$\circ$', the result $X \circ Y$ is contained within $X' \circ Y'$. This guarantees that evaluating a function using IA yields an interval enclosing all possible function values, providing mathematical rigor crucial for reliable CAD tasks like constraint solving or collision detection.

However, IA suffers from a significant challenge known as the dependency problem. When the same variable appears multiple times in an expression, IA treats each instance as independent. This lack of correlation often leads to a substantial overestimation of the true range. For example, consider $f(x) = x^2 + x$ for $x \in [-1, 1]$. While the actual range is $[-0.25, 2]$, a naive IA evaluation yields $[-1, 1]^2 + [-1, 1] = [-1, 2]$, a wider interval because the dependency between the two x terms is lost. This interval widening, sometimes called spurious widening or the wrapping effect, can become a major obstacle, often forcing algorithms to perform numerous subdivisions to refine the bounds, thereby increasing computational effort.

Despite this limitation, IA offers the fundamental benefit of guaranteed correctness: the computed interval is certain to contain the true value. This makes it invaluable for applications demanding certainty, such as verifying collision-free assemblies or finding all solutions to equation systems. Its historical use in CAD since the early 1990s, particularly in robust shape intersection and rendering [57, 40], highlights its utility. Furthermore, IA is relatively straightforward to implement and computationally inexpensive per operation, establishing a baseline for self-validated, reliable geometric computation in CAD.

The need to mitigate IA's overestimation, particularly due to the dependency problem, motivated the development of more sophisticated techniques. Affine arithmetic emerged as a significant advancement, designed specifically to track linear correlations between variables and thereby produce tighter, more informative bounds.

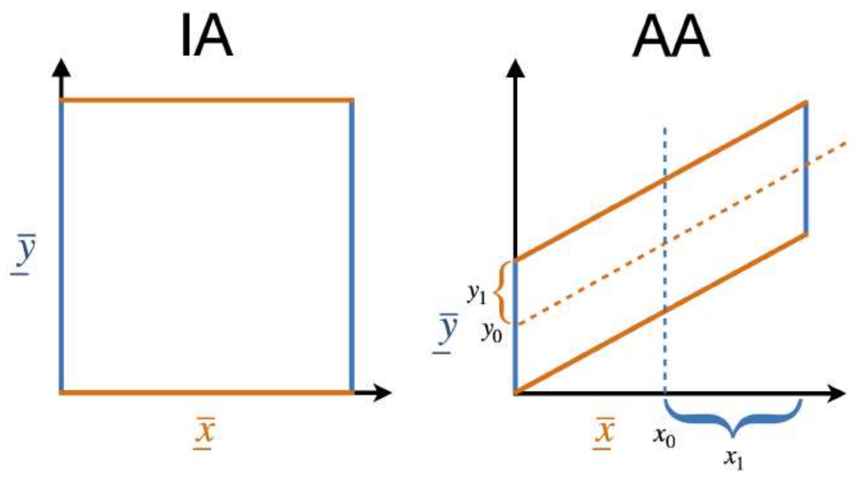


Figure 4.3: Interval bounds resulting from the combination of two interval (left) and affine (right) quantities.

Affine arithmetic

Affine arithmetic refines the interval concept by representing uncertain quantities not just as ranges, but as affine forms. An affine form expresses a quantity $\hat{x}$ as a linear combination of shared, primitive uncertainty sources, called noise symbols ε_i, plus a central value x_0:

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n. \tag{4.12}$$

Each ε_i is an independent variable assumed to lie in $[-1, 1]$, representing a fundamental source of uncertainty. The key insight is that if multiple quantities depend on the same underlying source, they will share the same ε_i term in their affine forms, thus explicitly encoding their dependency. The interval corresponding to $\hat{x}$ can be recovered as $[x_0 - R, x_0 + R]$, where the radius $R = \sum_{i=1}^{n} |x_i|$.

Crucially, AA remembers these linear relationships. For example, consider the expression $\hat{x} - \hat{x}$. If $\hat{x} = 2 + 1\varepsilon_1$, AA correctly computes $(2 + 1\varepsilon_1) - (2 + 1\varepsilon_1) = 0$, reflecting the perfect correlation. In contrast, IA would compute $[1, 3] - [1, 3] = [-2, 2]$, losing the dependency information. This ability to track linear correlations allows AA to avoid much of the overestimation inherent in IA.

Operations in AA are defined to propagate these dependencies. Affine operations (like addition or scalar multiplication) are exact: the resulting affine form is simply the corresponding linear combination of the operand forms, preserving all shared noise symbols. For instance, if $\hat{u} = u_0 + \sum u_i\varepsilon_i$ and $\hat{v} = v_0 + \sum v_i\varepsilon_i$, then their sum is $\hat{u} + \hat{v} = (u_0 + v_0) + \sum(u_i + v_i)\varepsilon_i$.

Non-affine operations (like multiplication, division, or transcendental functions) cannot generally be represented exactly as a new affine form. In these cases, AA employs a best linear approximation of the operation and introduces a new, independent noise symbol ε_{new} to rigorously bound the approximation error. The multiplication $\hat{u} \cdot \hat{v}$, for example, is approximated as:

$$\hat{u} \cdot \hat{v} \approx u_0 v_0 + \sum_i (u_0 v_i + v_0 u_i)\varepsilon_i + (\text{new error}) \cdot \varepsilon_{\text{new}}. \tag{4.13}$$

The coefficient of ε_{new} is chosen to tightly enclose the true range of the nonlinear part of the product. As computations proceed, the affine forms evolve, potentially accumulating more noise symbols with each nonlinear step. The final affine form encapsulates the linear dependencies on initial inputs plus the accumulated approximation errors. Converting back to an interval yields bounds that are typically much tighter than those from IA (see Figure 4.3).

The enhanced precision of AA comes at the cost of increased computational complexity and memory usage. Each affine form stores a central value and potentially many coefficients $(x_1, \ldots, x_n)$. The number of noise symbols, n, can grow with each nonlinear operation, potentially becoming large. Operations on these forms take $O(n)$ time, compared to $O(1)$ for IA, which can become a bottleneck in complex computations. Memory consumption also increases. Consequently, the scalability of pure AA can be limited for very large problems. Strategies like reduced affine arithmetic exist to prune noise terms, sacrificing some precision for better performance. Additionally, implementing nonlinear functions in AA requires careful derivation of approximations and error bounds, adding complexity. This computational expense can restrict applicability of AA in highly demanding interactive or large-scale simulation scenarios.

The performance and scalability challenges of standard AA prompted the development of revised affine arithmetic. RevAA aims to retain much of correlation-tracking benefit of AA while significantly reducing its computational overhead by limiting the growth of noise symbols.

Revised affine arithmetic

Revised affine arithmetic modifies the AA representation by using a fixed, limited number of noise symbols (often corresponding only to the initial input uncertainties) and aggregating all other errors into a single interval term. A RevAA form looks like:

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n + e_x[-1, 1]. \tag{4.14}$$

Here, n is fixed, and $e_x \geq 0$ is a scalar magnitude bounding the accumulated error not captured by the explicit $\varepsilon_1, \ldots, \varepsilon_n$ terms. Instead of introducing new noise symbols for nonlinear operations, RevAA updates the magnitude e_x of this single error interval. This creates a hybrid representation—partially affine (of fixed length), partially interval.

Operations are redefined accordingly. Affine operations combine the linear parts and sum the error magnitudes e_x. Nonlinear operations, like multiplication $\hat{z} = \hat{x} \cdot \hat{y}$, use specific formulas to compute the new linear part and the updated error magnitude e_z, without creating new noise symbols. For example, the multiplication formula includes terms for the linear approximation and an updated error e_{xy}:

$$\hat{x} \times \hat{y} = \left(x_0y_0 + \frac{1}{2}\sum_{i=1}^{n} x_iy_i \right) + \sum_{i=1}^{n}(x_0y_i + y_0x_i)\varepsilon_i + e_{xy}[-1, 1], \tag{4.15}$$

where e_{xy} is calculated based on e_x, e_y, and the coefficients using a formula like:

$$e_{xy} = e_xe_y + e_y(|x_0| + U) + e_x(|y_0| + V) + UV - \frac{1}{2}\sum_{i=1}^{n} |x_iy_i|, \tag{4.16}$$

where $U = \sum |x_i|, V = \sum |y_i|$. The key advantage is that the number of terms ($n+1$ coefficients) remains constant, ensuring bounded memory usage and computational cost per operation. The final interval is $[x_0 - R', x_0 + R']$ where $R' = \sum_{i=1}^{n} |x_i| + e_x$. First proposed by Vu et al. [62], RevAA sacrifices some potential precision (compared to full AA) for significant gains in efficiency, while still tracking the main input dependencies unlike IA.

The primary benefit of RevAA is its greatly improved performance and scalability compared to standard AA. With fixed-size forms and bounded operation costs, it is much faster and uses less memory, making it feasible for complex CAD models and even real-time applications where AA would be too slow [23]. This efficiency makes it suitable for demanding tasks like global optimization, constraint solving [62], and robust geometry processing (collision detection, CSG modeling), offering a compelling balance of reliability and speed.

The main compromise in RevAA is that aggregating all nonlinear errors into a single term e_x can lead to looser bounds compared to full AA, where multiple error sources remain distinct. If a problem involves many significant, independent nonlinear effects, RevAA's single error term might accumulate faster, resulting in more overestimation than AA (though typically still far less than IA). It manages overestimation rather than fully eliminating it. Like IA and AA, careful implementation is needed to handle floating-point round-off errors rigorously.

4.4 Sparse volume data structures

Sparse volume data structures have emerged as a crucial solution for managing high-resolution volumetric models in computer graphics and computer-aided design. As highlighted in Uchytil and Storti's work [61], these structures are essential when dealing with models containing hundreds of billions of voxel (three-dimensional counterpart to a pixel) elements, where memory requirements can reach terabytes of data if stored densely.

The fundamental concept behind sparse volume data structures is the efficient storage and access of volumetric data by only allocating memory for relevant regions of space. One of the most common implementations is the N^3-tree structure, which provides a hierarchical representation of the volume through multiple levels of spatial subdivision [18]. As demonstrated by Uchytil and Storti [61], this approach allows for significant memory savings, particularly when large portions of the volume are empty or contain uniform data.

A notable implementation of sparse volume structures is a VDB tree, developed by Museth [42], which has become an industry standard. Recent research discusses how modern approaches like GVDB [27] and NanoVDB [43] extend these concepts to achieve better performance on graphics hardware. These implementations typically use memory pooling systems to manage the tree topology efficiently, organizing data into pool groups at different tree depth levels.

The effectiveness of sparse volume structures is particularly evident in real-world applications. For instance, in additive manufacturing scenarios discussed by Bader et al. [10], these structures enable the representation of micron-level features across meter-sized build volumes. Uchytil and Storti [61] demonstrate this through a practical example of a wood screw model, where the sparse representation allows for efficient storage and manipulation of high-resolution geometric details.

Recent developments have also introduced dynamic node handling, as presented by Uchytil and Storti [61], where interior regions of models can be constructed on-demand rather than storing the complete volumetric data. This approach further optimizes memory usage while maintaining access to full resolution data when needed. The authors show how this can reduce memory requirements for storing model topology without sacrificing grid resolution.

Performance characteristics of sparse volume structures are heavily influenced by factors such as topology configuration and geometry pool capacity. Beyer et al. [13] provide a comprehensive review of state-of-the-art techniques, while Aleksandrov et al. [9] examine various voxelisation algorithms and data structures. These works demonstrate how these factors affect rendering speed and memory usage, showing that optimal leaf node sizes can vary depending on the specific application requirements.

Looking to the future, research directions in sparse volume structures continue to evolve, with focus areas including view-dependent evaluation, improved interval bounds for topology construction, and enhanced geometry evaluation techniques. Recent work by Kim et al. [31] on NeuralVDB demonstrates how neural networks can be used to learn sparse grid topology, significantly reducing memory requirements. The integration of these structures with modern GPU architectures and their application in real-time rendering and CAD systems represents an active area of development in computer graphics and geometric modeling.

In the proposed research, an octree data structure, which is called a **Sparse Voxel Octree (SVO)** in computer graphics [34], is used, due to its lowest memory consumption and simplicity of implementation. However, the main limitation of SVO is presence of very deep volume regions, where the SVO traversal can give significant overhead. Choosing best sparse volume data structure for geometry representation is an open problem, due to its highly dependence on geometry of visual scene, and it will be covered in future work.

Chapter 5

Proposed adaptive CSG framework

5.1 Efficient representation of CSG tree

The construction of a CSG framework necessitates the memory allocation of CSG tree of F-rep primitives, which is memory efficient and cache friendly. The input typically consists of function compositions expressed in *infix* notation, a conventional format for arithmetic and logical expressions where operators are positioned between operands (e.g., “2 + 2”). Traditional approaches involve constructing an expression tree, which requires recursive traversal for evaluation of CSG tree. This methodology presents significant limitations when dealing with large-scale trees. Consider a scenario with 1000 functions where only two require evaluation within a specific spatial region. The tree structure necessitates traversing a substantial portion of the hierarchy to access these functions, as direct node access is not constant-time. Given that pruning serves as a primary mechanism for accelerating F-rep evaluation, a data structure, which minimizes traversal overhead, should be chosen.

