A Function-Based Approach to High-Precision Volumetric Design and Fabrication

Christopher Uchytil

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2023

Reading Committee:

Duane Storti, Chair

Mark Ganter

Zach Tatlock

Program Authorized to Offer Degree:

Mechanical Engineering

©Copyright 2023
Christopher Uchytil

University of Washington

Abstract

A Function-Based Approach to High-Precision Volumetric Design and Fabrication

Christopher Uchytil

Chair of the Supervisory Committee:
Duane Storti
Department of Mechanical Engineering

This work addresses the challenges of representing, designing, and interacting with high resolution volumetric models for computer-aided design (CAD) applications by presenting a novel function-based representation (F-rep) geometric modeling kernel aimed at providing the capabilities to efficiently design, visualize, and interrogate large scale volumetric models. F-Reps provide a model basis which allows for unambiguous and highly parallelizable point membership classification (PMC) [8, 63, 42] while still supporting modeling techniques common to traditional boundary representation (B-Rep) geometry kernels. Our geometric modeling kernel framework consists of two components: a novel interpreter used to evaluate user-defined models and a sparse volume data structure which stores output produced by the interpreter for visualization and manipulation. Together these two components, in conjunction with the graphics processing unit (GPU), provide the means to support large-scale volumetric modeling applications at scales and speeds not currently achievable by existing software tools. Chapter 2 discusses the geometry kernel interpreter and sparse volume database implementations. Performance metrics are included which demonstrate significant improvements over existing methods in both evaluation and rendering performance as well as supported model size. In the interest of supporting an efficient design process, it is desirable to incorporate into the modeler analysis tools to interrogate properties of the model as it is created. As an initial step toward incorporating analysis tools, Chapter 3 covers a novel

integration technique that can be applied to volumetric models created through the F-Rep geometry kernel to measure integral properties, like center of mass, useful for model validation. The integration technique utilizes Federer's coarea formulation providing a means to integrate functions sampled over a grid when the integrand is not known analytically.

TABLE OF CONTENTS

Page

List of Figures . . . ii

Chapter 1: Introduction . . . 1

Chapter 2: F-Rep Geometry Kernel . . . 4
- 2.1 Background . . . 5
  - 2.1.1 Function Representation . . . 5
  - 2.1.2 Sparse Volume Data Structures . . . 13
  - 2.1.3 Contributions . . . 18
- 2.2 Programmable F-Rep Interpreter . . . 19
  - 2.2.1 Model Representation . . . 19
  - 2.2.2 JIT Operation Grouping . . . 20
  - 2.2.3 Operation Encoding . . . 23
  - 2.2.4 Compilation Pipeline . . . 23
  - 2.2.5 Pruning . . . 24
  - 2.2.6 Evaluation . . . 27
- 2.3 GPU Sparse Voxel Database . . . 32
  - 2.3.1 Data Layout . . . 32
  - 2.3.2 Construction . . . 35
  - 2.3.3 Dynamic Nodes . . . 39
  - 2.3.4 Visualization . . . 42
  - 2.3.5 Slice Decomposition . . . 43
  - 2.3.6 Bulk Properties . . . 44
- 2.4 Results . . . 45
  - 2.4.1 Performance Comparison . . . 45
  - 2.4.2 Model Slicing . . . 51

2.5 Future Work 52
2.5.1 Topology Evaluation 52
2.5.2 Geometry Evaluation 52
2.5.3 Interval Bounds 53
2.6 Discussion 53

Chapter 3: Coarea Grid-Based Volume Integrals 54
3.1 Background 55
3.1.1 Integration Techniques 55
3.1.2 Contributions 56
3.2 Volume Integral Formulation 56
3.3 Discretization 59
3.4 Implementation 61
3.5 Results 64
3.6 Discussion 68

LIST OF FIGURES

Figure Number Page

2.1 A defining function for a unit disk depicted as a function composition tree where each node in the tree represents an algebraic operation. x and y represent the x and y components of a given point in space for which the function is evaluated at. 20

2.2 A function composition tree representing the unit disk in which the algebraic operations have been merged into a single node. Instead of defining the disk using several functions as can be seen in Figure 2.1, JIT compilation allows for the defining function to be represented using one function which combines all algebraic operations together. This reduces the number of functions in the composition tree enabling faster evaluation by the interpreter. 21

2.3 A rendering of an F-Rep model with shading based on a color-coding of pruned function composition trees. Each color represents a unique pruned tree configuration assigned to a leaf node of the sparse volume structure used to store the model. There are a total of three pruned tree permutations: f_{sphere} pruned (red), f_{block} pruned (blue), and no pruning (green). f_{block} and f_{sphere} each represent an operation grouping containing the defining functions for the block and sphere respectively, while f_{min} defines the union operation via the min function. 27

2.4 A visual representation of the data and topology layout of our N^3-tree sparse data structure. Each depth level of the tree has an associated pool group which stores information on the tree. The pool group is divided into individual pools each storing a grouping of data, or "property", for all nodes at a given depth level. Every pool can be accessed via an index to obtain a node-specific copy of a property. The root depth of the tree consists of a single node while the internal and leaf depths can contain multiple nodes. In addition to the pool groups which describes the topology layout, a geometry pool group containing voxel bricks storing output of the interpreter is allocated alongside the leaf depth. The geometry pool group is divided into channels each of which describes a single value at a given voxel location. Leaf nodes reserve voxel bricks from the geometry pool group to store model information generated by the interpreter corresponding to the node's location in space. 33

2.5 Each data structure defines a property used to represent our N^3 tree. The value of N within the *Children* property represents the $\log_2$ number of children for a given tree depth level specified in the topology configuration. 37

2.6 A rendering of an F-Rep wood screw designed and visualized using our geometry kernel. The top row shows two views of the wood screw rendered with shading based on a simple diffuse illumination model. The images in the bottom row show the same views of the wood screw but with shading based on a color-coding of the pruned function composition tree. Each colored region on the screw corresponds to a particular pruned tree configuration belonging to an N^3-tree leaf node. The color is generated by converting the bits of a node's pruned tree into a single integer, scaling the integer to the range $[0, 1]$, and mapping that range to the HSV color space. Prune bits which appear earlier in the pruned tree cause larger color differences from pruned bits which appear later in the tree due to occupying more significant bits of the resulting integer value. 41

2.7 The bear head, gear, and architecture model from [42] used to benchmark our geometry kernel. 44

3.1 Level sets of the two-dimensional function $f(x, y) = \left((\frac{x}{5})^2+(\frac{y}{4})^2-1\right)\left((x-1)^2+(y+1)^2-1\right)$. The lightly shaded region shows the domain defined implicitly by $f < 0$. The darker region corresponds to the level-set contribution $\eta = -8$. 57

3.2 Visualization of the region of mixture fraction ratios between 0.3 and 0.4 of f_{DNS} 59

3.3 Histogram of the DNS data over the interval $[0, 0.5]$. Frequency corresponds to the fraction of grid points with whose f_{DNS} values lie in a bin of width 0.005. 62

3.4 The L^2-norm of the residual error in integrating all trivariate polynomials of order up to three ($x^i y^j z^k$ with $i, j, k \geq 0$ and $i + j + k \leq 3$) over a unit sphere. The data for the solid lines is the error for the Characteristic and AW methods (presented in [84]) as a function of octree depth (labeled on the upper horizontal axis). The data for dashed lines is the error for the coarea method averaged over 10 jittered trials with grid size 32^3, 64^3, and 128^3 respectively as a function of level set count (labeled on the lower horizontal axis). 63

Chapter 1

INTRODUCTION

Solid modeling refers to theories and computations that define and manipulate digital representations of physical objects and their properties. The representations and computations used in solid modeling are based on sound mathematical and physical principles, innovative and compact data structures, and efficient and reliable algorithms. Solid modeling supports the creation, exchange, visualization, evolution, analysis, animation, interrogation, annotation, manufacturing, and fabrication of digital models [82]. Solid models are developed through the use of computer-aided design (CAD) systems, which in turn typically utilize a geometric modeling kernel defining the operations and representations available for designing and storing models. Traditional CAD software tools implement geometry kernels which describe models using a boundary-representation (B-Rep) that aims to define a solid as a collection of surface patches properly connected to separate internal regions from the exterior ambient space. New fabrication technologies like additive manufacturing (AM) have pushed the boundaries on what is manufacturable, extending the gamut of parts that can be built beyond what is feasibly represented with existing B-Rep systems. AM technologies provide a mechanism for converting digital models to physical parts through direct control over local material deposition, meaning full utilization of AM systems necessitates a complete description throughout the interior of the model. The information needed for AM fabrication corresponds more directly to a volumetric representation; however, volumetric models encounter practical limitations. Conversion of B-Rep model descriptions into a volumetric form can be challenging due to the computational cost and potential robustness issues associated with B-Rep point membership classification (PMC). Volumetric representations can also be difficult to handle at scales supported by AM technologies, with existing AM systems capable of micron-level features (including internal variations in material or process properties) across meter-sized build volumes [4]. The corresponding voxel models can include hundreds of billions of elements with associated memory requirements running upwards of terabytes. Current modeling software is incapable of designing and representing models at these resolutions, and the issue will only be exacerbated by enhanced capabilities of future AM systems.

This work provides an alternative geometric modeling kernel which utilizes a function-based representation (F-Rep) for model description. The geometry kernel consists of an interpreter used to evaluate user-defined model functions and a sparse volume database which stores output produced by the interpreter. Chapter 2 discusses the geometry kernel interpreter and sparse volume database implementations and includes performance metrics demonstrating improvements in evaluation speeds, rendering performance, and supported model size compared to existing methods. In addition to the geometry kernel, Chapter 3 presents a novel integration technique which can be applied to models generated by the geometry kernel, useful for interrogating properties of the model during the design process.

Chapter 2

F-REP GEOMETRY KERNEL

2.1 Background

2.1.1 Function Representation

A function-representation (F-Rep) defines a solid object as a half-space via the inequality $\{f(r) \leq 0, r = (x, y, z, ...)\}$ where $f(r)$, the "geometric defining function", is a real valued continuous function. A geometric defining function implicitly defines the geometry of an object while a level-set of the function, typically the 0 level-set, specifies the boundary or surface of the solid. This implicit definition gives rise to the alternate terminology "implicit model". The sign of the function value produced when evaluating the defining function also indicates set membership (i.e. inside vs outside the model), allowing for robust point membership classification (PMC) [8, 18, 63]. F-Reps serve as a powerful alternative to B-Reps in representing solid models and have been used in numerous applications; from the modeling of micro-structures [65] to large scale terrain [23].

$$f_{\text{sphere}}(x, y, z, r) = x^2 + y^2 + z^2 - r^2 \tag{2.1}$$

Implicits are conventionally considered difficult to render directly. Traditionally there have been two methods by which an implicit can be visualized. The first involves polygonization of an implicitly defined level-set using algorithms like marching cubes or dual contouring [54, 37, 3] followed by polygon rendering via rasterization. The polygonization approach takes advantage of the extensive software and hardware support for polygonal mesh visualization, however there are also drawbacks involving robustness and accuracy, often leading to geometric features like spikes and sharp edges being lost [53, 20]. Polygonal meshes cannot exactly represent curved surfaces; shading methods can be employed to make the surface mesh appear smooth, but polygonization inevitably introduces geometric artifacts that degrade accuracy of the model. Polygonization is also not guaranteed to produce a mesh corresponding to the boundary of a valid 3D solid. As a result, it is wise to avoid relying on the robustness of PMC based on polygonized models [53, 20]. The second and more common method for F-Rep visualization is ray casting/tracing; models are rendered

by testing for intersections between the region described by the model's geometric defining function and rays cast through space. However, finding the F-Rep surface in a robust fashion while also limiting the computational cost associated with determining an intersection is a challenging problem equivalent to root-finding. The cost incurred by evaluating intersections for each of the hundreds of thousands or even millions of rays has proved to make real-time visualization a difficult task. The inability to render at a real-time rate has been a major contributing factor to the lack of wide-spread adoption of F-Reps as a form of modeling [27]. A sizable portion of research around F-Reps has been focused on reducing the computational cost required for evaluation and visualization. In the following sections we discuss the history of existing techniques aimed at reducing this computational burden.

Analytic Techniques I

Analytic techniques involve computing the ray-implicit intersection by solving the closed form expression of the geometric defining function along each ray. When the function representing the geometry is limited to polynomials of low degree the roots of these polynomial equations can be solved directly and correspond to intersections between the ray and the geometry's surface. Hanrahan [30] , Wijk and Jarke [88], Wyvill and Trotman [90], Nishita et al. [62], and Sherstyuk [80] present some of the earliest work involving applications of analytic techniques for solving ray-implicit intersections. Hanrahan [30] demonstrates the use of Descartes' rule of signs to solve polynomials of degree 5 or higher by subdividing the ray into disjoint segments each containing a single root. The segments can then be traversed to find the intersection point corresponding to the root. Wijk and Jarke [88] focuses on solving intersections for various sweep-defined objects through the use of Sturm sequences. Wyvill and Trotman [90] presents work utilizing Laguerre's method to robustly compute all roots of a polynomial along a ray in order to support model construction through CSG operations; models are represented through a tree composed of primitives. The roots of each primitive are computed independently along a ray and are then iterated over to find the first surface intersection. Nishita et al. [62] proposes an alternative to Laguerre's method, opting

to instead express the geometric defining function along the ray using Bezier functions and to then employ Bezier clipping, which uses the convex hull property of Bezier curves, to accelerate the intersection computations. Finally, Sherstyuk [80] illustrates how the geometric defining function can be approximated via piece-wise quadratic polynomials along the ray to increase intersection computations at the cost of a less accurate solution.