To address these limitations, *postfix* notation, also known as **Reverse Polish Notation (RPN)**, is implemented. This mathematical notation system positions operators after their operands, contrasting with *prefix* notation or **Polish notation (PN)**, where operators precede operands. RPN eliminates the need for parentheses when operators maintain fixed operand counts. The conversion from infix to postfix expressions is achieved through the *shunting-yard algorithm* [19] (see Algorithm 5.1), developed by Edsger W. Dijkstra. Grasberger et al. [25] were first, who analyzed it and showed, that evaluation of BlobTree (an extension of CSG tree with warping and blending operations [64]), transformed in RPN sequence, is faster in the order of magnitude than BlobTree, implemented in a tree structure.

The proposed framework utilizes the transformation of a CSG tree into an RPN sequence, hereafter is called RPN, where operands are F-rep primitives, and the operations are Boolean operators. These primitives are defined as user-specified functions designed for efficient evaluation. This approach fundamentally differs from the methods proposed by Keeter [30] and [61], which construct DAG from arithmetic operations. While their strategies offers greater generality, they result in less efficient simplification of the defining function. The proposed methodology builds upon Duff’s work [20], enabling faster CSG tree evaluation and better pruning capabilities. Additionally, the framework can be extended by incorporating operations such as blending, twisting and other deformations, significantly improving the design flexibility for complex shapes.

For illustration, consider the following function composition:

$$\neg((f_1 \wedge f_2) \vee (f_3 \wedge f_4)). \tag{5.1}$$

Its RPN sequence is:

$$f_1 f_2 \wedge f_3 f_4 \wedge \vee \neg. \tag{5.2}$$

The shunting-yard algorithm generates an array of operands and operators, leading to contiguous memory storage that improves cache utilization. This array structure eliminates traversal overhead, even in large-scale compositions. During function evaluation, it enables constant-time data access, offering a significant advantage over tree-based approaches. Previous research has explored

Algorithm 5.1: Shunting-yard algorithm [19].

```
Require: Infix expression $expr$
Ensure: Postfix expression
  Initialize empty array $Q$, stack $S$
  for token $t$ in $expr$ do
      if $t$ is operand then append to $Q$
      else if $t$ is operator then
          while $S$.top() has higher precedence than $t$ do
              $Q$.append($S$.pop())
          end while
          $S$.push($t$)
      else if $t$ is ’(’ then $S$.push($t$)
      else if $t$ is ’)’ then
          while $S$.top() $\neq$ ’(’ do
              $Q$.append($S$.pop())
          end while
          $S$.pop()
      end if
  end for
  while $S$ not empty do
      $Q$.append($S$.pop())
  end while
  return $Q$
```

stack-based [30] and tree-based [61] representations for function composition. While Keeter’s stack-based method shares similarities with our approach, it relies on stack storage, which suffers from poorer cache performance and linear-time access. Furthermore, maintaining separate pruned stack versions for distinct spatial regions imposes higher memory demands. Uchytil’s expression tree representation improves memory efficiency compared to Keeter’s by using tree bitmasks to track active and inactive components, avoiding redundant tree copies. However, this method incurs slower data access. Our proposed approach integrates these bitmask optimizations (see Section 5.2), achieving both a reduced memory footprint and faster data access.

5.2 Efficient storage and evaluation of pruned CSG tree

Optimizing storage of RPN sequences for volumes in spatial tree structures requires minimizing memory overhead. Following Uchytil and Storti approach, bitmask storage is implemented instead of maintaining multiple RPN copies. A bit value of ‘1’ denotes an *active* RPN node, while ‘0’ indicates an *inactive* or pruned node. Consequently, an unpruned RPN corresponds to a bitmask of all ones. The pruning methodology is detailed in Section 5.3.

To perform fast traversal across bitmask, an algorithm (see Algorithm 5.2), which efficiently computes the linear offset to the next set bit in a bit mask, is implemented. DeBruijn is a static table with 32 entries, and the constant 0x077CB531U is the hex representation of the 32-bit de Bruijn sequence [35]. Note that $b\& - b$ deletes all but the lowest bit (& denotes bitwise AND), while $\gg 27$ isolates the upper five bits ($\gg$ denotes bitwise right shift). Multiplication by the de Bruijn constant makes the upper five bits unique for each power of two, and the table lists the corresponding bit position of the lowest bit in b. Also note that the last line of the code computes the lowest bit position without branching, that is, conditionals. This technique allows us to efficiently iterate through RPN sequence without the necessity of copying it for each volume.

Algorithm 5.2: Computation of offset to next set bit in mask.

```
Require: bitmask mask, current offset
Ensure: linear offset to next set bit
  $n \leftarrow$ offset $\gg 5$
  $m \leftarrow$ offset&31
  $b \leftarrow$ mask$[n]$
  if $b\&(1 \ll m)$ then return offset                ▷ Trivial case: current bit is set
  end if
  $b \leftarrow b\&(0xFFFFFFFF \ll m)$                  ▷ Mask lower bits
  while $b = 0$ do
      $n \leftarrow n + 1$
      $b \leftarrow$ mask$[n]$                          ▷ Find next non-zero block
  end while
  return $(n \ll 5) +$ DeBruijn$[(b\&-b) * 0x077CB531U \gg 27]$
```

Similar approach is for fast traversal in reverse direction of bitmask. Complete algorithm of evaluation of RPN sequence is presented by Algorithm 5.3. In this thesis, common approach of RPN evaluation using stack is implemented.

5.3 CSG pruning

CSG simplification, or *pruning*, represents a fundamental mechanism for optimizing function evaluation efficiency. The proposed methodology for CSG pruning is based on R-function properties for set-theoretic operations. The companion function of R-functions provides significant opportunities for simplifying CSG trees.

Consider an object consisting exclusively of set-theoretic operations (union, intersection, and negation). For example, an F-rep defining function

$$F(f_1, f_2) = f_1 \vee f_2 \tag{5.3}$$

is expressed in ternary logic as

$$S[F(f_1, f_2)] = K[S(f_1), S(f_2)]. \tag{5.4}$$

When examining a specific region where only the object defined by function f_1 exists (i.e., $f_2 < 0$ and $S(f_2) = 0$), the equation transforms to:

$$S[F(f_1, f_2)] = K[S(f_1), S(f_2)] = K[S(f_1), 0] = S(f_1) \tag{5.5}$$

This transformation demonstrates that the union function F can be substituted by function f_1 while preserving the geometric properties of the object, as the sign distribution remains invariant. This principle extends to intersection operations and forms the basis of the proposed pruning methodology. The pruning process maintains the original RPN sequence while generating a modified bit mask with deactivated bits. For instance, a complete bitmask '111' transforms to a pruned variant '010', while maintaining analytical equivalence in object representation.

The introduction of blending operations introduces additional complexity to set-theoretic operation pruning. As blending operations are non-R-functions, which depend on magnitude and sign, operations preceding them become non-prunable. However, subsequent operations retain their pruning capability. To address this limitation, localized (bounded) blending operations (detailed in Section 4.1) can be used. They confine the blending effect to a defined spatial region while

Algorithm 5.3: CSG tree evaluation with bitmask iteration.

```
function EvaluateCSG(rpn, bitmask, point)
    stack ← $\emptyset$
    $i$ ← next_bit(bitmask, 0)
    while $i <$ rpn.size() do
        term ← rpn[$i$]
        if term is Function then
            stack.push(term(point))
        else if term is UnaryOperation then
            operand ← stack.pop()
            stack.push(term(operand))
        else                                                ▷ BinaryOperation
            right ← stack.pop()
            left ← stack.pop()
            stack.push(term(left, right))
        end if
        $i$ ← next_bit(bitmask, i + 1)
    end while
    return stack.top()
end function
```

ensuring analytical vanishing outside this domain. This characteristic enables:

1. Pruning of blending operations outside their defined regions,
2. Preservation of set-theoretic operation pruning capabilities,
3. Efficient handling of multiple blending zones without significant performance degradation.

Dimensions of the blending region vary depending on the participating objects but typically occupy substantially less space than the original geometries. Consequently, blending operations are frequently prunable, allowing set-theoretic operation pruning to proceed in most spatial regions.

Algorithm 5.4 presents the pruning methodology for set-theoretic operations, with bounded blending operations to be incorporated in future work. Here are the key steps of the RPN pruning algorithm:

1. Initialize bitmask copy, interval data array, and empty stack;
2. For each active bit in bitmask:
   - If term is a function: evaluate at point and push to stack,
   - If term is negation: apply to top stack element,
   - If term is binary operation (union/intersection):
     - Pop two operands from stack,
     - Apply operation,
     - Check if subtree can be pruned based on intervals,
     - If prunable, update bitmask and prune subtree,
     - Push result to stack;
3. Check final stack element:

    - If its interval allows pruning, prune its subtree;

4. Return updated bitmask.

The algorithm employs range evaluation techniques (described in Section 4.3) for each RPN element. Set-theoretic operations become prunable when a function's range does not intersect the zero point.

The pruning of a subtree for a prunable element is described in Algorithm 5.5. The algorithm proceeds as follows:

1. Input: RPN sequence, bitmask, number of elements to prune, starting index;
2. While elements remain to prune:
    - Clear the current bit in the bitmask,
    - Decrement the pruning counter,
    - Move to the previous active bit in the bitmask,
    - Check whether the pruning is complete;
3. For each processed element:
    - If a negation operation is encountered, add two additional elements to prune,
    - If a binary operation (union/intersection) is encountered, add three additional elements to prune.

Algorithm 5.4: Pruning of CSG tree.

```
Require: RPN sequence $rpn$, bitmask $bitmask$, coordinate $r$
Ensure: Pruned bitmask
1: Initialize $bitmask\_res \leftarrow bitmask$, array $I\_data[rpn.size()]$, stack $S$, $i \leftarrow$ nextbit$(bitmask\_res, 0)$
2: while $i < rpn.size()$ do
3:     $term \leftarrow rpn[i]$
4:     if $term$ is function or negation then
5:         $prev \leftarrow$ (is negation) ? $\neg I\_data[S.pop()]$ : $term(r)$
6:         $I\_data[i] \leftarrow prev$; $S.push(i)$
7:     else                                                    ▷ Binary operation
8:         $r, l \leftarrow S.pop(), S.pop()$
9:         $I\_data[i] \leftarrow$ ApplyOperation($term$, $I\_data[l]$, $I\_data[r]$); $S.push(i)$
10:        if IsPrunable($term$, $I\_data[l]$, $I\_data[r]$) then
11:            Clear bit $i$ in $bitmask\_res$
12:            $(prune, keep) \leftarrow$ DeterminePruneBits($term$, $l$, $r$)
13:            Replace stack top with $keep$
14:            pruneSubRPN($rpn$, $bitmask\_res$, GetPrunedNum($rpn[prune]$), $prune$)
15:        end if
16:    end if
17:    $i \leftarrow$ nextbit$(bitmask\_res, i + 1)$
18: end while
19: if IsPrunable($I\_data[S.top()]$) then
20:     prunePreRPN($rpn$, $bitmask\_res$, GetPrunedNum($rpn[S.top()]$), $S.top()$)
21: end if
      return $bitmask\_res$
```

Algorithm 5.5: Pruning of a CSG subtree in an RPN sequence.

```
Require: RPN sequence $rpn$, bitmask $bitmask$, number of nodes to prune $pruned\_num$, starting
    index $index$
 1: while $pruned\_num > 0$ do
 2:     Clear bit at position $index$ in $bitmask$
 3:     $pruned\_num \leftarrow pruned\_num - 1$
 4:     $index \leftarrow \texttt{prevbit}(bitmask, index)$
 5:     if $pruned\_num = 0$ then
 6:         break
 7:     end if
 8:     $term \leftarrow rpn[index]$
 9:     if $term$ is a Negation then
10:         $pruned\_num \leftarrow pruned\_num + 2$
11:     else if $term$ is a Union or Intersection then
12:         $pruned\_num \leftarrow pruned\_num + 3$
13:     end if
14: end while
```

This algorithm recursively prunes dependent elements in the RPN sequence by iteratively updating the bitmask.

The pruning methodology for set-theoretic operations represents a key contribution of this thesis. By combining efficient CSG compression with enhanced pruning via F-rep Boolean operations, the approach achieves substantial improvements in computational efficiency while preserving geometric invariance.

5.4 SVO construction

Spatial sampling constitutes the final major challenge addressed in this work. As discussed in Section 4.4, a variety of tree data structures exist, each with distinct memory footprints and data access complexity. Despite these differences, they share fundamental organizational principles, enabling relatively straightforward transitions between implementations. This thesis employs a Sparse Voxel Octree structure, selected for its widespread adoption and implementation simplicity.

The SVO construction process is outlined in Algorithm 5.6. Following RPN sequence pruning for a target spatial region, the algorithm assesses the count of active primitive functions and Boolean operations. If this count exceeds a predefined threshold, the d-dimensional region is subdivided into $n = 2^d$ equal subregions ($n = 8$ for an octree), and the RPN sequence is recursively pruned for each subregion. This adaptive subdivision criterion allows the octree to achieve greater depth in geometrically dense regions while retaining shallower depths in empty spaces. Our methodology diverges from Keeter [30] and Uchytil [61], who employ fixed tree depths, mandating that non-empty spaces undergo subdivision until reaching a predefined depth. Unlike our method, their approaches lack a threshold-based termination condition for recursion in non-empty regions. Figure 5.1 illustrates the resulting SVO structure, where empty regions exhibit minimal depth, while areas with high primitive density display maximal depth.

The proposed sparse volume data structure demonstrates spatial adaptivity, with the subdivision threshold acting as a tunable optimization parameter. The threshold selection can be guided by multiple factors: the number of primitives in the CSG tree, memory constraints, primitive complexity, and the complexity of non-R-functions. However, the identification of optimal threshold values remains a topic for future research, balancing computational efficiency against geometric fidelity.

Algorithm 5.6: Adaptive SVO construction.

```
Require: CSG tree $f$ (RPN sequence), root region $\mathcal{R}_0$, threshold $\tau$
Ensure: Sparse voxel octree whose every node stores a pruned $f$
 1: procedure BuildSVO($f, \mathcal{R}, \tau$)
 2:     $f' \leftarrow$ Prune($f, \mathcal{R}$)          ▷ Remove inactive sub-trees w.r.t. $\mathcal{R}$
 3:     $n \leftarrow$ CountNodes($f'$)          ▷ # active CSG nodes left
 4:     if $n > \tau$ then          ▷ Too complex $\rightarrow$ subdivide
 5:         $\{\mathcal{R}_i\}_{i=1}^{8} \leftarrow$ Subdivide($\mathcal{R}$)          ▷ 4 children for quadtree
 6:         for all $\mathcal{R}_i$ do
 7:             BuildSVO($f'$, $\mathcal{R}_i$, $\tau$)          ▷ Recurse on sub-regions
 8:         end for
 9:     else          ▷ $n \leq \tau$ $\rightarrow$ stop subdivision
10:         Store pruned F-rep $f'$ in current node
11:     end if
12: end procedure
```

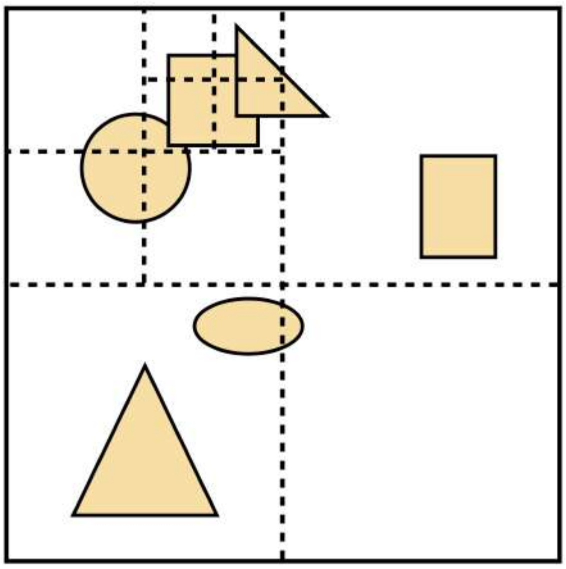


Figure 5.1: Constructed SVO for a CSG tree.

5.5 Evaluation complexity

The time required to evaluate a CSG tree stored in an SVO—hereafter termed the adaptive CSG tree—is governed by four consecutive steps:

1. **Point selection.** Pick a point $\mathbf{x}$ in the modelling domain.
2. **Locating the SVO leaf.** Traverse the SVO from the root to the leaf cell $\mathcal{C}$ that contains $\mathbf{x}$. If $K(L)$ is the depth of that leaf (a non-increasing function of the pruning threshold L), the cost is $\mathcal{O}\big(K(L)\big)$.
3. **Fetching the bit mask.** Each leaf stores a *compressed* representation (bitmask) of the pruned CSG tree of F-rep primitives. Retrieving it is a direct array access $\mathcal{O}(1)$.
4. **Evaluating the pruned CSG tree.** The leaf's CSG expression contains at most L tree nodes, so its evaluation costs $\mathcal{O}(L)$.

The total evaluation complexity comprises two dominant components: SVO traversal and pruned CSG tree evaluation:

$$T(L) \;=\; C_1\, K(L) \;+\; C_2\, L \;+\; C_3,$$

where C_1, C_2, C_3 are implementation-dependent constants (e.g., memory access latency, instruction pipelining efficiency). The proposed spatial sampling algorithm ensures that pruned CSG tree exhibits **constant evaluation complexity**, controlled by the subdivision threshold, at the cost of introducing SVO traversal overhead. Critically, since spatial sampling is decoupled from the volume data structure, identifying the optimal structure to minimize traversal time and memory consumption remains an open research question.

While the derived complexity is numerically valid, it requires adjustment for average-case analysis. For an octree, the probability of sampling a leaf at depth K is 2^{-3K}; for a quadtree, it is 2^{-2K}. Thus, the spatially-averaged evaluation complexity becomes:

$$\overline{T}(L) = C_1 \sum_{i=0}^{N} \frac{K_i}{2^{dK_i}} + C_2 \sum_{i=0}^{N} \frac{L_{K_i}}{2^{dK_i}} + C_3, \tag{5.6}$$

where d is the domain dimensionality, L_{K_i} denotes the pruned CSG tree complexity at depth K_i, and N is the number of SVO leaves. This formulation demonstrates that deeper leaves (higher K_i) contribute negligibly to the average complexity due to their exponentially diminishing probability.

As shown in Section 6, for real-world scenes, evaluation time is dominated by the pruned CSG tree evaluation. The SVO traversal introduces a constant overhead but does not significantly alter the asymptotic behavior of the algorithm.

5.6 Visualization

The visualization or rendering of an F-rep model can generally be accomplished through two principal approaches. The first method involves polygonization of implicit surfaces. This technique converts an implicit surface into B-rep using algorithms such as marching cubes [37] or dual contouring [29], subsequently rendering the 3D scene through rasterization or ray-tracing. While this approach benefits from high computational efficiency and GPU hardware acceleration, it inherently introduces approximation errors that may compromise geometric fidelity. Although diminishing the facet size can improve precision, this strategy exponentially increases memory requirements. Despite these limitations, polygonization remains one of the most computationally efficient and widely adopted visualization methods.

The second approach, implemented in this thesis, employs direct ray-tracing of implicit surfaces. This method originates from principles of physical light transport, approximated through geometric optics. Unlike forward light path simulation (from light sources to the observer), ray-tracing employs backward propagation by casting rays from the viewpoint (e.g., camera or virtual eye) and computing their interactions with scene objects. This methodology inherently supports advanced optical phenomena including specular reflections, refractive transmissions, and global illumination effects, making it indispensable for photorealistic rendering in modern graphics engines. In this work, a specific variant of ray-tracing called *ray-casting*, is implemented. Ray-casting computes only the first visible intersection point between a cast ray and scene geometry. For F-rep objects, this reduces to identifying the first root of the implicit function along each ray.

The root-finding problem constitutes a classical challenge in numerical analysis with particular relevance to computer graphics. Commonly, root-finding algorithms are divided into two classes: single-point and bracketing. Single-point methods, like Newton-Raphson and secant methods, require only one initial guess of root, and using extra data, e.g. derivative, can converge to

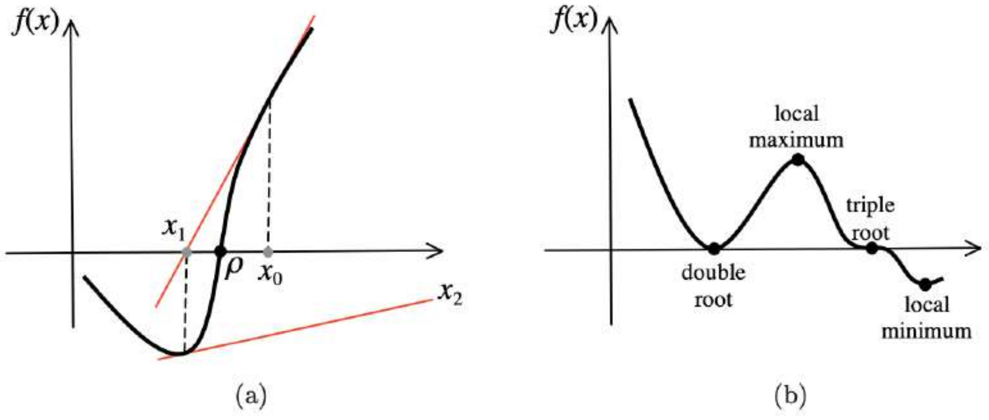


Figure 5.2: (a) Problem I: divergence; (b) Problem II: null derivative [24].

root very fast. However, it may fail to converge under following two circumstances:

1. *Problem I*—The initial guess is far from the root. The convergence of the Newton method is only guaranteed if the starting estimate is "sufficiently close" to the root. Otherwise, the method risk to converge slowly or even diverge (see Figure 5.2(a)).
2. *Problem II*—The derivative is very small or vanishes. The Newton iteration formula requires that the derivative f' does not vanish, i.e. $f'(x) \neq 0$. This means that method blows up at local extrema (i.e. local minima and maxima) and inflection points (see Figure 5.2(b)).

To address these limitations, bracketing methods (e.g., bisection, false position) operate on interval guarantees containing at least one root. While convergence rates are generally slower than single-point methods, these algorithms provide deterministic termination assurances. Our implementation enhances the classical bisection method through interval analysis techniques (IA, AA, RevAA), enabling efficient rejection of root-free regions. By mapping the ray parameter t to spatial intervals $(x(t), y(t), z(t))$ and evaluating the interval extension of the implicit function, the intervals where $0 \notin f([x, y, z])$, can be excluded, significantly accelerating root isolation (see Algorithm 5.7)

Ray-casting through the SVO proceeds exactly as outlined in Algorithm 5.8 and illustrated in Figure 5.3. Starting from the camera, the algorithm marches the ray from leaf to leaf, using the octree's hierarchical structure to identify the next voxel segment in constant time. Whenever the ray enters a *green* leaf, the voxel is known to be empty, so the algorithm skips directly to the exit point and continues; when it encounters a *blue* (filled) leaf, it retrieves the pruned CSG tree stored there and performs a one-dimensional root search for the corresponding implicit function along the ray segment bounded by entry of the voxel and exit parameters. If a root is found, the first intersection point and its normal are reported; otherwise, the traversal advances to the next leaf and repeats. Because empty space is discarded early and the CSG tree in any filled leaf contains at most the threshold number of nodes, this strategy keeps both traversal and evaluation costs low, yielding a fast, memory-efficient rendering of spatial representation of CSG tree.

Algorithm 5.7: Bisection method modified with IA for F-rep ray intersection.

```
Require:
    $expr$: F-rep expression tree
    $ray$: Ray with origin $\mathbf{o}$ and direction $\mathbf{d}$
    $\tilde{t}$: Initial parameter interval $[t_{min}, t_{max}]$
    $\epsilon$: Convergence threshold
Ensure:
    First root $t_r$ along ray or sentinel value $-1$
 1: Initialize stack $\mathcal{S}$ with $\tilde{t}$
 2: while $\mathcal{S} \neq \emptyset$ do
 3:     Pop interval $\tilde{t}_c = [t_0, t_1]$ from $\mathcal{S}$
 4:     Compute spatial coordinates:
 5:     $\mathbf{r} \leftarrow \mathbf{o} + [t_0, t_1] \cdot \mathbf{d}$                    ▷ 3D interval coordinates
 6:     Evaluate F-rep interval:
 7:     $[f_0, f_1] \leftarrow expr(\mathbf{r})$                         ▷ Interval evaluation
 8:     if $f_0 \cdot f_1 \leq 0$ then                       ▷ Possible root in interval
 9:         if $t_1 - t_0 < \epsilon$ then                        ▷ Convergence check
10:             if sign($expr(ray(t_0))$) $\neq$ sign($expr(ray(t_1))$) then
11:                 return $t_0$                    ▷ Return first root approximation
12:             end if
13:         else                                               ▷ Bisect interval
14:             $t_2 \leftarrow \frac{t_0+t_1}{2}$
15:             Push $[t_2, t_1]$ to $\mathcal{S}$
16:             Push $[t_0, t_2]$ to $\mathcal{S}$
17:         end if
18:     end if
19: end while
20: return $-1$                                               ▷ No root found
```

Algorithm 5.8: Ray–casting through an SVO of cells with pruned CSG trees.

```
Require: Ray origin $\mathbf{o}$, unit direction $\mathbf{d}$, SVO root node $\mathcal{O}$
Ensure: First surface hit $(\mathbf{p}, \mathbf{n})$ or $\emptyset$
 1: $(v,\ t_{\text{enter}},\ t_{\text{exit}}) \leftarrow$ FirstLeafOnRay($\mathcal{O}, \mathbf{o}, \mathbf{d}$)
 2: while $v \neq \emptyset$ do                    ▷ Step 2: iterate voxels crossed by the ray
 3:     if IsEmpty(v) then                          ▷ Step 3: skip empty (green) voxels
 4:         $(v,\ t_{\text{enter}},\ t_{\text{exit}}) \leftarrow$ NextLeafOnRay($v, \mathbf{d}, t_{\text{exit}}$)
 5:         continue
 6:     end if
 7:     $f \leftarrow$ pruned F–rep stored in $v$
 8:     $t_\star \leftarrow$ FindRootOnSegment($f,\ t_{\text{enter}},\ t_{\text{exit}},\ \varepsilon$)       ▷ Step 4: test red segment
 9:     if $t_\star \neq \emptyset$ then
10:         $\mathbf{p} \leftarrow \mathbf{o} + t_\star \mathbf{d}$
11:         $\mathbf{n} \leftarrow \nabla f(\mathbf{p})\ /\ \|\nabla f(\mathbf{p})\|$
12:         return $(\mathbf{p}, \mathbf{n})$                              ▷ hit found, terminate
13:     end if
14:     $(v,\ t_{\text{enter}},\ t_{\text{exit}}) \leftarrow$ NextLeafOnRay($v, \mathbf{d}, t_{\text{exit}}$)
15: end while
16: return $\emptyset$                                    ▷ no intersection on the ray
```

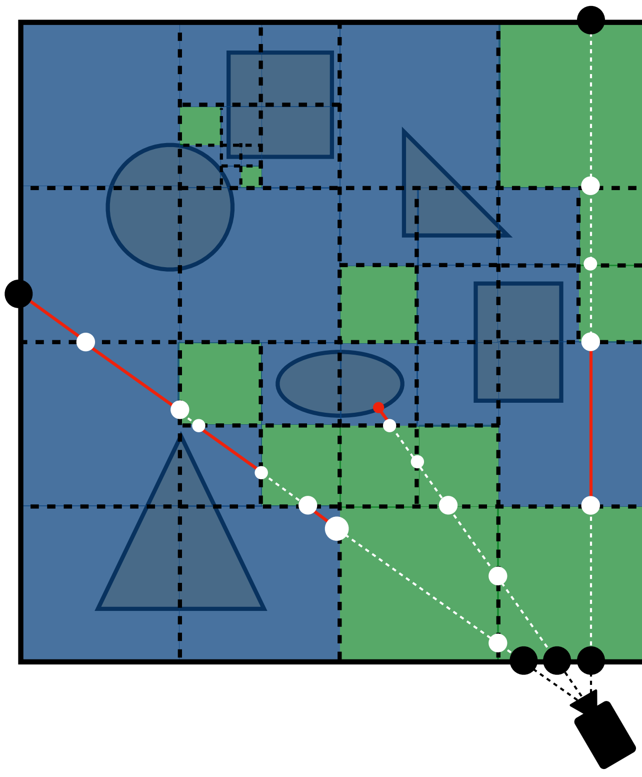


Figure 5.3: Ray traversal through a Sparse Voxel Octree. The camera emits a ray that moves from leaf to leaf: dashed white segments show the portions that pass entirely through *green* empty voxels, while solid red segments indicate testing inside *blue* filled voxels whose pruned F-rep masks may contain the surface. The first successful root on a red segment yields the visible intersection point and terminates the traversal.