Lipschitz Techniques I

Kalra and Barr [38] presents the first application of a Lipschitz-based ray tracing technique through the use of "LG-surfaces". An LG-surface is defined as an implicit with a bounded and computable first derivative. The L of LG-surface refers to the defining function's Lipschitz constant, the maximum rate of change of the defining function over a region $\mathcal{R}$,

$$\{f(x_1 - f(x_2)) < L||x_1 - x_2||, x_1, x_2 \in \mathcal{R}\} \tag{2.2}$$

while G refers to the maximum rate of change of the defining function along the parameterized ray.

$$\{g(t_a) - g(t_b)) < G||t_a - t_b||, t_a, t_b \in T, \text{where } g(t) = d \cdot \nabla f(d \cdot t + o)\} \tag{2.3}$$

With a bounded and known first derivative a tight fitting spatial partitioning can be constructed which is guaranteed to contain the implicit surface. The portion of the ray contained within the bounding box is then repeatedly subdivided based on the signs of the geometric defining function until the surface is found.

When a solution can not be computed analytically other approaches are required to determine ray-implicit intersections. A common brute force alternative used in several Lipschitz-based techniques is that of ray-marching. A ray intersection is determine first by dividing the ray into segments of a predefined size and then iterating over the segments, evaluating the defining function at each iteration. If a particular segment contains a value from the defining function that is less than the level-set specifying the surface the ray has intersected

the geometry. The difficulty of applying ray-marching is in defining an adequate step size: too large and the marching algorithm might skip over geometry, too small and the visualization may no longer be interactive. Hart introduces the concept of Sphere Tracing, first implemented in [32] and further expanded upon in [33], a robust adaptive-step ray-marching technique guaranteed not to penetrate an implicit surface. Sphere Tracing uses the Lipschitz constant which bounds the rate of change of the defining function to define a dynamic step size which reduces the total number of evaluations and is resilient to over-stepping of the surface. Each point along the ray defines the origin of an "unbounding sphere" with a radius equal to the step-size which reliably does not include a ray-surface intersection. In contrast to LG-surfaces, Sphere Tracing does not require the need to evaluate the derivative of the defining function. The only requirements are that the defining function be continuous and have a bound on the magnitude of the gradient.

Interval Techniques I

Interval analysis (IA) is a technique first introduced by Moore [57, 58] initially developed as a method for tracking error inherent in finite-precision floating point representation of real numbers. IA involves extending functions from operation on real inputs to operate over intervals representing sets of real numbers. A generic interval $\hat{x}$ is defined as:

$$\hat{x} = (\underline{x}, \bar{x}) \text{ where } \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \bar{x}\} \tag{2.4}$$

The bounds of the interval can be exactly represented in floating point and spans a range which contains the desired and possibly unrepresentable value. The interval extension, $\hat{f}$, of the real function f, defines how the function operations on intervals and satisfies the following condition:

$$\{f(x) \in \hat{f}(\hat{x}) \;\forall\; x \in \hat{x}\} \tag{2.5}$$

An interval extension (IE) operates on intervals and returns an interval whose range

contains all possible output values of the real function for any real value in the input interval [57, 83].

Mitchell [55] employ IA as a form of root isolation to aid in computing an analytic solution to determine ray intersections. The ray origin along with a specified end point define an interval which can be recursively bisected until only one root is contained within the interval range. In a follow up to Mitchell [55], Duff [15] demonstrates how IA can be used to construct a hierarchical spatial partitioning in much the same way as the Lipschitz-based spatial partitioning method presented by Kalra and Barr [38]. Further, Duff [15] shows that when constructing an implicit through CSG style operations, within each spatial partition a reduced order representation of the defining function can be constructed which preserves the geometry. By omitting geometric primitives comprising the defining function which do not overlap with a given region of space, the total number of function evaluations required to produce the implicit within that region is reduced. Affine arithmetic (AA) [19], an alternative form of range analysis developed to overcome the dependency problem inherent within IA, is incorporated by De Cusatis et al. [13] into a ray tracing algorithm in an attempt to generate tighter bounds than those observed when using IA.

GPGPU Computing

The early 2000s brought with it the introduction of general purpose GPU (GPGPU) computing model [31], marking a transition in focus of F-Rep based research. While not suitable for all workloads, the GPU provides significant performance gains over the CPU for problems compatible with GPU-based parallelism. Most methods described thus far meet this requirement making the GPU an ideal candidate for future research. There was both a heavy focus on migrating existing techniques to the GPU for new performance gains as well as shift away from the CPU for future research due to its compute limitations.

Analytic Techniques II

With the advent of the GPGPU computing model analytic techniques have fallen slightly out of favor. This is due in large part to the restrictions imposed on the geometric defining function, the often less GPU-friendly algorithms required to compute roots, and because the computational overhead required for alternative methods like ray-marching are much less significant when performed on the GPU [51]. Nonetheless, Loop and Blinn [51], Kanamori et al. [40], and Singh and Narayanan [81] each present analytic approaches which leverage the processing power of the GPU to greatly improve implicit visualization. Loop and Blinn implement a ray tracer targeted at rendering algebraic surfaces approximated by piecewise Bernstein polynomials. Kanamori et al. and Singh and Narayanan each present GPU adaptations of previously discussed work developed for the CPU, the former adapting Nishita et al. [62] and the latter adapting Mitchell [55].

Lipschitz Techniques II

Keinert et al. [43] present a modified sphere tracing technique designed to further reduce the number of steps required to intersect the surface. An over-stepping heuristic is implemented to accelerate the marching process and allows for step sizes that exceed the bounds of traditional sphere tracing. After each step, a check is performed to ensure that unbounding spheres of two consecutive marching steps overlap. In cases where this check fails the sphere tracing implementation falls back to normal sphere tracing. Seyb et al. [77] introduce a variant of sphere tracing, dubbed Non-linear Sphere Tracing (NLST), to support visualization of implicits that have undergone non-linear deformations, a feature useful for model articulation. Instead of applying the non-linear deformation to the geometric defining function, the deformation is applied to the ray such that the marching algorithm steps over the distorted ray in undistorted space. More recently Galin et al. [21] focused on improving sphere tracing through the inclusion of a local Lipschitz bound. Typically the bound used when sphere tracing is a global bound, limiting the overall efficiency when marching through

areas of the geometric defining function which contain smaller local Lipschitz bounds. When accounting for the local Lipschitz bound the total number of steps required to intersect the surface can be reduced significantly.

Interval Techniques II

Sanjuan-Estrada et al. [73], like Mitchell [55], illustrate the use of IA in isolating roots of analytic functions to find solutions to the ray intersection problem on the GPU. Knoll et al. demonstrate first on the CPU [48] and later on the GPU [45], how IA as well as reduced AA, a modified form of AA better suited for the GPU, can be applied to compute intersections of general implicit surfaces.

Foundational Literature: A Closer Look

In addition to difficulties associated with F-Rep rendering, F-Reps take a monolithic approach to modeling; all modifications made to a model's geometry reside in a single expression. As a model grows in complexity, even modifications that apply only to localized regions make the model globally more expensive to evaluate. Duff [15] addresses this issue through the use of IA as discussed in Chapter 2.1.1. However, Duff's approach is not implemented on the GPU and requires that models be defined exclusively through CSG. Here we discuss the work from Keeter [42] which extends the work from Duff in several ways and also provides the primary motivation for our work. We present an in-depth overview to give context to what we discuss in Chapter 2.2.

Keeter builds from ideas described in Duff and uses an IA-based algorithm to reduce the evaluation cost of F-Rep models. Duff's inclusion of IA allows for individual CSG primitives to be removed, or "pruned", within regions of space where the primitives do not contribute to the overall geometry. A pruned defining function contains fewer operations thus reducing the work required in evaluating the model's defining function. Instead of describing a model through a single equation, space is divided into regions each containing a pruned version of the original defining function. However, as Keeter points out, Duff's implementation is not

amenable to GPU-based hardware primarily due to the heterogeneous workloads required in evaluating each spacial region. Keeter presents an efficiently parallelizable alternative, replacing the geometric primitives of the CSG model representation with pure mathematical expressions. The base implementation supports algebraic (+, -, *, /, sqrt), transcendental (sin, cos, asin, acos, atan, exp, log), and piecewise (abs, min, max) functions.

To facilitate Keeter's approach to model representation, Keeter implements a GPU interpreter along with an encoding format which translates individual mathematical operations into a series of commands to be executed by the interpreter. While less efficient than a model-specific program, the inefficiencies resulting from interpreter overhead are often dwarfed by the computational gains resulting from avoidance of operation evaluations which do not contribute to the model's local geometry. In addition, compiling and storing hundreds of thousands of model-specific programs each containing their own unique pruned geometric defining function is computationally intractable.

The encoding format for the geometric defining function is produced from a directed acyclic graph (DAG) which describes the sequence of mathematical operations to be performed. Each operation inside the DAG is represented within the encoding as a "clause". Each clause contains an operation encoding, or "op code", and an accompanying list of slots, which contain the memory being operated on; unary operations are assigned one slot while binary operations are provided two. When grouped together, the clauses form a tape which the interpreter iterates over to compute the defining function. Tape evaluation occurs in two phases:

- Phase 1 receives both a tape as well as an interval region over which the tape should be pruned.
- Phase 2 is fed the pruned tape output of phase 1 along with position values for which the tape should be evaluated

Phase 1 can also be applied recursively over progressively smaller interval ranges to further

reduce the tape provided to the second phase. Pruning of the tape takes place when a minimum or maximum operation is encountered. If the range spanned by each input interval provided to the minimum or maximum function are disjoint, one of the two intervals will never contribute to the final solution. The section of tape which produces the redundant interval can be omitted without modification to the resulting geometry. It should be noted that a pruned tape is only valid (i.e. the underlying geometry remains unchanged) over the input interval region provided during its construction in phase 1. As a result of this two phase pipeline, the thousands of unique GPU programs are replaced with thousands of pruned copies of the original DAG, each applying to its own sub-region.

2.1.2 Sparse Volume Data Structures

Volumetric data structures, most commonly represented through the use of a voxel grid, are used in many application domains. The voxel grid serves as a discretized representation of three-dimensional Euclidean space. Points within voxel space are indexed with integer coordinates. Grid points of a voxel space hold information on a given property contained within the region of space: from binary values indicating occupancy, to floating point values representing a sampling from a continuous function. Voxel grids are typically stored in a dense form, as a block of contiguous memory. While convenient, dense voxel grid memory requirements are frequently too costly. Additionally, it can often be the case that only a fraction of the entire dense volume is needed at any given time. Sparse grids represent an attractive alternative, significantly reducing memory consumption by only allocating memory for data to describe the relevant properties in regions of space within a volume that are needed. Support for sparse grid functionality incurs costs such as a larger overhead in querying individual grid elements. There are a myriad of ways in which a grid can be made sparse, each with its own set of advantages and drawbacks. The severity of a given drawback is domain specific meaning no one-size-fits-all implementation exists.

The most frequent drivers of sparse grid development come from domains which require large scale simulations and/or visualization, both of which also benefit greatly from utiliza-

tion of the GPU. In turn, the GPU's limited and non-expandable video memory is less of a hindrance as a result of the decreased memory requirements brought by sparse grids. In addition to being domain dependent, sparse grid implementations are also data dependent; the types of data contained within a sparse grid play a significant role in how the grid is constructed. Here we present relevant literature and discuss how the data and domain requirements shape the development of GPU-based sparse volume data structures to better understand their strengths and weaknesses in various applications. It should be noted that we choose to focus on sparse volume data structures developed after the introduction of the GPGPU computing model. For a more thorough examination of sparse grid implementations, including CPU-based sparse grids, see [2, 5, 29].

Binary Data

Binary data voxel grids are traditionally what come to mind when thinking of voxels. Each point within the voxel grid stores a binary value indicating if the space spanned by the voxel is occupied. Voxels are often used as an alternative primitive type to triangles when representing surfaces or solid objects. Grids of voxels are frequently constructed from existing objects defined through surface-based representations.

Laine and Karras [49] demonstrate a novel GPU-based approach to the construction of a sparse voxel octree (SVO), a modified octree structure, which functions as a volumetric alternative to mesh-based surface representations. An SVO, like an octree, recursively bisects the spatial domain along each coordinate direction until a maximum depth level is reached. Because this subdivision process produces 8 child regions at each bisection, the sparse tree structure is referred to as an octree. Early termination occurs if the geometry contained within the region of space fully occupies the region. Leaf nodes of the SVO correspond directly to voxels in the grid while nodes at higher depth levels equate to voxels which span a larger region of the voxel grid. The sparse grid space saving measures reduce the total memory footprint of the voxel grid at the cost of increased voxel access time.