Chapter 6

Results and discussion

The proposed adaptive CSG framework was implemented in a C++ prototype geometry kernel. A prototype ray tracer following the methodology of [55] was also developed. To evaluate pruning strategies, three interval libraries were employed:

- **Interval Arithmetic (IA)**: implementation from the Boost C++ library [2],
- **Affine Arithmetic (AA)**: open-source *libaffa* library [5],
- **Revised Affine Arithmetic (RevAA)**: modified version of the library provided by the authors of [39].

This section presents: (1) evaluation of pruning performance on uniform voxel grids using IA, AA, and RevAA in Section 6.1; (2) analysis of SVO construction results across diverse 3D scenes and threshold values in Section 6.2; (3) performance assessment of the adaptive CSG tree in Section 6.3; (4) demonstration of rendering outcomes for pruned CSG trees in Section 6.4; and (5) comparison of the proposed methodology against state-of-the-art algorithms in Section 6.5.

All experiments in Section 6.2 were conducted on a MacBook Air M1 system with 8 GB RAM. Results in other sections were generated on an Intel® Core™ i7-8700 CPU platform with 32 GB RAM.

6.1 Analysis of range evaluation methods for uniform voxel grids

This section evaluates and compares the performance of various interval analysis methods—IA, AA, RevAA—for uniform voxel grids. By systematically measuring computational overhead and pruning efficiency across these methods, the analysis provides detailed insights into their strengths and limitations, enabling informed method selection for specific modeling tasks.

First, IA, AA and RevAA methods for the pruning algorithm are tested. The spatial domain of objects is subdivided uniformly into equal voxels, that are then sequentially pruned using the

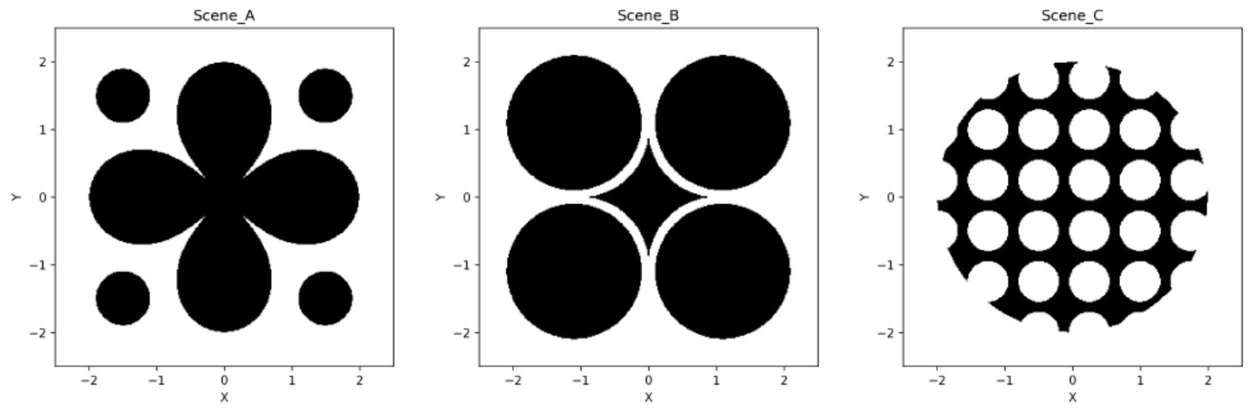
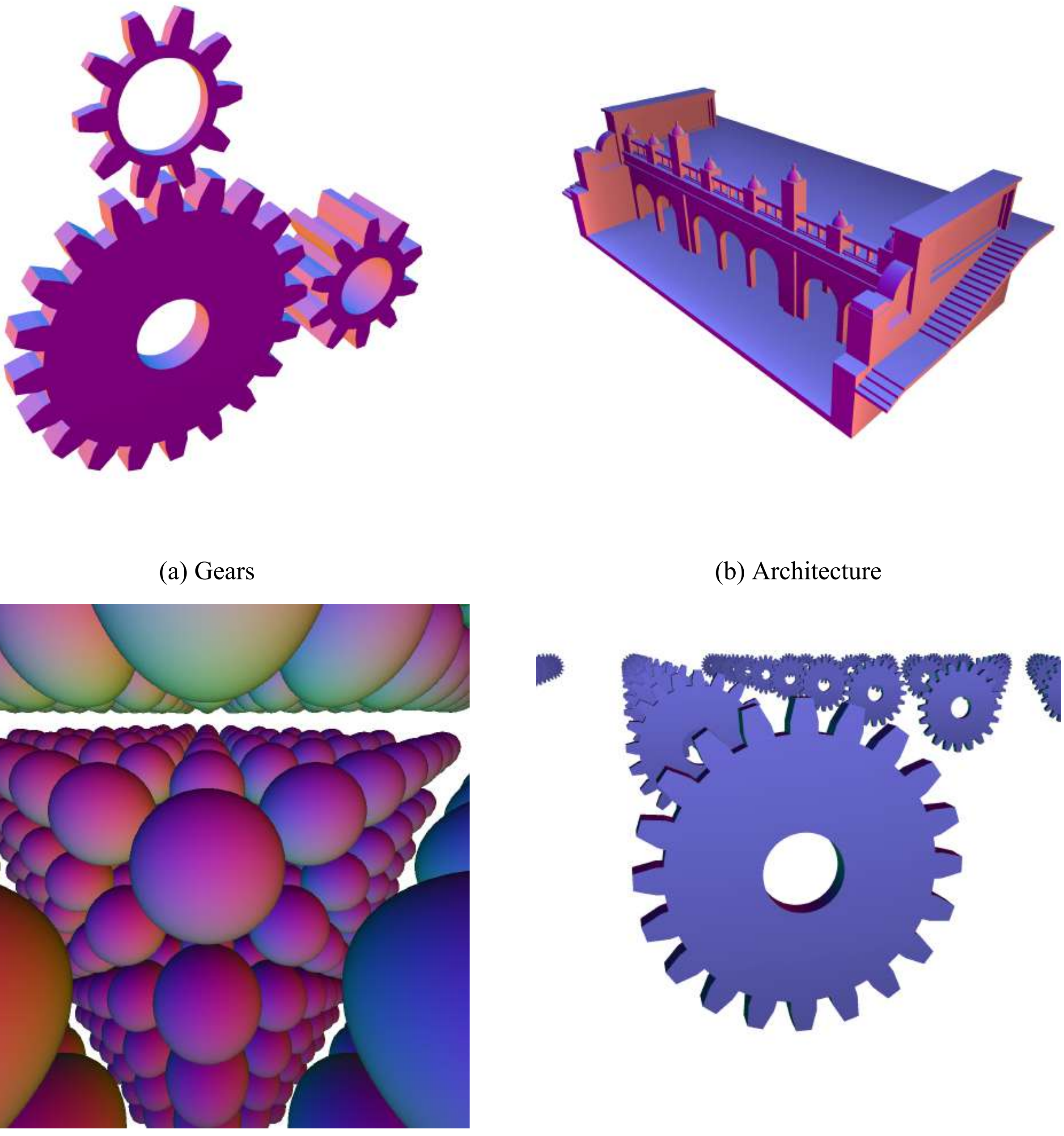


Figure 6.1: 2D scenes for testing the pruning algorithm.



(a) Gears (b) Architecture

(c) Many Spheres (d) Many Gears

Figure 6.2: 3D scenes rendered with RevAA-based ray-casting of adaptive CSG tree.

algorithm from Section 5.3, and the average time is measured. The performance has been evaluated for both 2D and 3D scenes (Scenes A, B, and C in Figure 6.1, Scene Gears in Figure 6.2). Scene A consists of two Bernoulli lemniscates and five circles; Scene B is built by taking unions and subtractions of circles with varying radii; and Scene C is a periodic pattern formed by subtracting many small circles from a large one. Scenes A and B are designed to test range evaluation techniques for polynomial implicit functions, whereas Scene C evaluates these techniques for R-functions. In Figures 6.3–6.5, each panel shows the shape in black overlaid on a uniform grid, where cell color encodes the degree of region compression (darker = more aggressive pruning).

In Scenes A and B, IA suffers from significant overestimation on polynomial primitives, resulting in many unpruned (red/yellow) cells. Both AA and RevAA handle polynomials much better, so they prune nearly identically and more effectively than IA. In Scene C, however, IA briefly

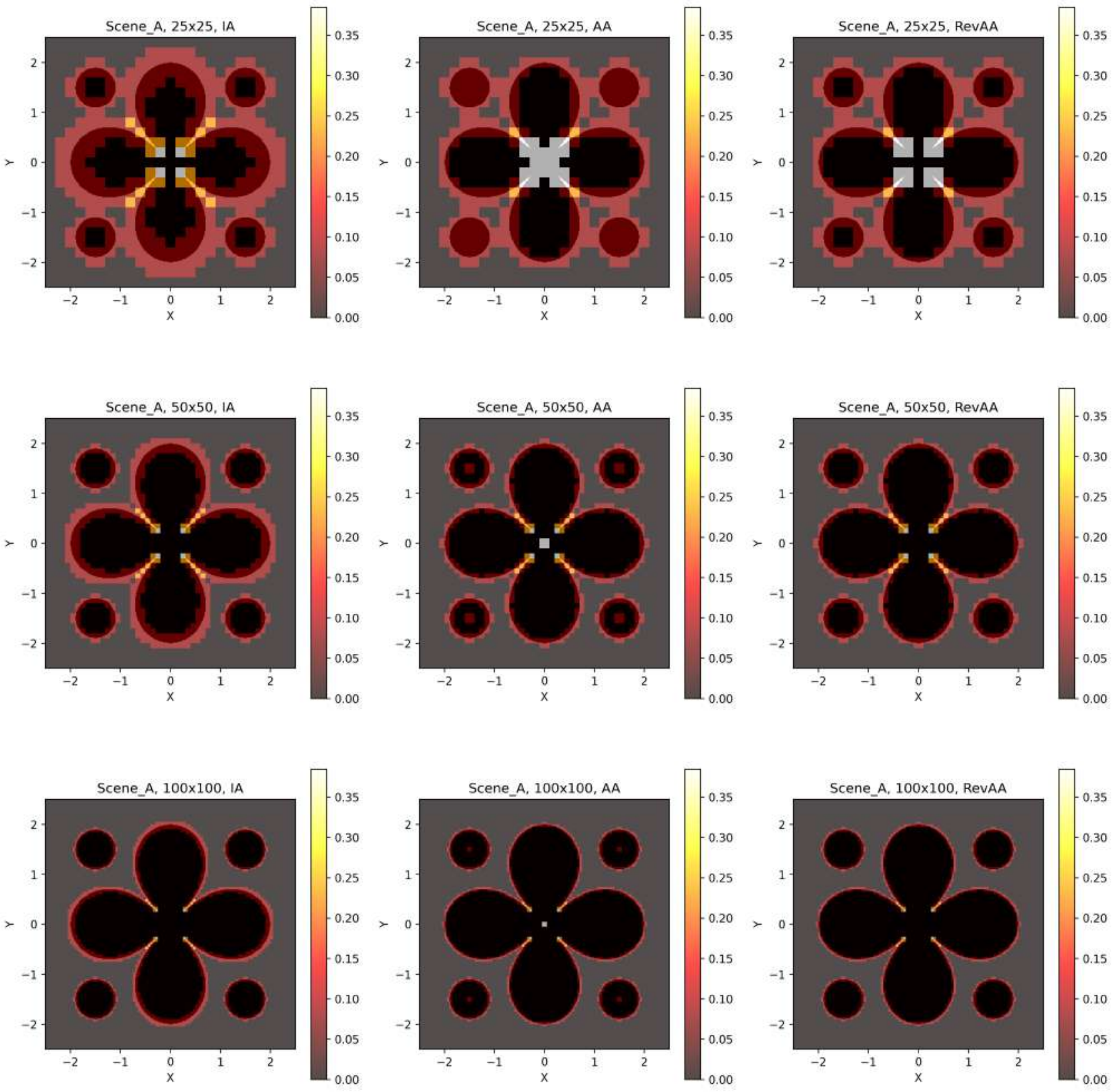


Figure 6.3: Comparison of pruning algorithm for Scene A at increasing grid resolutions. Color gradient indicates the extent of CSG compression (darker = more compression).

outperforms AA and RevAA on coarse grids: the scene's many non-polynomial CSG operations yield tighter IA bounds for large cells. As cell sizes shrink, AA and RevAA regain the advantage thanks to quadratic convergence, in contrast to linear convergence of IA.

Table 6.1 reports the average pruning time per voxel (in milliseconds) and the RPN length (number of CSG tree nodes) of each scene's CSG tree. Boldface indicates the fastest method in each row. RevAA is consistently the fastest—on average about $4\times$ faster than IA and $45\times$ faster than AA. Since RevAA matches pruning quality of AA, it is the clear winner for pruning applications. Pruning time also scales approximately linearly with RPN length when primitives have similar complexity.

A more complex Gears scene (see Figure 6.2) is also tested. In this scene, RevAA runs approximately $4\times$ faster than IA and $60\times$ faster than AA, confirming its superior performance on both 2D and 3D benchmarks.

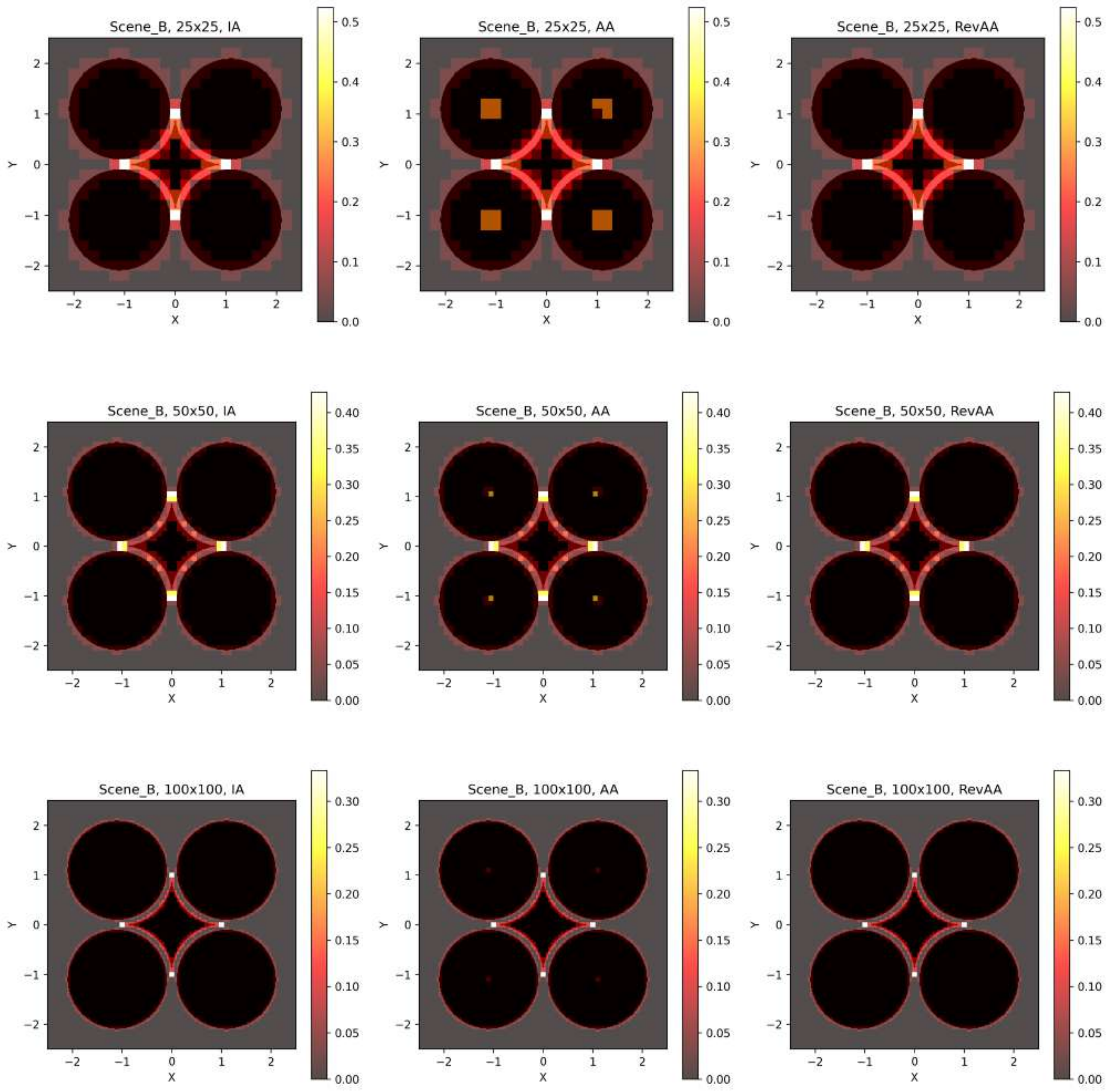


Figure 6.4: Comparison of pruning algorithm for Scene B at increasing grid resolutions. Color gradient indicates the extent of CSG compression (darker = more compression).

The speed of CSG tree evaluation for spatially pruned scenes and their unpruned counterparts is subsequently compared. Performance is measured by averaging CSG tree evaluation times at random points within the spatial domain. Table 6.1 shows average per point CSG tree evaluation times (in nanoseconds) for Scenes A, B, C, and the Gears scene. For fixed voxel counts, all three pruning methods (IA, AA, RevAA) demonstrate comparable performance. However, IA achieves faster CSG evaluation at larger voxel sizes due to its coarser pruning and superior culling efficiency. Conversely, AA and RevAA exhibit better performance at finer resolutions through more aggressive pruning, as theoretically expected.

The Gears scene exhibits the highest computational complexity, with CSG tree evaluation times at coarse (20^3) partitioning exceeding simpler scenes by orders of magnitude. However, refinement to finer voxel sizes dramatically reduces these times: at 20^3 partitioning, evaluation time per voxel of the Gears scene matches those of Scenes A–C despite its greater geometric complexity.

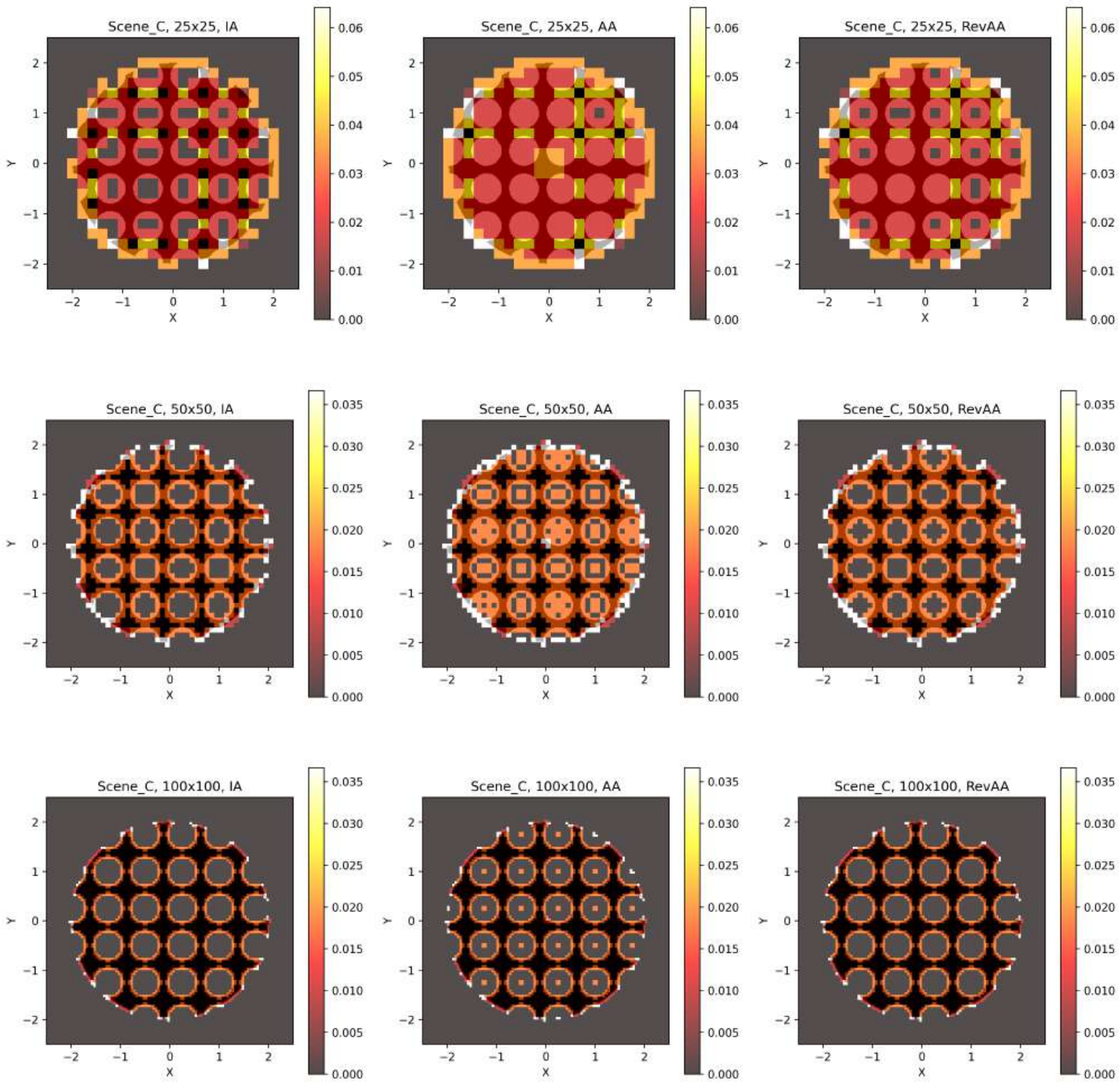


Figure 6.5: Comparison of pruning algorithm for Scene C at increasing grid resolutions. Color gradient indicates the extent of CSG compression (darker = more compression).

Table 6.2 presents minimum, maximum, and average speedup ratios across all partitioning schemes. The smallest improvements occur at coarse (20^3) resolutions due to interval overestimation and high primitive density per voxel in complex scenes. Notably, the Gears scene achieves a $17\times$ speedup even at 5^3 resolution – comparable to the best result in the Scene C – demonstrating the disproportionate benefits of pruning in complex models. Maximum gains reach $180\times$ speedup for the Gears at 80^3 partitioning.

These results highlight pure CSG tree evaluation speedups, free from spatial data structure overhead. Key observations reveal that pruned CSG evaluation times remain consistent across scenes, while original CSG evaluation times scale linearly with RPN sequence length. However, uniform grids exhibit two critical limitations: memory overconsumption, where empty voxels store redundant bit masks (consuming 35 MB for the Gears scene at 80^3 resolution despite optimization), and spatial rigidity, as fixed partitioning precludes adaptive detail focusing in geometrically complex regions.

Scene	RPN Len.	N_{voxels}	Pruning time (ms)			Evaluation time (ns)			
			IA	AA	RevAA	IA	AA	RevAA	Original
A	13	12^2	0.026597	1.594889	**0.00209**	0.151256	0.142345	**0.134512**	0.333334
		25^2	0.121000	1.026954	**0.00735**	0.120922	0.107388	**0.102070**	
		50^2	0.104312	0.481388	**0.022393**	0.093026	0.087280	**0.085088**	
		100^2	0.052760	0.288688	**0.018222**	0.084032	**0.077054**	0.077500	
		200^2	0.026556	0.238128	**0.011895**	0.074988	**0.071149**	0.075335	
		400^2	0.016813	0.226714	**0.00529**	0.074201	0.071113	**0.070054**	
B	21	12^2	0.021903	2.048535	**0.010708**	**0.135572**	0.162452	0.150457	0.431783
		25^2	0.152886	1.070888	**0.023208**	**0.103185**	0.105684	0.104150	
		50^2	0.111126	0.543312	**0.021323**	0.089956	**0.085836**	0.086438	
		100^2	0.064736	0.344306	**0.023013**	0.079744	**0.076627**	0.078837	
		200^2	0.031975	0.297223	**0.012784**	0.073745	0.076085	**0.072815**	
		400^2	0.020784	0.282687	**0.006336**	0.073760	0.070526	**0.070138**	
C	109	12^2	0.503076	4.452444	**0.127799**	**0.158375**	0.172607	0.167700	1.298400
		25^2	0.503206	2.164192	**0.105346**	**0.125002**	0.134785	0.129600	
		50^2	0.255265	1.415828	**0.083410**	**0.099188**	0.111439	0.103781	
		100^2	0.127284	1.228090	**0.052493**	0.089834	0.088647	**0.083830**	
		200^2	0.084320	1.186995	**0.025351**	0.082547	0.079024	**0.078960**	
		400^2	0.071903	1.180538	**0.014721**	0.075260	**0.073094**	0.075414	
Gears	571	5^3	2.565592	79.80448	**0.448344**	**0.516939**	0.517114	0.518626	8.922600
		10^3	0.975700	79.97710	**0.399905**	0.183808	**0.176739**	0.177581	
		20^3	0.530929	—	**0.142931**	0.091978	—	**0.090160**	
		40^3	0.464148	—	**0.078296**	0.061811	—	**0.060916**	
		80^3	0.455615	—	**0.072933**	0.049878	—	**0.049304**	

Table 6.1: Average pruning time per voxel (ms) and CSG tree evaluation time per point (ns). The "Original" sub-column gives the baseline cost of the unpruned expression for each scene.