When approximating smooth geometry with a voxel grid, significant error can be intro-

duced. To deal with such errors, Laine and Karras incorporate triangle data of the existing geometry into the SVO. Each occupied grid point of the SVO also stores information about the triangle(s) which intersect the voxel. When rendering each voxel, the normal used for rendering is based off of the normal of the triangle instead of the normal of the voxel allowing for finer detail without the need for further voxel grid refinement.

Kampe et al. [39] improve on the memory limitations imposed by the SVO by transforming the tree into a directed acyclic graph (DAG) to produce a new sparse voxel structure dubbed Sparse Voxel DAG (SVDAG). SVDAG saves on memory by merging identical subtrees contained within the SVO. Kampe et al. present several example SVDAGs that provide significant memory reduction over their SVO equivalent, with one example providing a memory reduction of 25x. In addition to a reduced memory, footprint traversal time is no worse than an SVO and, in some cases, may even improve as a result of many branches sharing the same memory address. Dado et al. [11] and Dolonius et al. [14] both improve on the initial implementation of SVDAG by incorporating color information into the sparse tree, something the original implementation was not capable of.

Further refinements to the SVDAG where proposed by Villanueva et al. [87] in which greater branch merging could be achieved. By storing nodes in a default orientation along with a transformation, rotationally symmetric subtrees can be merged together, further reducing the overall sparse volume structure.

Scalar Data

While binary data sparse voxel structures provide the largest compression rates, treating all data within a grid as binary is not always possible. Take for example a grid containing scalar values representing fluid data from a fluid simulation. The voxel values are associated with physical properties needed to compute the simulation; simply knowing which voxels have fluid in them is not sufficient. The techniques used for binary data are not all applicable to scalar data so alternative voxel structures are needed so that scalar data can be made sparse.

Hadwiger et al. [28] present a scalar valued voxel space subdivision strategy aimed at real-

time visualization of implicit surfaces sampled on a regular grid. The volume is subdivided into two grid levels: a course grid which partitions the primary volume into smaller sub-volumes, and a fine grid to facilitate empty space skipping. The sub-volumes which make up the course grid are referred to as bricks and are designed to circumvent memory limitations of graphics hardware. Individual bricks are designed to fit onto the GPU and can be streamed when needed allowing for volumes exceeding video memory to be rendered. The fine grid is composed of blocks each of which store the minimum and maximum function values of all voxels contained within. When rendering a given level-set of the implicit function, if not contained within the range spanned by the minimum and maximum function values, the block can be skipped. The memory limitations for a given volume no longer equal the total volume, but instead the number of bricks containing the level-set being rendered.

Ruijters and Vilanova [71] detail a similar subdivision strategy. Like, [28], the first grid level involves splitting the primary volume into bricks which fit onto the GPU. Where the two methods differ is in the subdivision strategy of the subsequent layer. Instead of decomposing each brick into blocks, each brick is assigned an octree containing information on voxel occupancy determined by the region's visibility. This octree is then used as an acceleration structure during the rendering process.

In an effort to eliminate volume bricking Knoll el al. [47, 46] devise an SVO-like octree structure used not only as a ray tracing accelerator, but also to hierarchical compress scalar volumes. The octree constructions is accomplished via a bottom up method in which neighboring voxels with matching values are recursively merged together. While memory reductions as high as 10x were reported, compression rates are tied heavily to data variance. Because the octree must fit completely onto the GPU this method is not suitable for all scalar volumes, especially those with large variance.

A fully adaptive method is presented by Gobbetti et al. [25] which further reduces the memory requirements for interactive visualization of large volumes. To achieve this Gobbetti et al. take advantage of the fact that only parts of the total volume are visible at any given time and it is only these visible regions that need to be moved to the GPU. A fixed size

working set of uniform bricks, or "brick pool", is allocated along with a view-dependent octree which is updated on each frame based on region visibility. During visualization of the grid, all visible octree nodes are assigned a brick and the necessary data is dynamically streamed to the GPU. In contrast to the SVO, whose nodes hold individual values, the octree leaves of this structure store groups of values stored in bricks. Octree visibility is assessed not only on occlusion but also on distance. Traversal of the octree is stopped early when far away and brick assignment is extended to internal bricks. When assigned a brick, internal nodes store a lower resolution sub-sampled regions reducing the total streaming overhead. This is especially useful when rendering volumes from far away where a full resolution brick isn't necessary.

Crassin et al. [10] produce work developed independently of [25] but which shares many similarities while also providing improvements in key areas. Instead of constructing an octree, Crassin et al. opt to instead implement an N^3-tree; a tree where each dimension is subdivided into N uniform children. (In the case of $N = 2$, this results in a standard octree). N^3-trees allow for a trade-off between traversal efficiency and memory efficiency by adjusting both the depth and breadth of structure. In the case of homogeneous regions, tree nodes can also be assigned constant values eliminating unnecessary brick streaming and increasing brick availability. Lastly, the node brick pointer overhead is greatly reduced decreasing the total tree memory overhead.

Hoetzlein [35] and Gao et al. [22] extend support of existing CPU sparse grids to the GPU. Gao et al. focus on adaption of the Sparse Page Grid (SPGrid), a structure proposed by Setaluri et al. [76] for large scale fluid simulations. SPGrid, and the GPU equivalent GSPGrid, forgo a tree-based adaptive scheme in favor of a pyramid of sparsely populated uniform grids designed to accelerate sequential and neighbor data accesses, operations commonly performed within a simulation. Hoetzlein introduces GVDB, a GPU-based sparse grid structure inspired by Museth's OpenVDB [61]. GVDB aims to provide functionality mirroring that of OpenVDB through support of efficient stencil operations and dynamic changes to topology. In an approach similar to Crassin et al., GVDB pairs an N^3-tree with a voxel atlas

comprised of bricks assigned to tree nodes. A novel memory pooling architecture is used to store the tree topology resulting in major performance improvements over existing methods. Wu et al. [89] further iterate on GVDB providing greater support for dynamic topology updates as well as develop a GPU optimized matrix-free conjugate gradient solver for fluid simulations. More recently Museth provided a GPU extension of OpenVDB in the form of NanoVDB [60]; a linearized, read-only representation of the OpenVDB structure. While limited in use outside of direct volume rendering due to its read-only nature, NanoVDB provides impressive results in render-based workloads. Kim et al. [44] also extend OpenVDB with NeuralVDB by incorporating neural networks to learn the sparse grid topology and volumetric data contained within. The trained networks greatly reduce memory needed to store the sparse grid, but the requirement for both training and decoding steps provide a significant challenge for implementation of real-time applications.

2.1.3 Contributions

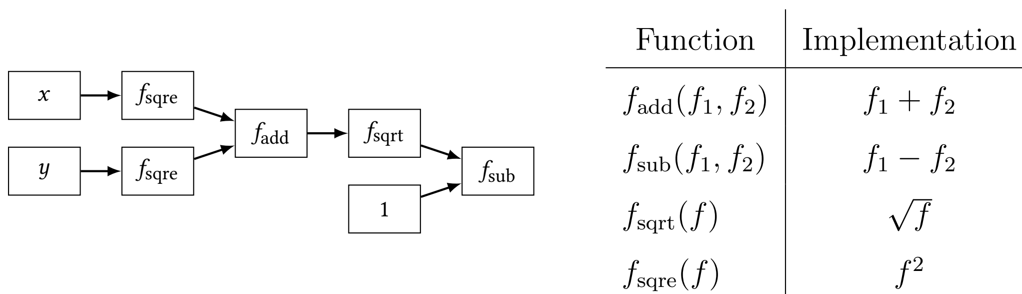
Our geometry kernel design addresses issues associated with current F-Rep geometry kernels and their applications to CAD. While existing kernels, like the one presented by Keeter [42], can achieve impressive rendering speeds, they do not provide a means of efficiently accessing data produced by an F-Rep model's defining function outside of a rendering context. The data derived from an F-Rep model is treated as ephemeral, existing only for the length of a render frame. Interacting with a model in any context outside of rendering is typically implemented by first tessellating the model's surface. This approach not only makes common CAD operations like computing a model's bulk properties reliant on the robustness and accuracy of the tessellation process, it also results in the discarding of all volumetric information obtained by querying the F-Rep. To overcome such limits on access to data for the underlying F-Rep, our geometry kernel is designed to both improve performance over existing F-Rep geometry kernels and to address the challenges of managing large scale volumetric data produced by F-Rep evaluation, avoiding the need for surface tessellation. Contributions include:

- Just-in-time (JIT) compilation of user-created model defining functions, improving interpreter performance.
- Enhanced pruning capabilities which extend prunability to the arithmetic operations, allowing for additional defining function optimization.
- Reduced memory requirements for storing optimized copies of a model's defining function.
- Integration of a sparse volume data structure to efficiently manage and store data produced by the interpreter.
- A demonstration of how the sparse volume data structure can be utilized to efficiently perform CAD operations like slice decomposition and bulk property evaluation without the need for surface tessellation.
- A thorough evaluation of rendering performance, demonstrating improvements in framerate and supported model grid size compared to existing methods.

2.2 Programmable F-Rep Interpreter

2.2.1 Model Representation

Within modeling software, the geometry kernel defines the allowable set of operations which can be used to describe a model and forms the foundation from which a model is constructed. Our approach is to implement a geometry kernel which uses a general-purpose interpreter on the GPU to handle model evaluation. Models are defined through function composition; individual functions representing mathematical operations are exposed to the user and can be composed form a hierarchical graph structure, or "function composition tree", describing a model's geometric defining function. The mathematical operations behave as an assembly language for authoring models. This approach is similar in principle to Keeter, but deviates



Function	Implementation
$f_{\text{add}}(f_1, f_2)$	$f_1 + f_2$
$f_{\text{sub}}(f_1, f_2)$	$f_1 - f_2$
$f_{\text{sqrt}}(f)$	$\sqrt{f}$
$f_{\text{sqre}}(f)$	f^2

Figure 2.1: A defining function for a unit disk depicted as a function composition tree where each node in the tree represents an algebraic operation. x and y represent the x and y components of a given point in space for which the function is evaluated at.

significantly in regards to implementation to enable support of additional features such as JIT compilation.

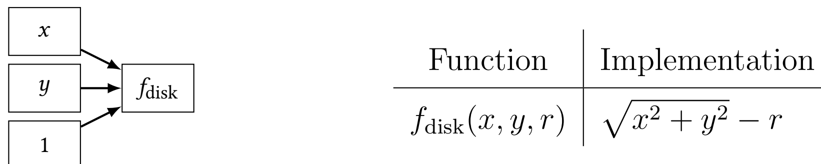
An example function composition tree associated with Equation 2.6, which defines a unit disc in $\mathbb{R}^2$ or a unit radius cylinder aligned with the z-axis in $\mathbb{R}^3$, is provided in Figure 2.1.

$$f_{\text{disk}}(x, y) < 0 \quad \text{where} \quad f_{\text{disk}}(x, y) = \sqrt{x^2 + y^2} - 1 \tag{2.6}$$

The interpreter traverses the composition tree, evaluating the function associated with each node, to build the model. While this approach to evaluation is typically less efficient when compared to a model-specific program, we JIT compile and locally optimize the defining function to reduce interpreter overhead.

2.2.2 JIT Operation Grouping

Our interpreter supports a range of built-in mathematical operations, including functions corresponding to algebraic operations, which can be used to construct a model. In addition to the fixed set of base operations, our interpreter uses a novel JIT compilation scheme, allowing for dynamic expansion of the interpreter's function set to reduce the cost associated with interpreter evaluation.



Function	Implementation
$f_{\text{disk}}(x, y, r)$	$\sqrt{x^2 + y^2} - r$

Figure 2.2: A function composition tree representing the unit disk in which the algebraic operations have been merged into a single node. Instead of defining the disk using several functions as can be seen in Figure 2.1, JIT compilation allows for the defining function to be represented using one function which combines all algebraic operations together. This reduces the number of functions in the composition tree enabling faster evaluation by the interpreter.

A primary drawback of interpreters is the overhead required to fetch and execute instructions. Each node in the function composition tree has a corresponding instruction which represents a mathematical function to be called by the interpreter. The larger the tree, the more functions that need to be called, the greater the interpreter overhead necessary to evaluate the tree. When performing localized evaluation, the total number of instructions can be reduced by eliminating, or "pruning", branches of the function composition tree which cannot affect local geometry [42]. We apply pruning when possible but, as detailed in Chapter 2.2.5, only certain classes of operations are capable of pruning the function composition tree. Defining functions containing few to no prunable candidate operations either cannot be locally optimized or may not offset evaluation cost enough to warrant usage of the interpreter. We employ JIT compilation as an additional optimization strategy to operations which do not meet the pruning classification.

To reduce interpreter overhead the function composition tree is simplified by grouping together sequential non-prune candidate operations. Each sequence of operations is merged, or "fused", together to form a single function, or "operation group", which requires only one interpreter instruction to execute. Operations which are capable of pruning the function

Algorithm 1: Constructs operation groupings for a function composition tree.

```
Function ConstructOpGroup(node, op_group) is
    for child in node.children do
        if child can prune then
            child_op_group ← new OpGroup()
            ConstructOpGroup(child, child_op_group)
            op_group.children.append(child_op_group)
        else
            op_group.merge(child)
            ConstructOpGroup(child, op_group)
```

composition tree are kept independent of operation groups as to preserve all possible local reductions of the function composition tree. Figure 2.2 demonstrates how the algebraic operations which form the unit disk's defining function from Equation 2.6 can be packaged into a single new function replacing the tree of composed algebraic functions from Figure 2.1.