Scene	Lowest		Highest		Avg.
	N_voxels	Ratio	N_voxels	Ratio	Ratio
Scene A	12^2	2.478100	400^2	4.758234	3.704211
Scene B	12^2	3.184896	400^2	6.156195	4.760794
Scene C	12^2	8.198267	400^2	17.763567	12.596680
Scene Gears	5^3	17.260425	80^3	180.962722	49.900200

Table 6.2: Lowest, highest, and average performance improvement ratios per scene.

Scene	RPN length	Time per point, μs	Time per pixel, ms
Gears	506	6	20
Architecture	1 431	20	3 990
Many Spheres	5 489	75	16 000
Many Gears	30 377	409	840 885

Table 6.3: Average times for original (unpruned) CSG tree evaluation and rendering.

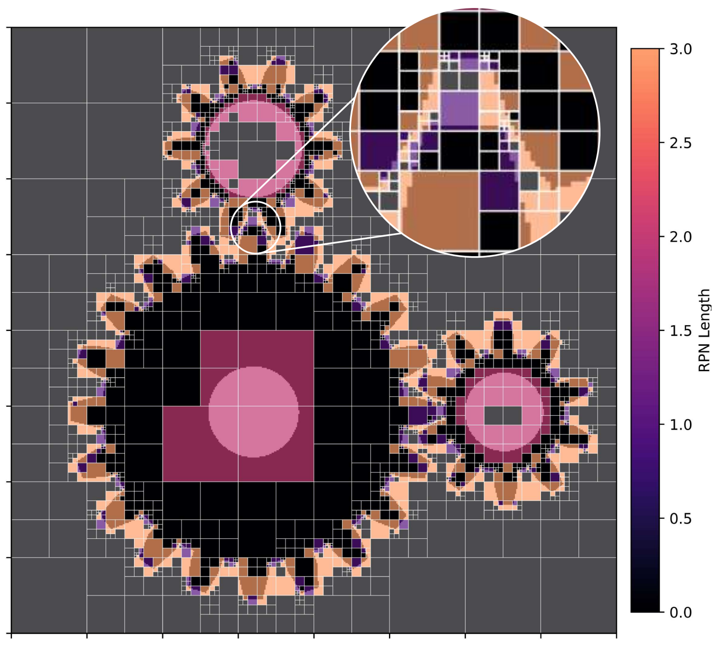


Figure 6.6: Sparse Voxel Octree for the Gears test scene. The inset shows the fine-grained pruning along the gear tooth boundary, and the color bar encodes the local RPN length within each leaf node.

6.2 SVO construction

This section investigates the process of constructing Sparse Voxel Octrees (see Section 5.4), focusing on efficiency and adaptability. It explores how different adaptive thresholds impact octree depth, computational complexity, and memory efficiency. Particular attention is given to analyzing the trade-offs between computational costs during construction and resulting performance benefits in storage and evaluation of the adaptive CSG tree.

SVO construction performance is measured across varying thresholds and four benchmark scenes (see Figure 6.2): Gears, Architecture, Many Spheres, and Many Gears. The Gears and Architecture scenes are derived from Keeter's work [30] and reimplemented in a prototype geometric kernel. The Many Spheres and Many Gears scenes have been introduced to benchmark cases with a significant number of CSG nodes, testing performance of the algorithm under extreme conditions. The complexity (RPN length) of the original CSG trees for all scenes is provided in Table 6.3.

Figure 6.6 shows a cross-section of the SVO for the Gears scene with a threshold value of 3. The color of voxels encodes the local RPN length within each leaf node. The constructed SVO adapts to geometric object boundaries, maintaining shallow depths in empty spaces (both interior and exterior to objects) and achieving a maximum depth of 10 in regions of high primitive density. Notably, pruning occurs in voxels of the middle gear, where the SVO remains shallow despite proximity to geometric boundaries. This occurs because these voxels contain only one primitive, eliminating the need for further simplification. The memory footprint of the SVO is 1 MB despite an effective resolution of $8^{10} = 1\,073\,741\,824$ voxels, compared to 35 MB for a uniform grid with $80^3 = 512\,000$ voxels.

Tables 6.4–6.7 present SVO construction results for all test cases using the RevAA range evaluation technique with maximum tree depth fixed at 15. The data include average SVO depth and SVO construction time across varying threshold values. Reducing the threshold increases both average tree depth and construction time. However, for thresholds ≤ 20 in the Architecture and

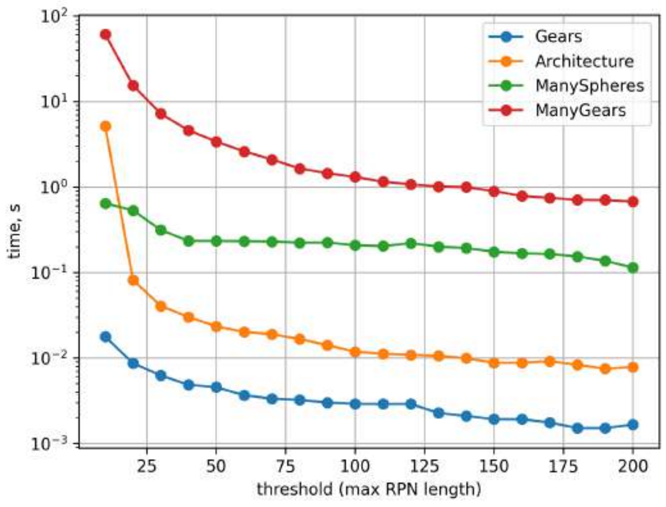


Figure 6.7: Sparse Voxel Octree construction time over various thresholds.

Many Gears scenes, the CSG tree cannot be sufficiently pruned due to reaching the maximum depth limit. Consequently, some voxels contain CSG nodes exceeding the threshold, though this has negligible impact on spatially-average evaluation complexity. Figure 6.7 compares SVO construction times across all test scenes at varying thresholds. Construction time increases with scene complexity, demonstrating a clear correlation between geometric intricacy and computational requirements.

6.3 Evaluation of adaptive CSG tree

This section examines the evaluation performance of the proposed adaptive CSG tree representation, emphasizing computational speed in handling complex geometrical structures. It assesses how effective pruning and adaptive spatial strategies contribute to maintaining constant evaluation complexity despite increased model complexity.

Evaluation of the adaptive CSG tree is performed after the SVO construction. Performance is measured by averaging adaptive CSG tree evaluation times at random points within the spatial domain. Tables 6.4–6.7 present evaluation times for all test cases. The average traversal (avg trav.) time represents SVO leaf node retrieval, the average evaluation (avg eval.) time corresponds

Table 6.4: CSG tree evaluation and rendering benchmark for the Gears scene over various thresholds.

Thresh.	Avg depth	Max RPN len.	Avg RPN len.	SVO constr. time (μs)	Avg trav. time (ns)	Avg eval. time (ns)	Avg total time (ns)	Avg render. time (μs)
200	1.83	176	56.41	1655	69.87	1339.76	1409.63	2336.11
180	1.83	176	56.41	1509	64.83	1334.30	1399.13	2432.47
160	1.93	146	38.40	1917	68.23	837.25	905.48	1446.56
140	1.96	135	32.82	2096	63.45	617.16	680.61	919.37
120	2.45	100	22.07	2893	64.95	510.43	575.38	560.03
100	2.45	100	22.07	2897	65.49	509.64	575.14	560.55
80	2.45	78	18.47	3224	62.67	264.70	327.37	365.39
60	2.67	60	15.32	3663	64.63	218.98	283.61	262.92
40	3.18	40	10.90	4848	64.53	147.29	211.82	133.71
20	4.12	20	5.37	8674	65.31	64.50	129.81	30.91
10	5.28	10	2.92	17706	66.42	45.46	111.88	12.72

Table 6.5: CSG tree evaluation and rendering benchmark for the Architecture scene over various thresholds.

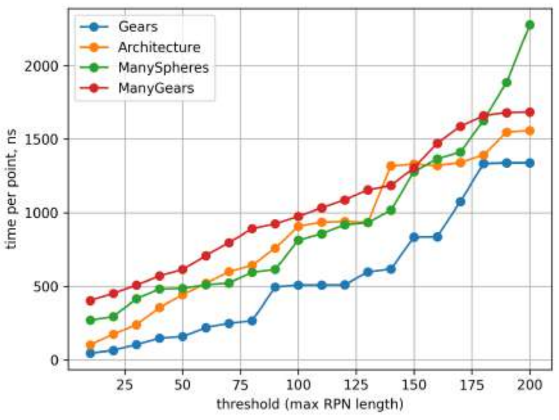
Thresh.	Avg depth	Max RPN len.	Avg RPN len.	SVO constr. time (μs)	Avg trav. time (ns)	Avg eval. time (ns)	Avg total time (ns)	Avg render. time (μs)
200	2.58	187	74.55	7836	75.68	1558.57	1634.25	4652.44
180	2.70	171	62.52	8314	74.55	1392.61	1467.16	3488.06
160	2.74	147	59.86	8754	73.53	1321.72	1395.24	3118.12
140	3.05	138	48.75	9897	74.59	1319.26	1393.85	2867.49
120	3.07	118	44.47	10861	72.88	940.08	1012.96	2151.27
100	3.28	99	37.99	11801	71.34	908.14	979.48	2032.45
80	3.54	80	23.79	16628	74.87	643.22	718.09	748.27
60	3.78	60	19.50	20122	74.64	519.93	594.58	448.40
40	4.39	40	12.20	30135	74.60	355.95	430.56	155.21
20	6.70	23	5.49	81770	77.32	174.74	252.06	43.17
10	13.87	23	3.56	5129835	82.51	102.11	184.62	17.70

Table 6.6: CSG tree evaluation and rendering benchmark for the Many Spheres scene over various thresholds.

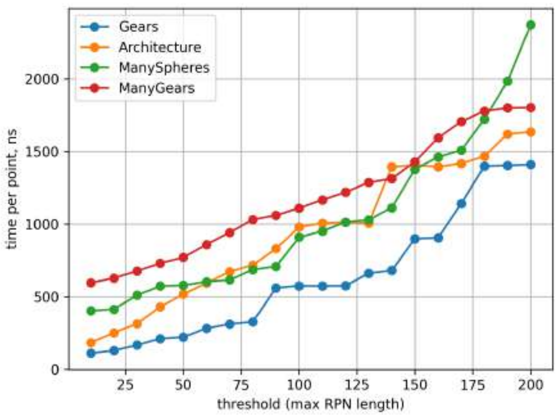
Thresh.	Avg depth	Max RPN len.	Avg RPN len.	SVO constr. time (μs)	Avg trav. time (ns)	Avg eval. time (ns)	Avg total time (ns)	Avg render. time (μs)
200	3.00	199	136.90	114166	93.57	2277.76	2371.32	20713.10
180	3.74	177	48.35	153825	94.72	1626.63	1721.35	845.75
160	3.83	159	39.08	167120	96.09	1365.68	1461.77	627.96
140	3.91	137	31.09	192536	95.60	1017.30	1112.91	388.71
120	3.93	115	29.65	219632	96.76	917.43	1014.19	353.27
100	3.94	99	27.90	207704	95.28	813.91	909.19	299.19
80	3.98	79	23.52	222234	92.33	594.70	687.02	255.35
60	3.99	59	22.26	231738	92.41	511.22	603.62	250.41
40	4.00	39	21.62	233590	91.68	481.57	573.25	250.54
20	4.94	19	4.26	532425	119.67	293.85	413.52	197.04
10	5.05	9	3.28	641786	134.01	268.66	402.67	47.37

Table 6.7: CSG tree evaluation and rendering benchmark for Many Gears scene over various thresholds.

Thresh.	Avg depth	Max RPN len.	Avg RPN len.	SVO constr. time (μs)	Avg trav. time (ns)	Avg eval. time (ns)	Avg total time (ns)	Avg render. time (μs)
200	4.00	197	65.63	671513	117.73	1684.22	1801.95	1513.76
180	4.13	177	59.20	706011	117.67	1660.25	1777.92	1427.13
160	4.24	160	49.93	783475	121.61	1472.17	1593.78	1333.27
140	4.51	140	35.15	991223	127.78	1185.32	1313.10	1002.81
120	4.58	120	31.36	1070470	131.49	1087.22	1218.71	986.72
100	4.74	100	25.87	1305762	134.19	975.93	1110.12	912.69
80	5.05	80	19.71	1639748	140.65	891.01	1031.66	753.81
60	5.42	60	13.03	2603849	150.53	709.00	859.53	498.23
40	5.95	40	8.78	4585594	160.56	570.46	731.01	157.96
20	7.11	20	4.20	15410538	177.25	451.51	628.77	45.66
10	9.21	11	2.30	60767306	192.11	403.33	595.44	27.95

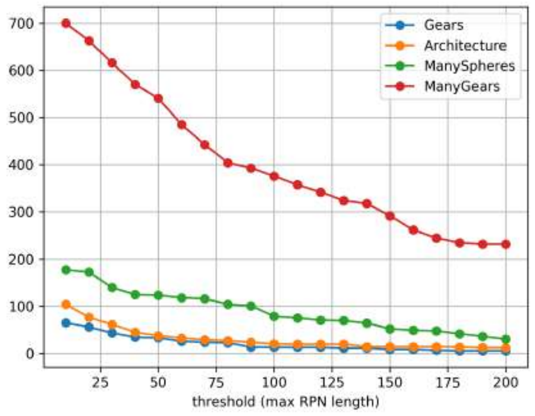


(a) No traversal

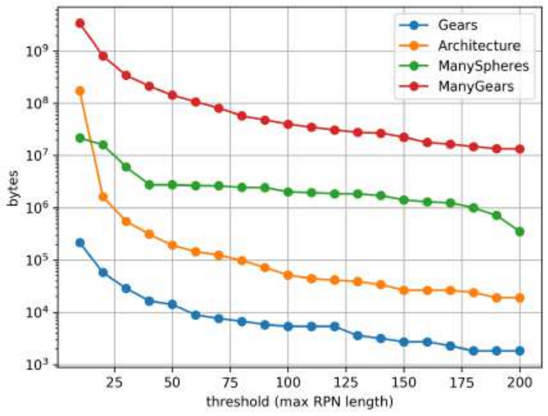


(b) Traversal included

Figure 6.8: Average evaluation of pruned CSG tree for all test scenes over various thresholds.