Operation groups are generated automatically through the geometry kernel by traversing the function composition tree and recursively merging parent nodes with their children as described in Algorithm 1. Once the function composition tree is fully traversed, an additional optimization pass is applied to each operation group where repeat expressions and intermediate values are identified for reuse and stored locally within slots. It should be noted that we do not implement this step directly and instead take advantage of NVIDIA's CUDA compiler to handle value reuse identification. Instead, we topologically sort each operation group subgraph and store values using static single-assignment. The compiler then produces optimized code during interpreter compilation.

Listing 2.1 shows the result of the topological sorting pass applied to the f_{disk} operation group from Figure 2.2. For reasons explained in Chapter 2.2.6, identification of repeat

expressions is limited to operation groups and is not applied to the function composition tree as a whole.

Listing 2.1: C++ implementation of the f_{disk} operation group generated by the geometry kernel. Static single-assigment is used so that the CUDA compiler is able to further optimize the operation group implemenation.

```
float f_disk(float* stack){
    float var_0 = f_sqre(stack[0]); //x
    float var_1 = f_sqre(stack[1]); //y
    float var_2 = f_add(var_0, var_1);
    float var_3 = f_sqrt(var_2);
    float var_4 = f_sub(var_3, stack[2]); //1
    return var_4;
}
```

2.2.3 Operation Encoding

In order to facilitate communication between the CPU, where function composition trees are constructed, and the GPU interpreter, where function composition trees are evaluated, we assign each node of the function composition tree an operation identifier, or “opcode”, which the interpreter uses to obtain a reference to the associated function. The set of built-in operations are assigned opcodes which index into a fixed function table. For operation groups, the geometry kernel assigns opcodes that index into a dynamically generated JIT function table.

2.2.4 Compilation Pipeline

In addition to opcode generation, we register newly constructed operation groups with the interpreter so they can be compiled for use during evaluation on the GPU. We construct a string representation of each registered operation group to be JIT compiled using NVRTC, NVIDIA’s runtime compilation library [9]. NVRTC produces parallel thread exe-

cution (PTX) files which are loaded by the CUDA linker to generate a CUDA module. Each time a new operation group is defined, the module must be recompiled to update the JIT function table. Caching is employed at several levels of the compilation pipeline to both prevent unnecessary compilation and to make recompilation as fast as possible. Compilation is deferred until just before the interpreter is executed to avoid repeated compilations when introducing multiple new functions.

2.2.5 Pruning

After interpreter compilation, we move the function composition tree to the GPU. A final preprocessing step is applied to the defining function where branches of the function composition tree which have no affect on localized regions of the geometry are omitted, or "pruned". Pruning produces locally simplified function composition trees that preserve model geometry while avoiding unnecessary function evaluations. Because functions in the composition tree often influence the geometry only in localized regions, it is worthwhile to avoid unnecessary evaluations of a component function outside of the region where it contributes to defining the geometry. Without pruning, adding new localized features makes the function composition tree globally more expensive to evaluate, and large amounts of compute time can be spent evaluating functions within the composition tree which do not contribute to the local geometry.

The pruning component of the GPU interpreter is supported by employing interval arithmetic (IA) [57, 58], a technique initially developed as a method for tracking error inherent in finite-precision floating point representation of real numbers. IA involves extending functions from operating on real inputs to operating over intervals representing sets of real numbers. A generic interval $\hat{x}$ is defined as:

$$\hat{x} = (\underline{x}, \bar{x}) \text{ where } \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \bar{x}\} \tag{2.7}$$

The interval extension, $\hat{f}$, of the real function f, satisfies the following condition:

$$\{f(x) \in \hat{f}(\hat{x}) \;\forall\; x \in \hat{x}\} \tag{2.8}$$

An interval extension (IE) operates on intervals and returns an interval whose range contains all possible output values of the real function for any real value in the input interval. We use IA to enable pruning by identifying components of the function composition tree that do not contribute to the model's geometry in a given sub-region [42].

Pruning is performed during evaluation of the IE of the model's defining function which we generate by replacing each function, f_n, comprising the function composition tree with its IE, $\hat{f}_n$. We built our own IA library which defines the IE of standard mathematical operations using a suite of rounding-controlled floating-point intrinsics provided within CUDA. A point, $\boldsymbol{p} = \{x, y, z\}$, provided as input to the defining function is also replaced with an axis aligned box represented through coordinate intervals, $\hat{\boldsymbol{p}} = \{\hat{x}, \hat{y}, \hat{z}\}$. As the function composition tree is traversed, attempts to prune the tree are initiated at any node containing an operation designated as a "prunable candidate". Any binary function belonging to the fixed set of built-in interpreter operations is classified as a candidate for pruning if assigned a "prune function" within the geometry kernel. A prune function specifies the criteria by which one of the binary function's interval inputs is incapable of contributing to the output. When the prune function successfully identifies an input as not contributing, the branch of the function composition tree producing the non-contributing interval can be omitted without modification to the resulting geometry within the interval region $\hat{\boldsymbol{p}}$. The pruned function composition tree can then be used in place of the unpruned function composition tree when evaluating any point or subinterval contained in $\hat{\boldsymbol{p}}$.

The most straightforward candidates for pruning in a F-Rep implementation of CSG are min and max, which implement union and intersection respectively. The prune functions associated with min and max are able to identify non-contributing components of the function composition tree by checking for disjoint input intervals (Algorithm 2). An illustration which uses color-coding to demonstrate how pruned function composition trees manifest themselves within a model is provided in Figure 2.3. The model's defining function consists of a single

Algorithm 2: Prune function associated with the min operation. Each choice is comprised of two bits, one indicating if an input has been pruned and the other specifying which input was pruned.

```
Function PruneFuncMin(lhs, rhs) is
    if lhs < rhs then
        return LHS_CHOICE
    else if rhs < lhs then
        return RHS_CHOICE
    else
        return NO_CHOICE
```

prune candidate function, $f_{\min}$, which combines a sphere and block resulting in a function composition tree with a total of three pruned tree permutations. Each color on the model corresponds to one of the three pruned tree arrangements. The first pruned configuration, shown in green, appears at the interface between the sphere and the block. In this region the input intervals provided to $f_{\min}$ during tree pruning are not disjoint, indicating that both f_{sphere} and f_{block} are capable of contributing to the model's geometry. Pruning in this region produces a function composition tree with no omitted operations and evaluation of the defining function requires traversal of the full unpruned tree. In the red and blue regions of the model the input intervals to $f_{\min}$ are disjoint, producing pruned function composition trees with removed branches. Areas of the model in red denote a pruned tree which omits f_{sphere} while blue regions omit f_{block}.

We extend the list of supported prunable candidate functions beyond the min and max operations in the geometry kernel from Keeter [42] by including prune functions for addition, subtraction, multiplication, and division. Inclusion of the algebraic operation also motivates our support to introduce user control to selectively disable pruning candidacy. While pruning of min and max inputs is generally desired, the algebraic functions require much stricter criteria when identifying non-contributing inputs and do not typically result in successfully

Figure 2.3: A rendering of an F-Rep model with shading based on a color-coding of pruned function composition trees. Each color represents a unique pruned tree configuration assigned to a leaf node of the sparse volume structure used to store the model. There are a total of three pruned tree permutations: f_{sphere} pruned (red), f_{block} pruned (blue), and no pruning (green). f_{block} and f_{sphere} each represent an operation grouping containing the defining functions for the block and sphere respectively, while f_{min} defines the union operation via the min function.

pruned inputs. For example, addition requires one input to be tightly bounded to $(0, 0)$ for pruning to take place. However, there are many modeling operations like lofting, which routinely occur with intervals known to meet such algebraic pruning criteria. User-controlled disabling of operation prunability allows for pruning optimizations to be applied in situations where an operation is expected to produce a pruned output, and saves the interpreter from having to call prune functions when identification of non-contributing intervals is unlikely. The geometry kernel is also free to incorporate operation which are no longer prunable into operation groups, further simplifying the function composition tree.

2.2.6 Evaluation

Evaluation of either the model's defining function or IE involves traversing the function composition tree. A vector valued input $\boldsymbol{p}$, specifying either a three-dimensional region or point in space is provided to the interpreter to begin evaluation. Regions, represented through intervals, describe axis-aligned volumes over which the IE should be evaluated. Points, on

Algorithm 3: The interpreter's main loop. A context is constructed to store the evaluation stack. Traversal is achieved by alternating between descending and ascending the function composition tree until each node has been visited.

```
Function EvaluateTree(tree, p) is
    ctx ← CreateContext(p)
    node ← tree.root_node
    while true do
        node ← Descend(node, tree, ctx)
        node ← Ascend(node, tree, ctx)
        if node is tree.root_node then
            break
    return ctx.stack.pop()
```

the other hand, define a location for which the defining function should be computed. The input type of $\boldsymbol{p}$ controls if the scalar or IE function tables are loaded, dictating whether the defining function or its IE is evaluated. Given a point $\boldsymbol{p}$, the interpreter's main execution loop (Algorithm 3) traverses the function composition tree using a top-down approach, alternating between descent and ascent until each node has been visited (Algos. 4 and 5). Functions associated with internal nodes are evaluated after their children have been visited while leaf nodes are executed when reached (Algos. 6 and 7). Intermediate values produced while traversing the tree are stored within a stack-based memory system; leaf node evaluations push new values onto the stack and internal node evaluations both pop old values from, and push new values onto the stack. The essential distinction made between evaluation of the defining function and its IE is found in Algorithm 6, where we call the prune function associated with the prunable candidate operation which can lead to pruning of the input branches.

Each region provided as input to the IE of the defining function can potentially have a distinct pruned function composition tree. Minimizing the memory consumption of prune

Algorithm 4: Descends the function composition tree to a leaf node. Nodes which have been pruned use the pruned tree bitmask to alter traversal directions.

```
Function Descend(node, tree, ctx) is
    while node has children do
        if node has been pruned then
            choice ← ctx.choice[node.idx]
            node ← tree.nodes[node.children[choice]]
        else
            count ← ctx.count[node.depth]
            node ← tree.nodes[node.children[count]]
    return node
```

Algorithm 5: Ascends the function composition tree from a leaf node until an internal node is reached which has not been pruned and has children yet to be traversed. Each node reached during the ascent which no longer has children to traverse and has not been pruned has its function evaluated.

```
Function Ascend(node, tree, ctx) is
    while node is not tree.root_node do
        node ← tree.nodes[node.parent]
        count ← ++ctx.count[node.depth]
        if count = len(node.children) then
            EvaluateNode(node, ctx)
        else if node is not pruned then
            break
    return node
```

trees is crucial, as we are concerned primarily with large volumes containing many regions over which a significant number of pruned trees may be generated. To avoid excessive memory utilization we store the locally optimized trees in a compact representation which does not require generating copies of the initial function composition tree. Successfully pruned branches are encoded as traversal decisions which are used during subsequent evaluations to skip over irrelevant components of the defining function. Each prune decision is stored as two bits, a "prune bit" and a "choice bit". An active prune bit specifies that a branch of the tree has been pruned, while the choice bit determines the direction to traverse if pruned. During evaluation of the defining function's IE, when a prunable candidate operation is encountered with an inactive prune bit its prune function is evaluated. If either input to the function is determined not to contribute, the prune and choice bit associated with the prunable operation are updated and used on subsequent visits to direct traversal, skipping pruned branches (Algorithm 4 and 6). Each prune and choice bit pair constitute a single prune decision and, when combined with all other prune and choice bits, form a bitmask representing a pruned function composition tree. The bitmask serves as a compact and lightweight representation allowing for millions of pruned tree copies (associated with numerous spatial sub-domains) to be stored with minimal memory overhead. The bitmask-based approach helps maximize memory access speeds by maintaining only a single copy of the function composition tree which can be cached and accessed efficiently by all GPU threads.

Our need to minimize pruned tree memory usage also makes it difficult to apply the slot-based value-reuse optimization, used within operation groups and in earlier work by Keeter, to the entire function composition tree. Complications arise in the slot-based method with the inclusion of branch pruning; as branches are pruned, slots may become inactive. Information pertaining to the slots each operation reads from and writes to must be updated and maintained for each pruned tree. This information becomes prohibitively expensive when storing even a moderate number of pruned trees. While implementation differences prevent direct comparison between our pruned tree representation and Keeter's, estimates can be made to illustrate the memory savings of our approach. For example, Keeter benchmarks

Algorithm 6: Determines if a tree node can be pruned and calls the necessary prune function if prunable. After attempting to prune the node, the function associated with the node is called.

```
Function EvaluateNode(node, ctx) is
    if ctx.type is INTERVAL_TYPE and node is prunable then
        args ← ctx.stack.pop()
        choice ← ctx.prune_table[node.opcode](args)
        ctx.choice[node.idx] ← choice
        ctx.stack.push(args)
    EvaluateFunction(node, ctx)
```

Algorithm 7: Evaluates the function associated with the node and pushes the return value onto the context stack. The fixed table consists of all the built-in operations supported by the interpret while the JIT table contains all JIT compiled operation groups.

```
Function EvaluateFunction(node, ctx) is
    args ← ctx.stack.pop()
    switch node.table do
        case FIXED_TABLE do
            f ← ctx.fixed_table[node.opcode](args)
        case JIT_TABLE do
            f ← ctx.jit_table[node.opcode](args)
    ctx.stack.push(f)
```

his geometry kernel using a bear head model, provided by Hazel Fraticelli and Anthony Taconi, with a defining function consisting of 541 terms, 27 of which are CSG operations. Keeter allocates pruned tree nodes in blocks of 64, totalling 512 bytes (8 bytes per node). Provided every pruned tree is reduced to a single 64 node chunk, the lower bound on memory consumption for a single pruned tree is 512 bytes. Our pruned tree bitmasks representation requires 2 bits per CSG operation, totaling 8 bytes per pruned tree, consuming at least 64 times less memory. Storing $1,000,000$ unique pruned trees requires between 512 MB and 4.5 GB using Keeter's geometry kernel while our implementation needs only 8 MB.