(a)



(b)

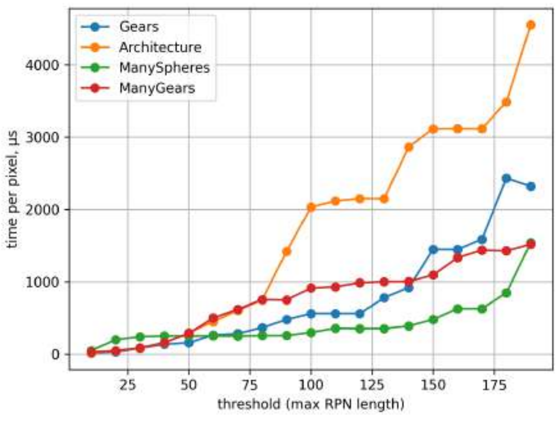
Figure 6.9: Average speedup of CSG tree evaluation (a) and SVO memory consumption (b) over various thresholds.

to pruned CSG tree computation, and the total time is their sum. Figure 6.8 compares pruned CSG tree evaluation times with and without SVO traversal overhead. The data show that traversal time becomes negligible for thresholds $\gtrsim 100$, but dominates evaluation time as thresholds approach zero.

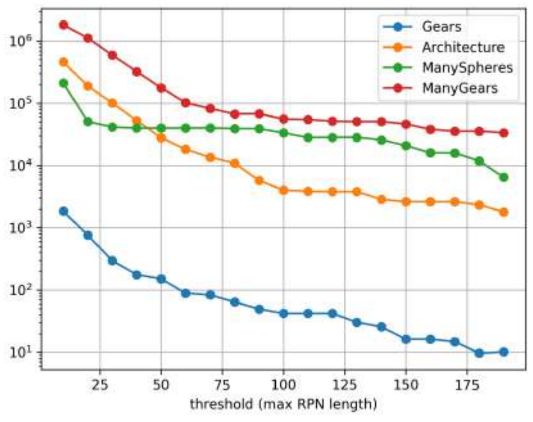
The implemented CSG framework achieves massive speedup proportional to scene complexity (see Figure 6.9(a)), measured relative to original CSG evaluation times (see Table 6.3). The Many Gears scene exhibits the largest gains, with speedups ranging from 200× to 700×, while even the simplest Gears scene demonstrates significant improvements of 5× to 60×.

However, the massive speedup comes at the cost of memory overhead from constructing large Sparse Voxel Octrees. Figure 6.9(b) compares SVO memory consumption across all test cases at varying threshold values. For the Many Gears scene, the maximum speedup requires approximately 3.5 GB to store pruned CSG trees. In contrast, achieving a 200× speedup for the same scene consumes only 10 MB of memory. This trade-off demonstrates the flexibility of the proposed pruning algorithm, which balances performance and memory efficiency for diverse environments—even those with stringent memory constraints.

Furthermore, while SVO construction incurs upfront computational costs, it remains viable



(a)



(b)

Figure 6.10: Average rendering performance of CSG tree (a) and its speedup (b) over various thresholds.

for applications requiring repeated evaluations. For the Many Gears scene with a threshold of 200, SVO construction becomes advantageous when the number of CSG tree evaluations exceeds approximately 1630.

6.4 Rendering

This section assesses the rendering capabilities of the adaptive CSG framework. It discusses the effectiveness of the hybrid ray-casting method enhanced by interval-based root-finding techniques (see Section 5.6), evaluating its performance in accurately visualizing complex implicitly defined objects. Average rendering time per pixel is measured for each scene in Figure 6.2.. Rendering times for all scenes are documented in Tables 6.4–6.7. Figure 6.10 compares rendering times and speedup factors between pruned and original CSG trees across all scenes, where speedup is measured relative to the original CSG tree's rendering times (see Table 6.3). The data reveal that while original CSG tree rendering times scale with geometric complexity, adaptive CSG rendering times depend critically on the threshold parameter. At higher thresholds, rendering times vary significantly across scenes, but as thresholds approach zero, rendering times converge to nearly identical values regardless of scene complexity. This threshold-driven behavior enables speedups ranging from $10\times$ to $10^6\times$ compared to original CSG implementations.

The observed speedup arises from two key factors: **pruned CSG trees** and **spatial adaptivity**. First, in the unpruned case, the original CSG tree must be fully evaluated during root-finding, whereas the adaptive approach substitutes this with a pruned CSG tree (see Section 6.3), drastically reducing computational load. Second, with no pruning, root-finding begins across the entire ray-modeling domain intersection interval, requiring recursive subdivision until interval analysis confirms root absence. By contrast, the adaptive approach restricts root-finding to non-empty voxels—guaranteed subdomains smaller than the full modeling volume. For sparse scenes, this eliminates root-finding for rays intersecting only empty voxels, limiting overhead to voxel traversal.

As the threshold approaches zero, the proposed approach achieves rendering times of tens of microseconds per pixel, enabling real-time modeling and rendering - a capability unattainable with the original methodology. Remarkably, these results were obtained using the classical bisection algorithm, which is among the most robust yet computationally intensive root-finding methods in the literature. Implementing faster root-finding algorithms (e.g., Newton-Raphson or interval-guided methods) could further accelerate rendering performance.

6.5 Comparison with state-of-the-art approaches

The proposed methodology advances state-of-the-art approaches for constructing sparse volume data structures. To the author's knowledge, all existing spatial sampling algorithms for CSG trees [20], SDFs [30, 61], and level sets [42] rely on a common principle: recursive subdivision continues until either an empty voxel is detected or a predefined maximum depth is reached. The proposed approach introduces a novel adaptive stopping criterion that terminates subdivision earlier than the maximum depth limit. By incorporating a threshold value, the method enables flexible control over geometric complexity, halting subdivision in regions where further refinement is unnecessary. This represents a fundamental departure from prior work, with the added advantage that the proposed framework can replicate existing methods by setting the threshold to zero. In this degenerate case, recursive subdivision persists in all non-empty voxels until reaching the maximum depth, mirroring conventional implementations.

Figure 6.11 provides a comprehensive comparison of the proposed methodology against state-of-the-art approaches across all test scenes. Results are generated by systematically varying the threshold and maximum SVO depth values, constrained only by available RAM capacity. The left, middle, and right columns respectively illustrate the speedup in CSG tree evaluation time (proposed vs. state-of-the-art), speedup in rendering time, and memory consumption. Speedup values were computed via element-wise division of rows (fixed threshold, varying maximum depths) by the first row (zero threshold, varying maximum depths), quantifying the proposed method's improvement relative to state-of-the-art baselines at identical maximum SVO depths.

For small maximum SVO depths, speedup is negligible because when the maximum depth approaches zero, the threshold-based stopping criterion in the proposed SVO framework cannot activate, resulting in identical performance to state-of-the-art methods. For large thresholds with variable maximum depths, the proposed approach reduces memory consumption at the cost of slower performance—a trade-off suitable for memory-constrained environments requiring moderate performance gains. Conversely, with small thresholds and large maximum depths, the proposed method matches or exceeds state-of-the-art performance.

In the Architecture and Many Gears scenes, optimal thresholds for adaptive CSG tree evaluation and rendering performance fall within the 2–10 range. As thresholds approach zero, average SVO depth increases substantially. Here, near-complete CSG tree pruning halts further geometric simplification, forcing deeper SVO traversal and slower evaluation of CSG tree. The Many Spheres scene highlights the methodology's potential: evaluation speeds improve by up to 1.5 times, rendering accelerates by 2 times, and memory usage decreases by 194 times compared to state-of-the-art baselines.

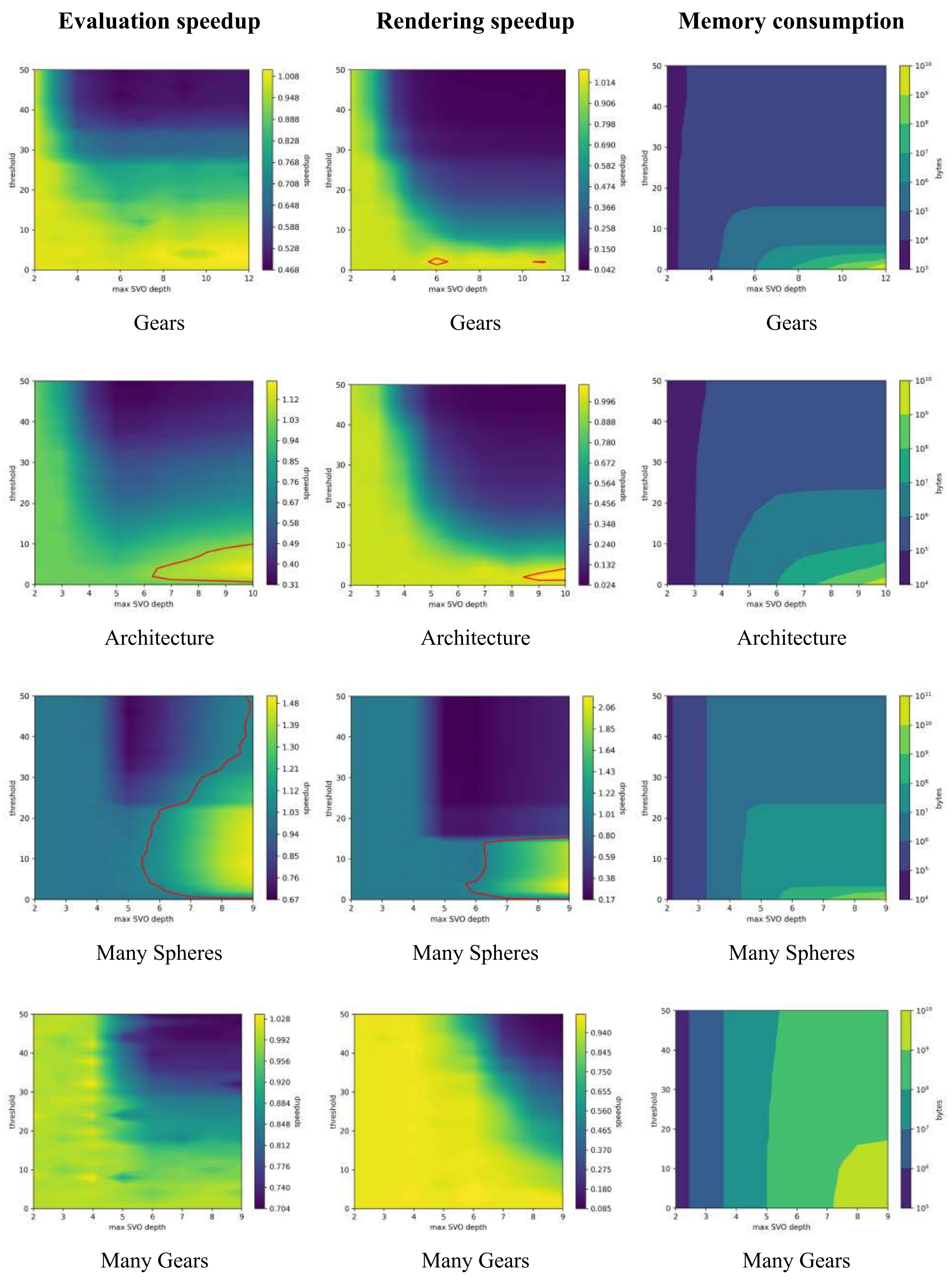


Figure 6.11: Comparison of the proposed and state-of-the-art approaches for all scenes in terms of CSG tree evaluation and rendering performance, and memory consumption. Red levels enclose areas, where speedup ≥ 1.05.

Chapter 7
Future work

This section outlines various directions for future research, highlighting potential performance enhancements and extended modeling capabilities of the proposed mathematical framework.

7.1 Traversal of sparse volume data structures

The proposed framework leverages an SVO data structure to store pruned versions of CSG trees. The octree can become excessively deep in regions, where multiple primitives intersect, increasing traversal time. Replacing the octree with alternative spatial data structures - such as the VDB tree [42], which exhibits lower average tree depth - would reduce traversal time and improve both CSG tree evaluation and rendering performance.

7.2 Storage of pruned CSG tree variants

The proposed framework uses bitmasks to store pruned variants of CSG trees. However, the size of each bit mask scales linearly with the number of CSG nodes, leading to high memory consumption for complex trees with thousands of nodes. Deallocating memory for bitmasks in SVO leaves lacking geometry boundaries would reduce overall memory usage.

7.3 Pruning performance

The proposed pruning algorithm uses iterative bottom-up traversal of the CSG tree, which is converted to an RPN sequence via the Shunting-yard algorithm and stored contiguously in memory. Consequently, pruning occurs only when Boolean operations are encountered during traversal. Performance could be enhanced by pruning Boolean operations earlier—for example, during a union operation, if the first operand is positive throughout a region, evaluation of the second operand becomes redundant. Pruning immediately after evaluating the first operand would reduce unnecessary function evaluations.