Instead of using the slot-based memory system, our solution is to implement the hybrid scheme detailed in this section. A stack is used to store output produced during traversal of the function composition tree, while slots are used within the confines of operation groups to maintain support of value reuse. Prunable candidate operations are excluded from operation groups preventing value reuse between branches, leaving slots within operation groups unaffected when a branch is pruned. The hybrid implementation allows for value reuse without the overhead required to track slot related information.

2.3 GPU Sparse Voxel Database

2.3.1 Data Layout

To manage high resolution models we use a spacial subdivision approach to store function values and pruned function composition trees produced by the interpreter in a compact and accessible form. We implement an N^3-tree sparse volume data structure which builds from the memory pooling system of both OpenVDB and GVDB [61, 35]. The design of the N^3-tree consists of nodes at multiple levels forming a grid hierarchy. The subdivision factor, N, of the hierarchy is configurable in much the same way as OpenVDB and GVDB; a target tree topology configuration is specified as a vector of values representing the $\log_2$ resolution of each grid level. For example, a configuration of $\langle 4, 3, 2\rangle$ corresponds to a 3-level tree where each leaf brick contains $(2^4)^3$ voxels, interior nodes contain $(2^3)^3$ leaf node children, and top

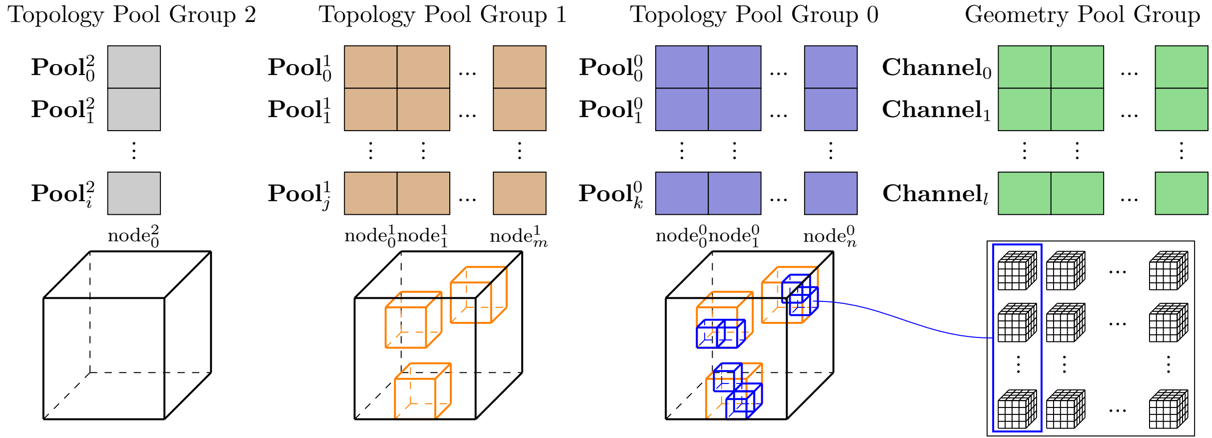


Figure 2.4: A visual representation of the data and topology layout of our N^3-tree sparse data structure. Each depth level of the tree has an associated pool group which stores information on the tree. The pool group is divided into individual pools each storing a grouping of data, or “property”, for all nodes at a given depth level. Every pool can be accessed via an index to obtain a node-specific copy of a property. The root depth of the tree consists of a single node while the internal and leaf depths can contain multiple nodes. In addition to the pool groups which describes the topology layout, a geometry pool group containing voxel bricks storing output of the interpreter is allocated alongside the leaf depth. The geometry pool group is divided into channels each of which describes a single value at a given voxel location. Leaf nodes reserve voxel bricks from the geometry pool group to store model information generated by the interpreter corresponding to the node’s location in space.

level nodes contain $(2^2)^3$ interior node children. The resulting sparse grid supports up to 512^3 voxels.

We store the N^3-tree in an efficient memory layout through the use of an allocator which provides the means to access the N^3-tree data on both the CPU and GPU. Museth and Hoetzlein observe that while node data and resolution vary between grid levels, nodes at a particular level are divided similarly. As such, we allocate a "pool group" at each tree depth level responsible for storing node data (Figure 2.4). A pool group is organized into individual pools of the same number of elements. Each pool within a pool group describes a collection of data, or "property", for all nodes, while entries of each pool can be accessed via an index to obtain all properties for a single node. Organization through memory pools and node properties provides a highly configurable approach to tree representation.

The default configuration of our N^3-tree consists of a primary pool group allocated at each depth level. This primary pool group is referred to as the topology pool group as it stores the connectivity information of the hierarchical grid. The topology pool group is composed of three properties; *Node*, *Children*, and *Geometry* (Figure 2.5). The *Node* property's primary responsibility is to maintain information related to the following:

- *depth*: topology pool group which the node belongs to
- *topology_idx*: location of the node within its topology pool group
- *parent*: topology_idx of the node's parent, located in the pool group one depth level higher
- *origin*: node's spatial position within the sparse grid

Together, *depth* and *topology_idx* form a value pair which uniquely define a node within the tree.

The *Children*, and *Geometry* properties track depth dependent information. Topology pool groups at depths other than the leaf depth store the *Children* property, maintaining

information on each node's children. The *Children* property defines a bitmask used to track active children and an index to store the position of the first active child in the topology pool group one depth level lower. During N^3-tree construction, a node's children are allocated in a single batch so that they are contiguous in memory. A child node can be accessed by adding *children_idx* to the sum of all active bits within *children_mask* up to the desired child node's position.

At the leaf depth of the N^3-tree, the subdivisions are associated with the grid at its full resolution. Node children are replaced with function values produced by interpreter evaluation of the model's defining function, sampled at each grid location spanned by the leaf node. The sample function values represent model geometry and are stored in small groups of voxels, or "voxel bricks", containing a number of elements equal to the leaf depth subdivision factor. The *Children* property is replaced with the *Geometry* property to track leaf node voxel brick ownership. Voxel bricks are kept in a separate pool group, referred to as the geometry pool group, which is allocated along side the leaf depth topology pool group. Individual pools within the geometry pool group are referred to as channels, terminology we adopt from GVDB, and contain voxel bricks which store information describing a single attribute (e.g. function values representing model geometry). The *Geometry* property contains an index, *geometry_idx*, pointing into the geometry pool group representing node ownership of a voxel brick. As voxel bricks are assigned to nodes, we update the contents via the interpreter. Additionally, the N^3-tree can be extended to support multi-resolution grids by allocating additional geometry pool groups and including the *Geometry* property alongside depth levels outside the leaf depth.

2.3.2 Construction

N^3-tree construction involves identifying and selectively activating nodes within the tree which contain model geometry. We denote the set of active N^3-tree nodes which intersect the model as the model's topology. To obtain a model-node intersection, we must determine if the surface level-set of the defining function intersects the node (i.e. a multi-dimensional

equivalent to the root finding problem). The process of obtaining model-node intersections has historically limited usage of sparse volume structures due to the associated computational cost. Traditional methods involve evaluating a majority of the grid and activating regions which contain grid values within some user-definable range of the level-set defining the surface, a process which is computationally intractable for large grids.

Recall that an interval returned by a function's IE contains all possible output values of the real function for any real value in the input interval. We leverage this property of IA to determine active regions of the N^3-tree using the IE of the model's defining function. The output interval, $\hat{F}$, produced by evaluating the defining function's IE, contains all possible real values of the defining function within the input interval region $\hat{\boldsymbol{p}}$. The interval range of $\hat{F}$ then determines if the region $\hat{\boldsymbol{p}}$ may contain model geometry.

To build the N^3-tree we implement a recursive algorithm on the GPU which takes advantage of the interpreter and model defining function to simultaneously prune the function composition tree and determine active nodes of the N^3-tree (Algorithm 8). Our algorithm, *ConstructTopology*, is launched to initially operate on the root node of the N^3-tree. For each potential child node, a thread is created on the GPU to evaluate the IE of the defining function using the interpreter. Evaluation of a topology node's children occurs within a single thread-block on the GPU so that all threads in the block iterate over the same pruned function composition tree, eliminating thread divergence. The input $\hat{\boldsymbol{p}}$ provided to the interpreter by each thread corresponds to an axis-aligned box defined by intervals which cover the region occupied by the child node. Evaluating $\hat{F} = \hat{f}(\hat{\boldsymbol{p}})$ allows for complete classification of the child node. One of three possible outcomes is produced based on the output interval result:

- $0 < \hat{F}$: The child node is fully outside the model and does not need to be allocated.
- $0 \in \hat{F}$: The child node cannot be classified as fully interior or exterior to the model. It may contain a portion of the model so further subdivide is required.

Node property	
Attribute	Type
depth	uint8
flags	uint16
origin	uint32[3]
parent	uint64
topology_idx	uint64

Children property	
Attribute	Type
children_mask	uint32[$(2^N)^3/32$]
children_idx	uint64

Geometry property	
Attribute	Type
geometry_idx	uint64

Figure 2.5: Each data structure defines a property used to represent our N^3 tree. The value of N within the *Children* property represents the $\log_2$ number of children for a given tree depth level specified in the topology configuration.

- $0 > \hat{F}$: The child node is fully interior to the model. If dynamic topology, introduced in Chapter 2.3.3, is enabled then the node is marked as dynamic to be constructed later if needed. Otherwise, the node is subdivided to further resolve the grid.

In each case where the grid must be subdivided, a recursive call to *ConstructTopology* is made in which the root node input is replaced by the newly constructed child node. A parent node's interval range is subdivided such that each child node interval is a sub-interval of its parent, allowing the pruned function composition tree output of the parent node to be used in place of the non-pruned function composition tree during evaluation. Subdivision is repeated until a maximum depth is reached or all children have been fully classified. The minimal memory footprint of our bitmask-based pruned tree representation allows for all pruned trees generated during topology construction to be stored alongside the associated node by introducing a new pool property to the N^3-tree which stores each bitmask. Additionally, the node classification provided by $\hat{F}$ is encoded within each N^3-tree node to differentiate nodes interior to the model from nodes which might contain a portion of the model's surface.

Once model topology is determined, model geometry is evaluated on-demand at active

Algorithm 8: Identifies and activates nodes of the N^3-tree using the IE of the model's defining function.

```
Function ConstructTopology(tree, depth, parent) is
    N ← log2_child_count(depth)
    for n in (2^N)^3 in parallel
        p̂ ← child_region(parent, n)
        F̂ ← EvaluateTree(tree, p̂)
        if 0 ∈ F̂ then
            set_bit_on(parent.children_mask, n)
        else if F̂ < 0 then
            if dynamic topology enabled then
                set_bit_on(parent.dynamic_mask, n)
            else
                set_bit_on(parent.children_mask, n)
    count ← count_on(parent.children_mask)
    children ← reserve_memory(count)
    for n in (2^N)^3 in parallel
        if bit_is_on(parent.children_mask, n) then
            child ← initialize_child(parent, children, n)
            if depth ≠ 0 then
                ConstructTopology(tree, depth - 1, child)
```

N^3-tree leaf nodes. Voxel brick assignment and geometry evaluation occur when a value from the grid is requested. The N^3-tree leaf node containing the desired grid value reserves a voxel brick and updates the brick contents using the interpreter. The pruned function composition tree associated with the region spanned by the node is used in place of the unpruned tree to reduce evaluation time. When the number of active leaf nodes is less than or equal to the geometry pool group size, each leaf node can be assigned a unique voxel brick. In situations where the leaf node count exceeds geometry pool group capacity, voxel bricks are obtained from the geometry pool group in a least recently used (LRU) fashion. By using the interpreter to evaluate the geometry as-needed, we reduce the in-memory footprint of the sparse grid to that of just the grid topology.

2.3.3 Dynamic Nodes

Our N^3-tree structure optionally makes use of "dynamic nodes" which we introduce to decrease the memory required to store model topology by allocating and constructing a bulk of the N^3-tree dynamically. Models which approximately fill the space spanned by an N^3-tree occupy close to the entire volume, resulting in nearly every node at every level of the N^3-tree being allocated. In such scenarios, nodes interior to the model comprise a majority of the total node allocations. As a result, the handling of "interior nodes", nodes fully inside the model, has the potential to drastically reduce memory requirements for storing model topology. Some sparse grid structures, including OpenVDB and GVDB, support alternative approaches to interior node representation. One such approach is to replace interior node children with a constant value, assumed to apply to all grid entries spanned by the node, indicating the grid should not be subdivided further. Another strategy is to disregard interior node allocations altogether and only allocate memory for nodes intersecting the model's narrow band surface level-set. In rendering-related applications these approaches are often satisfactory as the focus is on the model's surface geometry. In other applications like CAD, function values within the model can be sufficiently important as to require the grid be fully resolved interior to the model.

Existing internal node representations do not provide ways to reduce topology memory requirements without sacrificing grid resolution. Our solution to handling the issue of interior node allocation is to implement dynamic nodes, which provide a means to fully resolve the grid through dynamic construction of the model's topology. To accommodate dynamic nodes the topology pool groups comprising the N^3-tree are initially sized to fit only non-dynamic, or "static", nodes (i.e. nodes which cannot be classified as fully interior or exterior) and are extended by a user-controlled value, D, of new nodes which represent dynamic nodes. An additional property is added to each topology pool group to track dynamic children as well as dynamic node ownership.

During N^3-tree assembly, when dynamic topology is enabled, nodes determined to be interior to the model's geometry are classified as dynamic and are not subdivided further. The newly added dynamic property contains a bitmask which uses set bits to indicate dynamic status of child nodes. Once the N^3-tree is built, if a child node is requested which corresponds to an active entry in the parent node's dynamic bitmask, the parent node reserves one of the allocated dynamic nodes, sets the dynamic node as its child, and records the node index to reflect dynamic node ownership. This process is repeated at each depth level until a dynamic leaf node is constructed, which can then be assigned a voxel brick to be evaluated by the interpreter. The result is a "dynamic branch" which extends from the first non-dynamic parent node down to a leaf node and remains active until any dynamic node within the dynamic branch is reassigned to another dynamic branch. By adjusting the value of D, internal node allocations can be directly controlled. Setting D to zero eliminates internal node allocations altogether, useful in situations where grid values inside the model are not needed (e.g. when rendering the surface of the model). This produces topology with a memory footprint comparable to existing alternative interior node representations. In situations where interior nodes are needed, the number of allocated dynamic node count can be set accordingly.

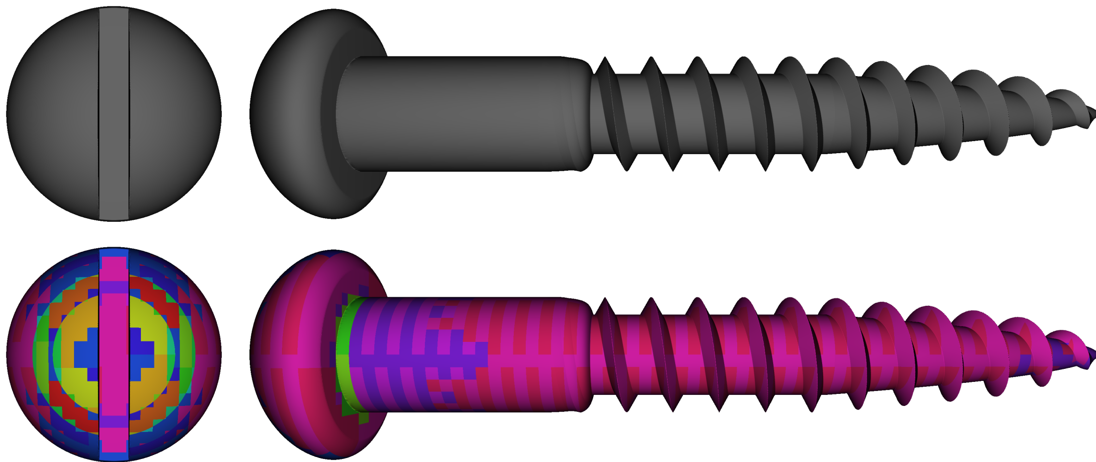


Figure 2.6: A rendering of an F-Rep wood screw designed and visualized using our geometry kernel. The top row shows two views of the wood screw rendered with shading based on a simple diffuse illumination model. The images in the bottom row show the same views of the wood screw but with shading based on a color-coding of the pruned function composition tree. Each colored region on the screw corresponds to a particular pruned tree configuration belonging to an N^3-tree leaf node. The color is generated by converting the bits of a node's pruned tree into a single integer, scaling the integer to the range $[0, 1]$, and mapping that range to the HSV color space. Prune bits which appear earlier in the pruned tree cause larger color differences from pruned bits which appear later in the tree due to occupying more significant bits of the resulting integer value.

2.3.4 Visualization

We display models with a ray-casting rendering algorithm that makes use of the hierarchical 3D digital differential analyzer (DDA) developed by Museth [59] and later adapted to the GPU by Hoetzlein. We divide the rendering process into two phases which are repeated until each ray intersects the model or exits the N^3-tree. We also allocate a node buffer and a ray buffer to track ray information between passes. The node buffer stores *topology_idx* values of leaf nodes while the distance buffer maintains information on total distance traveled along each ray.

The first rendering phase involves determining ray-model intersections. To start, rays are tested for intersection against the N^3-tree. Any intersection found results in traversal of the N^3-tree via the 3D DDA until a leaf node is found or the tree is exited. Successful leaf node intersections result in traversal of the leaf node's voxel geometry brick. We assume only C^0 continuity of the model's defining function and use a uniform step size to march through a voxel brick. Interpolation is used to estimate values of the defining function at points along the ray until either the brick is exited or a ray-surface intersection is found. For shading purposes, surface normals are determined by computing the gradient of the function at the point of intersection (e.g. using finite differences). In cases where the class of defining function is limited, for example to signed distance functions, techniques like sphere tracing can be integrated into the DDA to dynamically set the voxel step size to enhance performance.

During traversal, active leaf nodes encountered which have not yet been assigned a voxel brick cause the ray to stop traversal and record the current ray distance and leaf node *topology_idx* in the distance and node buffer. To prevent a node from being added to the node buffer multiple times, one bit is reserved within the *flag* attribute of the *Node* property which is checked and atomically set before an attempt is made to record the node. The second rendering phase is triggered if the node buffer contains a non-zero number of entries after the first phase completes, indicating that active leaf nodes were hit that have not yet

been assigned geometry. Unassigned voxel bricks, or voxel bricks belonging to nodes which have already been traversed during the first phase, are assigned to nodes recorded in the node buffer. The contents of the newly assigned voxel bricks are updated by the interpreter and the node flag is reset. Because the interpreter is implemented on the GPU, bricks can be updated directly on the GPU avoiding costly memory transfers between the CPU and GPU. The first phase is repeated and each ray uses the value recorded in the distance buffer to resume traversal from where it left off. These two rendering phases are repeated until the first phase finishes with an empty node buffer indicating that the rendering is completed. Because geometry is evaluated on-demand and evaluation is informed by ray-model intersection, only visible N^3-tree nodes have their voxel bricks contents updated, reducing total evaluation overhead when viewing large models.

2.3.5 Slice Decomposition

Model slice decomposition, or "slicing", is the process by which a model is decomposed into a representation 3D printers can interpret to enable manufacturing of a physical copy of the model. For high-resolution volumetric models, a raster-based printing process is typically used as the means of manufacturing and takes as input a stack of images corresponding to horizontal slices of the model. Model slicing within our framework is enabled through the N^3-tree and consists of two steps. The first step involves iterating over active N^3-tree leaf nodes and collecting all voxels along a particular z-height into a slice buffer. Similar to visualization, all leaf nodes which have not yet been assigned geometry are assigned voxel bricks from the geometry pool group which are updated using the interpreter before their voxel contents are extracted. The second and final step is to move the slice buffer from the GPU to the CPU where the buffers are saved as images. Support can be extended to vector-based printers by introducing a third step which extracts the level-set contours defining the model's surface from the slice buffer.

Printer resolution along the Cartesian axes of a raster-based additive manufacturing system are typically not identical, for example the Stratasys J750 PolyJet 3D printer has

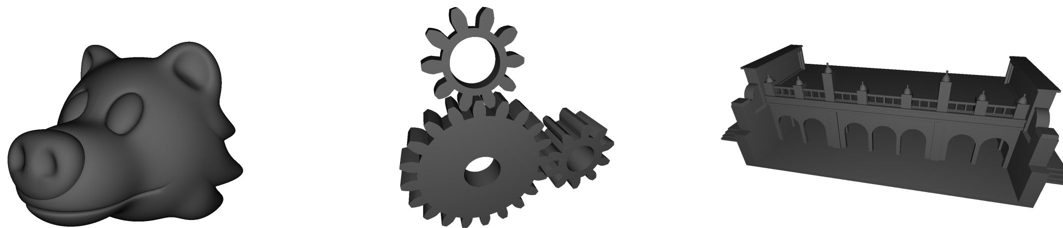


Figure 2.7: The bear head, gear, and architecture model from [42] used to benchmark our geometry kernel.

an x and y resolution of 42 microns while its z axis has a resolution of 14 microns. To accommodate axes with different resolutions, we implement coordinate-specific grid spacings to support sampling on non-cubic voxels. Evaluation of the defining function and its IE remain the same, all that is altered is the value $\boldsymbol{p}$ provided as input to the interpreter. The resulting slice images are distorted but recover the original geometry when printed due to the printer scaling along each axis.

2.3.6 Bulk Properties

A model's physical characteristics, like center of mass, are useful for validating aspects of the model's design. Computing these properties requires integrating over the model's geometry. We evaluate volume, surface, and line integrals using methods presented by Yurtoglu et al. [91], an integration approach which operates on arrays of function values sampled on a regular grid. Integral contributions arise from a stencil computation applied to each grid point. Non-zero contributions occur where the stencil crosses over the level-set defining the model's surface. The node classification provided by IA during construction of the model topology allows for efficient filtering of topology nodes which do not contain the model's surface. Only nodes which produce a non-zero integral contribution are evaluated.

2.4 Results

Waiting minutes to hours to construct and view models has a severely negative effect on productivity. The productivity consideration puts a significant emphasis not just on supporting large scale volumetric models, but doing so at reasonable rates on the kind of computing hardware typically used by designers and engineers. To validate our geometry kernel's responsiveness we characterize its performance in two ways. First, we compare our geometry kernel with the kernel presented in Keeter [42], which at the time was a state-of-the art GPU implementation of libfive [41], a library used as a basis for the kernel within the CAD software nTopology. Comparisons are made through a render-based workload involving complex models. Second, we benchmark the geometry kernel under a standard additive manufacturing workload involving generation of raster slice data of a high resolution model. In addition to benchmarking, we utilize our slicing pipeline to produce slice stack data of a wood screw which we fabricate using a 3D printer. All discussion and tables below describe performance on two systems: a lower end laptop equipped with a GTX 1650-Max Q GPU and Intel i7-1065G7 CPU as well as a higher end desktop computer containing an RTX 3090 and Intel i7-11700K. These systems are chosen to illustrate the capabilities of the geometry kernel on hardware corresponding to well-separated portions of the spectrum of computing power.

2.4.1 Performance Comparison

Due to substantial differences in geometry kernel implementation, direct comparisons of individual components of our kernel and the kernel presented by Keeter are difficult. The kernel implementation from Keeter requires that the interval pruning and geometry evaluation of a model be reevaluated every frame, and uses a screen-aligned and orthographic coordinate system which ties the model resolution directly to the resolution of the screen (e.g. a 2048^3 model volume requires a 2048^2 screen resolution to visualize). Within our kernel implementation the N^3-tree stores pruning information as well as model geometry, thus evaluation of these properties need not be performed each frame. The N^3-tree is built once,

Table 2.1: Rendering benchmark results for the bear head model over various grid sizes and topology configurations. Performance was measured on a laptop equipped with a GTX 1650 Max-Q and a desktop containing a GTX 3090. Table entries containing a dash denote results which are omitted due to requiring either an unreasonable time scale to complete or more memory than is available on the GPU.

	Configuration				FVD								MPR	
Grid	Topology	# of Nodes		Node	Build (ms)		Trav. (ms)		Eval. (ms)		FPS		FPS	
		Static	Dynamic	Occup.	1650	3090	1650	3090	1650	3090	1650	3090	1650	3090
512^3	$\langle 3,3,3\rangle$	34,696	55,547	34%	15	2.3	110	31	81	12	42	150	12	84
	$\langle 4,3,2\rangle$	8,247	5,585	42%	3.1	0.7	120	29	110	18	32	140		
	$\langle 5,3,1\rangle$	1,882	475	58%	1.0	0.7	160	45	200	51	23	100		
1024^3	$\langle 3,3,3,1\rangle$	139,874	504,411	31%	62	7.8	140	32	190	25	40	150	3	23
	$\langle 4,3,3\rangle$	34,696	55,547	34%	15	2.3	180	54	290	44	26	100		
	$\langle 5,3,2\rangle$	8,247	5,585	42%	3.0	0.7	210	55	560	94	18	78		
2048^3	$\langle 3,3,3,2\rangle$	556,700	4,300,532	29%	210	30	200	43	2400	300	28	110	—	4
	$\langle 4,3,3,1\rangle$	139,874	504,411	31%	61	8.0	190	51	810	100	22	87		
	$\langle 5,3,3\rangle$	34,696	55,547	34%	15	2.3	280	84	1600	220	14	58		
4096^3	$\langle 3,3,3,3\rangle$	2,223,035	35,495,926	28%	640	120	—	92	—	930	—	84	—	—
	$\langle 4,3,3,2\rangle$	556,700	4,300,532	29%	210	30	—	55	—	1300	—	63		
	$\langle 5,3,3,1\rangle$	139,874	504,411	31%	61	8.5	—	88	—	640	—	49		
8192^3	$\langle 3,3,3,4\rangle$	8,897,005	288,394,003	27%	—	370	—	170	—	950	—	68	—	—
	$\langle 4,3,3,3\rangle$	2,223,035	35,495,926	28%	—	115	—	100	—	3900	—	48		
	$\langle 5,3,3,2\rangle$	556,700	4,300,532	29%	—	—	—	—	—	—	—	—		

before rendering begins, and geometry is evaluated and stored as needed in the geometry pool group of the N^3-tree. Additionally, our rendering approach allows for a fixed dimension screen to render a model of any resolution. The N^3-tree also provides support for advanced rendering techniques like level-of-detail control which significantly improves performance for higher resolution models, when geometry voxel bricks approach the size of a pixel. Given these large differences, we use rendering frame-rate of the model at its maximum resolution as the metric for comparison as it best quantifies end-to-end performance of each geometry kernel.