7.4 Extension of modeling operations

The pruning algorithm currently supports only Boolean operations in CSG trees. Extending it to CAD-specific operations (e.g., blending) would significantly enhance the framework's modeling capabilities and enable the creation of more intricate objects.

7.5 Root-finding algorithm

The ray-casting algorithm employs a bisection method augmented with interval evaluation for root-finding. Substituting this with higher-convergence-rate algorithms (e.g., the secant method) would improve ray-casting performance.

7.6 Parallelization

In the current C++ implementation, both SVO construction and rendering algorithms utilize only a single CPU core. Parallelization is straightforward: replacing a single SVO with a forest of SVO structures allows easy distribution across threads with no inter-task dependencies. Additionally, rendering individual screen pixels is embarrassingly parallel. Leveraging CPU or GPU multithreading would accelerate both pruning and rendering procedures.

Chapter 8
Conclusion

This thesis presents a mathematical foundation for an adaptive Constructive Solid Geometry (CSG) framework and introduces algorithms for spatial sampling and pruning. These algorithms have been implemented in a prototype geometry-rendering kernel developed entirely in C++. The pruning algorithm leveraging interval analysis techniques has been implemented and comprehensive comparison of Interval Arithmetic (IA), Affine Arithmetic (AA), and Revised Affine Arithmetic (RevAA) has been conducted across diverse geometric scenes. RevAA has emerged as the fastest interval technique. Additionally, the spatial sampling algorithm has been developed for constructing Sparse Voxel Octrees (SVO). Experimental results demonstrate significant speedups in adaptive CSG tree evaluation and rendering compared to original (unpruned) CSG trees.

The proposed adaptive CSG framework generalizes state-of-the-art SVO construction methodologies. By introducing a threshold parameter to terminate SVO subdivision, the framework achieves near-constant evaluation complexity governed solely by the threshold value. While SVO traversal introduces overhead, shallower volume data structures can mitigate this cost. Comparative analysis demonstrates that the proposed approach performs at least as well as state-of-the-art methods in CSG tree evaluation speed, rendering efficiency, and memory usage across multiple test cases.

The current framework implementation exhibits several limitations: only Boolean operations and affine transformations are supported for modeling; pruned CSG tree variants consume a large amount of memory despite bitmask optimization; and geometric modifications require full SVO reconstruction. Addressing these challenges is the subject of future research.

Bibliography

[1] Autodesk Within: Generative design optimized for 3D printing — adsknews.autodesk.com. `https://adsknews.autodesk.com/en/news/autodesk-within-generative-design-optimized-for-3d-printing/`. [Accessed 02-06-2025].

[2] Boost interval arithmetic library — boost.org. `https://www.boost.org/doc/libs/1_88_0/libs/numeric/interval/doc/interval.htm`. [Accessed 26-05-2025].

[3] How implicits succeed where B-reps fail | nTop — ntop.com. `https://www.ntop.com/resources/blog/how-implicits-succeed-where-b-reps-fail/`. [Accessed 25-05-2025].

[4] Indreams — dreams.mediamolecule.com. `http://dreams.mediamolecule.com/`. [Accessed 25-05-2025].

[5] Libaffa - C++ affine arithmetic library for GNU/Linux. `https://www.nongnu.org/libaffa/`. [Accessed 26-05-2025].

[6] Libfive — libfive.com. `https://libfive.com`. [Accessed 25-05-2025].

[7] Shadertoy BETA — shadertoy.com. `https://www.shadertoy.com/`. [Accessed 25-05-2025].

[8] Adzhiev, V., Cartwright, R., Fausett, E., Ossipov, A., Pasko, A., and Savchenko, V. HyperFun project: A framework for collaborative multidimensional F-rep modeling. *Implicit Surfaces '99, Eurographics/ACM SIGGRAPH Workshop* (01 2002).

[9] Aleksandrov, M., Zlatanova, S., and Heslop, D. J. Voxelisation algorithms and data structures: A review. *Sensors 21*, 24 (2021).

[10] Bader, C., Kolb, D., Weaver, J. C., Sharma, S., Hosny, A., Costa, J., and Oxman, N. Making data matter: Voxel printing for the digital fabrication of data across scales and domains. *Science Advances 4*, 5 (2018), eaas8652.

[11] Balsys, R. J., and Suffern, K. G. Visualisation of implicit surfaces. *Computers & Graphics 25*, 1 (2001), 89–107. Shape Blending.

[12] Barbier, W., Sanchez, M., Paris, A., Michel, E., Lambert, T., Boubekeur, T., Paulin, M., and Thonat, T. Lipschitz pruning: Hierarchical simplification of primitive-based SDFs. *Computer Graphics Forum n/a*, n/a, e70057.

[13] Beyer, J., Hadwiger, M., and Pfister, H. State-of-the-art in GPU-based large-scale volume visualization. *Computer Graphics Forum 34*, 8 (2015), 13–37.

[14] Blinn, J. F. A generalization of algebraic surface drawing. *ACM Trans. Graph. 1*, 3 (July 1982), 235–256.

[15] Bloomenthal, J., and Bajaj, C., Eds. *Introduction to Implicit Surfaces*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann, Oxford, England, sep 1997.

[16] Burger, B., Paulovic, O., and Hasan, M. Realtime visualization methods in the demoscene. In *Proceedings of the central european seminar on computer graphics* (2002), pp. 205–218.

[17] Cartwright, R., Adzhiev, V., Pasko, A., Goto, Y., and Kunii, T. Web-based shape modeling with HyperFun. *IEEE Computer Graphics and Applications 25*, 2 (2005), 60–69.

[18] Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, Association for Computing Machinery, p. 15–22.

[19] Dijkstra, E. Algol 60 translation : An Algol 60 translator for the X1 and making a translator for Algol 60. jan 1961.

[20] Duff, T. Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry. *SIGGRAPH Comput. Graph. 26*, 2 (July 1992), 131–138.

[21] Ephtracy. MagicaVoxel — ephtracy.github.io. `https://ephtracy.github.io/index.html?page=magicacsg`. [Accessed 25-05-2025].

[22] Frisken, S. F., Perry, R. N., Rockwood, A. P., and Jones, T. R. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., p. 249–254.

[23] Fryazinov, O., Pasko, A., and Comninos, P. Fast reliable interrogation of procedurally defined implicit surfaces using extended revised affine arithmetic. *Computers & Graphics 34*, 6 (2010), 708–718. Graphics for Serious Games Computer Graphics in Spain: a Selection of Papers from CEIG 2009 Selected Papers from the SIGGRAPH Asia Education Program.

[24] Gomes, A., Voiculescu, I., Jorge, J., Wyvill, B., and Galbraith, C. *Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[25] Grasberger, H., Duprat, J.-L., Wyvill, B., Lalonde, P., and Rossignac, J. Efficient data-parallel tree-traversal for BlobTrees. *Computer-Aided Design 70* (2016), 171–181. SPM 2015.

[26] Hart, J. Sphere Tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12* (06 1995).

[27] Hoetzlein, R. K. GVDB: Raytracing Sparse Voxel Database Structures on the GPU. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics* (2016), U. Assarsson and W. Hunt, Eds., The Eurographics Association.

[28] Jones, M., Baerentzen, J., and Sramek, M. 3D distance fields: A survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics 12*, 4 (2006), 581–599.

[29] Ju, T., Losasso, F., Schaefer, S., and Warren, J. Dual contouring of Hermite data. *ACM Trans. Graph. 21*, 3 (July 2002), 339–346.

[30] Keeter, M. J. Massively parallel rendering of complex closed-form implicit surfaces. *ACM Trans. Graph. 39*, 4 (Aug. 2020).

[31] Kim, D., Lee, M., and Museth, K. NeuralVDB: High-resolution sparse volume representation using hierarchical neural networks. *ACM Trans. Graph. 43*, 2 (Feb. 2024).

[32] Knoll, A., Hijazi, Y., Kensler, A., Schott, M., Hansen, C., and Hagen, H. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. *Computer Graphics Forum* (2009).

[33] Kumar, V., and Dutta, D. An assessment of data formats for layered manufacturing. *Advances in Engineering Software 28*, 3 (1997), 151–164.

[34] Laine, S., and Karras, T. Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), I3D ’10, Association for Computing Machinery, p. 55–63.

[35] Leiserson, C., Prokop, H., and Randall, K. Using de Bruijn sequences to index a 1 in a computer word. *Available on the Internet from http://supertech.csail.mit.edu/papers.html* (02 1970).

[36] Li, W., and Hahn, J. K. Efficient ray casting polygonized isosurface of binary volumes. *The Visual Computer 37*, 12 (Oct. 2021), 3139–3149.

[37] Lorensen, W. E., and Cline, H. E. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1987), SIGGRAPH ’87, Association for Computing Machinery, p. 163–169.

[38] Loterie, D., Delrot, P., and Moser, C. High-resolution tomographic volumetric additive manufacturing. *Nature Communications 11*, 1 (Feb. 2020).

[39] Maltsev, E., Popov, D., Chugunov, S., Pasko, A., and Akhatov, I. An accelerated slicing algorithm for Frep models. *Applied Sciences 11*, 15 (2021).

[40] Mitchell, D. P. Robust ray intersection with interval arithmetic.

[41] Moore, R. E., Kearfott, R. B., and Cloud, M. J. Introduction to interval analysis.

[42] Museth, K. VDB: High-resolution sparse volumes with dynamic topology.

[43] Museth, K. NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks* (New York, NY, USA, 2021), SIGGRAPH ’21, Association for Computing Machinery.

[44] Nishimura, H., Hirai, M., Kawai, T., Kawata, T., Shirakawa, I., and Omura, K. Object modelling by distribution function and a method of image generation. *The Transactions of the Institute of Electronics and Communication Engineers of Japan J68-D* (1985), 718–725.

[45] nTopology Inc. Implicit modeling for engineering design, 2019. `https://www.ntop.com/resources/blog/implicit-modeling-for-mechanical-design` [Accessed: 2025-04-17].

[46] Ogayar-Anguita, C., Garcia-Fernandez, A., Feito-Higueruela, F., and Segura-Sanchez, R. Deferred boundary evaluation of complex CSG models. *Advances in Engineering Software 85* (2015), 51–60.

[47] Osher, S., and Fedkiw, R. *The Level Set Methods and Dynamic Implicit Surfaces*, vol. 57. 05 2004, pp. xiv+273.

[48] Pasko, A., Adzhiev, V., Sourin, A., and Savchenko, V. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer 11*, 8 (Aug 1995), 429–446.

[49] Pasko, A., Fryazinov, O., Vilbrandt, T., Fayolle, P.-A., and Adzhiev, V. Procedural function-based modelling of volumetric microstructures. *Graphical Models 73*, 5 (2011), 165–181.

[50] Pasko, G., Pasko, A., Ikeda, M., and Kunii, T. Bounded blending operations. In *Proceedings SMI. Shape Modeling International 2002* (2002), pp. 95–103.

[51] Popov, D., Maltsev, E., Fryazinov, O., Pasko, A., and Akhatov, I. Efficient contouring of functionally represented objects for additive manufacturing. *Computer-Aided Design 129* (2020), 102917.

[52] Requicha, A., and Voelcker, H. Constructive solid geometry. *Tech. Memo, 25 Production Automation Project* (November 1977).

[53] Ricci, A. A constructive geometry for computer graphics. *The Computer Journal 16*, 2 (01 1973), 157–160.

[54] Rvachev, V. L. Method of R-functions in boundary-value problems. *Soviet Applied Mechanics 11*, 4 (Apr. 1975), 345–354.

[55] Shirley, P., Black, T. D., and Hollasch, S. Ray tracing in one weekend, August 2024. `https://raytracing.github.io/books/RayTracingInOneWeekend.html`.

[56] Shusteff, M., Browar, A. E. M., Kelly, B. E., Henriksson, J., Weisgraber, T. H., Panas, R. M., Fang, N. X., and Spadaccini, C. M. One-step volumetric additive manufacturing of complex polymer structures. *Science Advances 3*, 12 (2017), eaao5496.

[57] Snyder, J. M. Interval analysis for computer graphics. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1992), SIGGRAPH '92, Association for Computing Machinery, p. 121–130.

[58] Sourin, A., and Pasko, A. Function representation for sweeping by a moving solid. *IEEE Transactions on Visualization and Computer Graphics 2*, 1 (1996), 11–18.

[59] Stolfi, J., and de Figueiredo, L. H. An introduction to affine arithmetic. *Trends in Applied and Computational Mathematics 4* (2003), 297–312.

[60] Tilove, R. B. A null-object detection algorithm for constructive solid geometry. *Commun. ACM 27*, 7 (July 1984), 684–694.

[61] Uchytil, C., and Storti, D. A function-based approach to interactive high-precision volumetric design and fabrication. *ACM Trans. Graph. 43*, 1 (Sept. 2023).

[62] Vu, X.-H., Sam-Haroud, D., and Faltings, B. Enhancing numerical constraint propagation using multiple inclusion representations. *Annals of Mathematics and Artificial Intelligence 55* (2009), 295–354.

[63] Winchenbach, R., Möller, M., and Kolb, A. Lipschitz-agnostic, efficient and accurate rendering of implicit surfaces. *The Visual Computer* (Jan. 2024).

[64] Wyvill, B., Guy, A., and Galin, E. Extending the CSG tree. warping, blending and Boolean operations in an implicit surface modeling system. *Computer Graphics Forum 18*, 2 (1999), 149–158.

[65] Zanni, C. Synchronized tracing of primitive-based implicit volumes. *ACM Trans. Graph. 44*, 1 (Nov. 2024).