Tables 2.1, 2.2, and 2.3 contain performance metrics for three different models, shown in Figure 2.7, from [42] using both Keeter's and our geometry kernel. The bear head model supports limited pruning due to the small number of min and max functions, making it a good candidate for JIT optimization. The gear and architecture model contain larger numbers of min and max functions, limiting JIT optimization. Results for our kernel are presented under the table header "FVD" (function-based volumetric design) while Keeter's are contained under the "MPR" (massively parallel rendering) header. We include results for each geometry kernel up to the point at which either the GPU runs out of memory or produces a frame-rate that is lower than one frame-per-second, denoted by a dash.

Table 2.1 includes an extensive parameter sweep of various topology configurations to characterize the affects of model topology on rendering performance and topology construction, while the remaining tables present more concise results covering only the topology configuration which produces the most favorable rendering performance for a given grid size. The "Configuration" header in Table 2.1 details the topology configuration, number of active N^3-tree leaf nodes, and occupancy rate specifying the percentage of the dense grid occupied by the model. To obtain the number of active voxel elements contained within the sparse grid, the grid size can be multiplied by the occupancy fraction. For our geometry kernel, in addition to frame-rate, we include timing results which detail additional work required for the first render frame. In particular, we present timing results for the sparse grid construction, as well as visible geometry identification and evaluation. Lastly, all rendering results

Table 2.2: Rendering benchmark results for the gear model.

Grid	FVD								MPR	
	Build (ms)		Trav. (ms)		Eval. (ms)		FPS		FPS	
	1650	3090	1650	3090	1650	3090	1650	3090	1650	3090
512^3	20	5.0	64	26	73	10	50	180	30	133
1024^3	55	10	96	23	250	27	40	150	11	70
2048^3	240	29	160	34	860	81	31	120	3	25
4096^3	650	110	270	53	5600	800	22	86	1	7
8192^3	—	480	—	120	—	1100	—	65	—	—

for our kernel are determined using a fixed screen size of 1280×720.

Comparing frame-rate results between geometry kernels shows that our kernel outperforms Keeter's on each of the three models, with rendering improvement of over 10×. We are also able to visualize models at interactive rates at grid sizes previously unachievable. Variability in rendering performance of our geometry kernel is due in large part to geometry pool capacity and N^3-tree topology configuration which are discussed in detail in the following two paragraphs.

Geometry pool capacity dictates the number of geometry bricks which can be stored and reused for multiple rendering frames. Topology nodes which have not yet been assigned geometry prevent rays from progressing through the N^3-tree while rendering, requiring that new geometry be evaluated by the interpreter and that the tree be traversed again once the geometry has been updated, as described in Section 2.3.4. The timings associated with this process are provided under the "Traverse" and "Evaluate" table columns. Minimizing the number of tree traversal passes is key to achieving an optimal frame-rate. The ratio of visible geometry to geometry pool capacity should preferably be less than or equal to one

Table 2.3: Rendering benchmark results for the architecture model.

	FVD								MPR	
Grid	Build (ms)		Trav. (ms)		Eval. (ms)		FPS		FPS	
	1650	3090	1650	3090	1650	3090	1650	3090	1650	3090
512^3	3.1	0.5	72	18	32	4.5	35	150	47	250
1024^3	11	1.2	89	23	88	14	32	117	16	116
2048^3	43	3.7	110	25	310	50	24	91	6	44
4096^3	180	15	140	28	1100	150	21	79	2	14
8192^3	—	59	—	30	—	180	—	77	—	—

so that complete tree traversal can be achieved in a single pass. The drop in frame-rate to below one frame-per-second when rendering each of the three models using our kernel is a result of visible geometry exceeding the allocated geometry pool capacity (and each GPU's total memory capacity). Each frame requires close to a complete reevaluation of all visible geometry greatly reducing performance.

The N^3-tree topology configuration specifies the spatial subdivision of the grid, and thus how the underlying grid is partitioned. Varying the N^3-tree topology alters 3D DDA traversal time as well as geometry voxel brick dimension. When looking at variations in leaf depth topology configuration for a given grid size in Table 2.1, the downward trend in frame-rate timings indicate that smaller leaf node sizes support more responsive rendering. Larger leaf nodes correspond to larger geometry bricks and an increased grid occupancy. The larger geometry bricks are also more costly to traverse when intersected during rendering and limit geometry pool brick allocations due to requiring more memory. Conversely, smaller leaf nodes increase total N^3-tree node count, raising model topology memory requirements. N^3-tree build times, shown under the "Build" column of each table, also increase with smaller

Table 2.4: Slicing benchmark results for the wood screw model over various grid sizes and topology configurations. Performance was measured on a laptop equipped with a GTX 1650 Max-Q and a desktop containing a GTX 3090. Table entries containing a dash denote results which are omitted due to requiring an unreasonable time scale to complete.

Grid	Topology	# of Nodes		Node	Evaluate (s)		CPU to GPU (s)	
		Static	Dynamic	Occup.	1650	3090	1650	3090
$128^2 \times 512$	$\langle 3,3,3 \rangle$	4,587	1,168	35%	0.130	0.055	0.086	0.023
	$\langle 4,3,2 \rangle$	1,170	48	59%	0.066	0.032		
	$\langle 5,3,1 \rangle$	228	0	89%	0.065	0.032		
$256^2 \times 1024$	$\langle 3,3,3,1 \rangle$	17,311	13,068	23%	0.390	0.120	0.220	0.085
	$\langle 4,3,3 \rangle$	4,587	1,168	35%	0.190	0.065		
	$\langle 5,3,2 \rangle$	1,170	48	59%	0.140	0.055		
$512^2 \times 2048$	$\langle 3,3,3,2 \rangle$	67,329	122,107	18%	1.300	0.450	1.200	0.409
	$\langle 4,3,3,1 \rangle$	17,311	13,068	23%	0.470	0.160		
	$\langle 5,3,3 \rangle$	4,587	1,168	35%	0.320	0.120		
$1024^2 \times 4096$	$\langle 3,3,3,3 \rangle$	265,867	1,054,306	16%	3.900	1.700	7.500	2.900
	$\langle 4,3,3,2 \rangle$	67,329	122,107	18%	2.100	0.780		
	$\langle 5,3,3,1 \rangle$	17,311	13,068	23%	1.400	0.360		
$2048^2 \times 8192$	$\langle 3,3,3,4 \rangle$	1,056,319	8,763,007	15%	23.000	8.400	59.000	21.000
	$\langle 4,3,3,3 \rangle$	265,867	1,054,306	16%	12.000	3.600		
	$\langle 5,3,3,2 \rangle$	67,329	122,107	18%	12.000	3.700		

leaf node sizes as a result of having to evaluate the IE of the model's defining function over more node regions. An optimal topology configuration is therefore application dependent, with smaller leaf nodes being more favorable for rendering and larger leaf nodes being more favorable to tree construction. It should be noted that we do not provide rendering results over varying internal node topology configurations. We use the 3D DDA algorithm from GVDB, for which metrics on how internal node size variations affects frame-rate are discussed by Hoetzlein [35]. We concur with the conclusion that an internal node configurations on the order of $N = 3$ provide optimal results.

2.4.2 Model Slicing

The second metric by which we gauge performance of our geometry kernel is through the model slicing process described in Section 2.3.5. To demonstrate the effectiveness of our slicing pipeline and characterize how it scales with grid size we design and slice a wood screw model depicted in Figure 2.6. Table 2.4 presents slicing performance metrics for the screw model over several topology configuration and increasingly larger grid sizes. The "Evaluation" column reports time taken to evaluate all leaf node geometry and extract the geometry into slice buffers, while the "GPU to CPU" column reports time needed to move the slice buffers from the GPU to the CPU. Total slice time is obtained by adding the two columns together.

Evaluating and filling the slice buffer dominates total slice time for smaller grid sizes while GPU to CPU transfer time becomes the dominant factor as the grid size increases. For a given grid size, slice evaluation time is affected by N^3-tree topology, primarily leaf node size. Leaf node geometry bricks with a voxel count that allows the GPU to achieve maximum compute throughput when evaluating a model's defining function produce more optimal timings. This typically corresponds to larger leaf node sizes. However, as leaf node size increases voxel grid occupancy also rises, equating to more total evaluations of the model's defining function. Evaluation count and compute throughput compete against one another resulting in grid-size dependent optimal topology configurations. For smaller grid

sizes, where the total voxel count is low, larger leaf brick sizes perform more favorably. As the grid size increases, a leaf node size which strikes a balance between voxel grid occupancy and compute throughput produces the best timings. For both the GTX 1650 and RTX 3090, the preferred configuration initially corresponds to one with a leaf node size of $(2^5)^3$ but transitions to $(2^4)^3$ at the largest grid resolution. We also use the slicing pipeline to fabricate the wood screw using a Stratasys J750 PolyJet 3D printer and provide images of a representative slice stack as well as the finished print in Figure 1.

2.5 Future Work

Here we present several directions for future research with a focus on performance improvements to the geometry kernel.

2.5.1 Topology Evaluation

Our topology construction algorithm builds a model's topology in a single pass. However, during visualization a majority of a model's topology may be occluded. Ideally, only the visible topology would be constructed. Extending our view-dependent geometry evaluation scheme to model topology construction would reduce both the compute and memory requirements needed for visualization. A view-dependent evaluation would also aid in further reducing pruned tree memory requirements as only the visible topology regions would produce pruned tree data.

2.5.2 Geometry Evaluation

Geometry evaluation consumes a bulk of total model evaluation time. We use techniques such as tree pruning and view-dependent evaluation to significantly reduce this evaluation cost. However, after any edit has been made to the defining function we discard all geometry and rebuild the model from scratch. Incorporation of strategies which identify geometry that does not change between model edits would further reduce the compute overhead required when frequently modifying models, a common occurrence in CAD applications.

2.5.3 Interval Bounds

We use IA to inform topology construction so that model geometry is not needed to determine region occupancy. IA is conservative in its evaluation estimate resulting in spurious topology node allocations which do not contain model geometry. Replacing or augmenting IA with alternative techniques, for example affine arithmetic, which improve accuracy would aid in generating topology that more tightly fits to the model's geometry, reducing both model evaluation time and memory requirements.

2.6 Discussion

We have presented an F-Rep geometry kernel designed for large scale volumetric modeling applications including large-volume, high-resolution CAD and additive manufacturing. The modeler consists of a novel F-Rep GPU interpreter and a sparse volume data structure. Our F-Rep geometry kernel builds from previous work by Keeter and provides improvements in the form of JIT compilation, increased function prunability, and a reduction of the memory footprint for storing pruned trees. The sparse volume data structure we develop is used to store interpreter output and provides the means to visualize and manipulate high resolution volumetric data. In addition, our geometry kernel framework performs well on GPU-enabled systems ranging from laptops to desktops, and further performance enhancements are expected as a result of ongoing improvements in GPU computing throughput, data transfer rates, and memory capacity.

Chapter 3

COAREA GRID-BASED VOLUME INTEGRALS

3.1 Background

3.1.1 Integration Techniques

Numerical evaluation of integrals is a task that finds significant application in a variety of areas including fluid mechanics [66], dynamics and vibrations [56, 72], naval architecture [6], economics [24], and many other fields. When dealing with numerical integration of functions of a single variable, this topic is often referred to as numerical quadrature [12]. There is a vast literature discussing methods for performing numerical quadrature [67] several of which date back to famous names like Newton [16] and Gauss [34]. Gonnet provides an extensive review of quadrature methods with a detailed discussion of error estimates [26].

Numerical integration over general multi-dimensional domains is identified by Thiagarajan and Shapiro as a central problem in engineering and scientific computation. They discuss two typical approaches: (1) conversion to a boundary integral using the divergence theorem, and (2) approximation of the domain by a union of simple cells for which there are well established integration methods. Noting that good approximations of complex geometry by non-conforming cells leads to boundary fragmentation and high computing costs, they proceed to develop an effective adaptive integration method with conforming boundary cells [85].

Yurtoglu et al. [91] present an alternative approach that combines voxel methods with wavelet analysis to compute integrals over implicitly defined domains. The voxelized nature of this approach enables computation of integrals based on grids of sample values of the integrand and the implicit function that defines the integration domain. The quadrature values arise as a sum of voxel contributions from a generalized stencil computation that is highly parallelizable enabling fast implementations. Formulations were presented for path, surface, and volume integrals; however the volume integral formulation relies on construction of a vector potential whose gradient corresponds to the integrand. This would require knowledge of the analytic form of the integrand.

3.1.2 Contributions

This works fills the gap in Yurtoglu et al. [91] by presenting an approach to fast computation of volume integrals based on regular grids of sampled data without requiring an analytic form of the integrand. The essential inputs to the integration method proposed in this paper consist of grids of regularly sampled values of the integrand and the function that implicitly defines the domain of integration. If instead the integrand and implicit defining functions are given directly as functions, those functions can be sampled on a regular grid to provide the expected inputs. Alternatively, if the domain is specified geometrically, there is extensive literature on how to construct function-based representations (F-Reps) for a wide range of geometry [17, 64, 79, 78]. The theoretical basis of this new approach is provided by combining Yurtoglu et al.'s grid-based surface integral formulation [91], traditional one-dimensional quadrature, and Federer's coarea formula [75]. Note that the method presented in this paper is intended for applications involving grids of sampled data (e.g. 3D imaging and direct numerical simulation).

3.2 Volume Integral Formulation

Federer's coarea formula provides a direct and reliable means for converting an integral over a domain implicitly defined by a function f to an integral over the level sets of f:

$$\iiint_{\Omega(0)} q||\nabla f||\ dV \equiv \int_{\mathbb{R}<0} \iint_{f^{-1}(\eta)\ \cap\ \Omega(0)} q\ dS\ d\eta \tag{3.1}$$

where $\Omega(\eta) = \{(x, y, z) \in \mathbb{R}^3 \mid f < \eta\}$ and we follow the convention that the integration domain is bounded by the level set $\eta = 0$.

The right hand side of the coarea formula can be separated into two parts: the inner two-dimensional integral which computes an area integral over a given level set and the outer one-dimensional integral which sums the contributions from each level set to produce the volume.

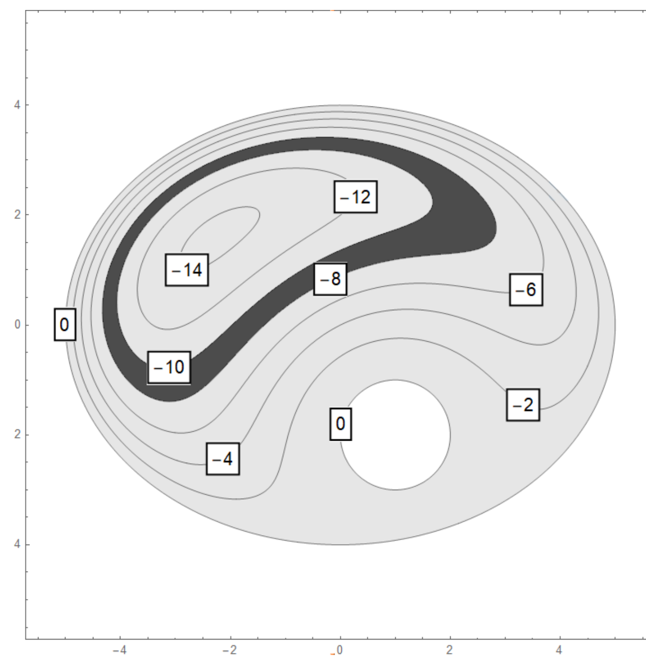


Figure 3.1: Level sets of the two-dimensional function $f(x,y) = \left((\frac{x}{5})^2 + (\frac{y}{4})^2 - 1\right)\left((x-1)^2 + (y+1)^2 - 1\right)$. The lightly shaded region shows the domain defined implicitly by $f < 0$. The darker region corresponds to the level-set contribution $\eta = -8$.

$$I_S(q, f, \eta) = \iint_{f^{-1}(\eta)\ \cap\ \Omega(0)} q\ dS = \iint_{\partial\Omega(\eta)} q\ dS \tag{3.2}$$

We evaluate the integral on the right-hand side of Eq. (3.2) using the grid-based surface integral formulation [91]:

$$\begin{aligned} I_S(q, f, \eta) &= \iint_{\partial\Omega(\eta)} q\ dS \\ &= \iiint_B -q\nabla\chi(f, \eta) \cdot \nabla f / ||\nabla f||\ dV \end{aligned} \tag{3.3}$$

where the occupancy function is

$$\chi(f, \eta) = \begin{cases} 1 & f \leq \eta \\ 0 & f > \eta \end{cases} \tag{3.4}$$

and B is an axis-aligned bounding box satisfying $\Omega(0) \subset B \subset \mathbb{R}^3$.

Computing the volume integral $\iiint_{\Omega(0)} g\ dV \equiv I_V(g, f, 0)$ corresponds to choosing $q = g||\nabla f||^{-1}$ which results in:

$$\begin{aligned} I_V(g, f, 0) &= \iiint_{\Omega(0)} g\ dV \\ &= \int_{\mathbb{R}<0} I_S(g||\nabla f||^{-1}, f, \eta)\ d\eta \\ &= \int_{\mathbb{R}<0} \iiint_B -g\ \nabla\chi(f, \eta) \cdot \nabla f / ||\nabla f||^2\ dV\ d\eta \end{aligned} \tag{3.5}$$

Since the geometric domain is implicitly defined by $f < 0$, the non-trivial contributions over $\mathbb{R}$ come from the finite interval $\eta \in [f_{min}, 0]$, where f_{min} is the minimum of f in $\Omega(0)$. The final version of the volume integral formula becomes

$$I_V(g, f, 0) = \int_{f_{min}}^{0} \iiint_B -g\nabla\chi(f, \eta) \cdot \nabla f / ||\nabla f||^2\ dV\ d\eta \tag{3.6}$$

Note that application of the coarea formula introduces an additional factor of $||\nabla f||^{-1}$ which acts as a weighting term that accounts for the local spacing between level sets. While

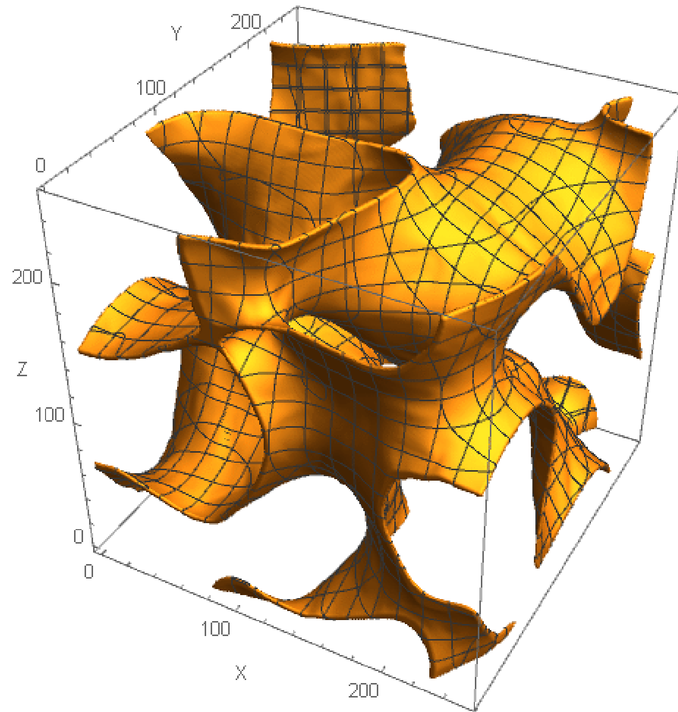


Figure 3.2: Visualization of the region of mixture fraction ratios between 0.3 and 0.4 of f_{DNS}

this factor can introduce infinities at singular points of f, the grid-based surface integral formulation already includes this factor so identification and removal of contributions from isolated singular points is built-in. (For non-pathological functions, the singular points form a set of measure zero, and their exclusion is assumed in the underlying convergence proof [69].)

3.3 Discretization

The coarea formula gives the volume integral as an integral over the level set values of integrals on the level sets, and our computational approach combines established methods for each. We employ the grid-based surface integral formulation for the integrals over the

level sets, and we apply a one-dimensional quadrature to sum the contributions from each level set. For simplicity of presentation we employ Simpson's rule, a well-known method from the Newton-Cotes quadrature family. (Newton-Cotes quadrature rules have uniform sampling, but other rules can be applied equally well.) We denote the spacing between level sets to be $\delta\eta$ and $\widetilde{I}_S(\eta)$ is defined as the surface integral of $g||\nabla f||^{-1}$ over the level set $f = \eta_t \equiv -t\,\delta\eta$.

$$\begin{aligned}\widetilde{I}_S(\eta) &\equiv I_S(g||\nabla f||^{-1}, f, \eta) \\ &= \iiint_B -g\nabla\chi(f,\eta)\cdot\nabla f/||\nabla f||^2\; dV\end{aligned} \tag{3.7}$$

Simpson's rule approximates the volume integral as a linear combination of the weighted surface integrals on the level sets corresponding to the discrete set of values $\{\eta_0, ..., \eta_T\}$ as follows:

$$\begin{aligned}I_V^{Simp}(g,f,0) = \frac{\delta\eta}{3}(&\widetilde{I}_S(\eta_0)+ \\ &4\widetilde{I}_S(\eta_1) + 4\widetilde{I}_S(\eta_3) + \cdots + 4\widetilde{I}_S(\eta_{T-3}) + 4\widetilde{I}_S(\eta_{T-1})+ \\ &2\widetilde{I}_S(\eta_2) + 2\widetilde{I}_S(\eta_4) + \cdots + 2\widetilde{I}_S(\eta_{T-4}) + 2\widetilde{I}_S(\eta_{T-2})+ \\ &\widetilde{I}_S(\eta_T))\end{aligned} \tag{3.8}$$

Each term corresponds to a surface integral described by Eq. (3.7) evaluated using the grid-based method [91] to produce voxel contributions of the form

$$\begin{aligned}C_{i,j,k}(\eta_t) &= h^3\left[-g\frac{\nabla f\cdot\nabla\chi(f,\eta_t)}{\nabla f\cdot\nabla f}\right]_{i,j,k} \\ &= h^3\left[-g\frac{\frac{\partial f}{\partial x}\frac{\partial\chi(f,\eta_t)}{\partial x}+\frac{\partial f}{\partial y}\frac{\partial\chi(f,\eta_t)}{\partial y}+\frac{\partial f}{\partial z}\frac{\partial\chi(f,\eta_t)}{\partial z}}{\frac{\partial f}{\partial x}^2+\frac{\partial f}{\partial y}^2+\frac{\partial f}{\partial z}^2}\right]_{i,j,k}\end{aligned} \tag{3.9}$$

with uniform grid spacing h. (For simplicity we assume that each coordinate direction has the same uniform grid spacing.)

We compute the Cartesian representations of the gradients, ∇f and $\nabla \chi$, so the components correspond to partial derivatives with respect to the coordinates. The partial derivatives are computed numerically using a stencil based on the genus two Daubechies wavelet connection coefficients [69] (which coincides exactly with the usual fourth order accurate central difference stencil). For example, the x-components are computed according to the formulas:

$$\frac{\partial f_{i,j,k}}{\partial x} = \frac{1}{h}\left[\frac{1}{12}f_{i-2,j,k} - \frac{2}{3}f_{i-1,j,k} + \frac{2}{3}f_{i+1,j,k} - \frac{1}{12}f_{i+2,j,k}\right] \tag{3.10}$$

$$\frac{\partial \chi_{i,j,k}}{\partial x} = \frac{1}{h}\left[\frac{1}{12}\chi_{i-2,j,k} - \frac{2}{3}\chi_{i-1,j,k} + \frac{2}{3}\chi_{i+1,j,k} - \frac{1}{12}\chi_{i+2,j,k}\right] \tag{3.11}$$

3.4 Implementation

Performing the quadrature specified in Eq. (3.5) leads to summing contributions from integrals over level sets with a weighting function accounting for the variable spacing between adjacent level sets; thus we find it both convenient and appropriate to refer to the contribution from an individual level set as a "shell". Figure 3.1 illustrates the essential concepts of implicitly defined domains, level sets, and shells. The shaded region indicates the domain of integration defined implicitly by $f(x,y) = \big((\frac{x}{5})^2 + (\frac{y}{4})^2 - 1\big)\big((x-1)^2 + (y+1)^2 - 1)\big) < 0$. The solid curves indicate level sets where $f(x,y) = \eta$ for $\eta \in [-14, -12, -10, -8, -6, -4, -2, 0]$, with spacing $d\eta = 2$. The darker shading indicates the shell bounded by the level sets associated with $\eta = -10$ and $\eta = -8$.

We created a parallel implementation of the grid-based coarea volume integral formulation in Python using the Numba package [50] that allows access to CUDA, a GPU-based parallel computing platform [9]. The integration scheme is implemented in two ways, one using a single kernel function to compute a fixed number of level sets and another that launches a kernel for a single level set allowing for adaptive refinement of the one-dimensional quadrature.

For the adaptive method a kernel function is launched for each shell to compute an

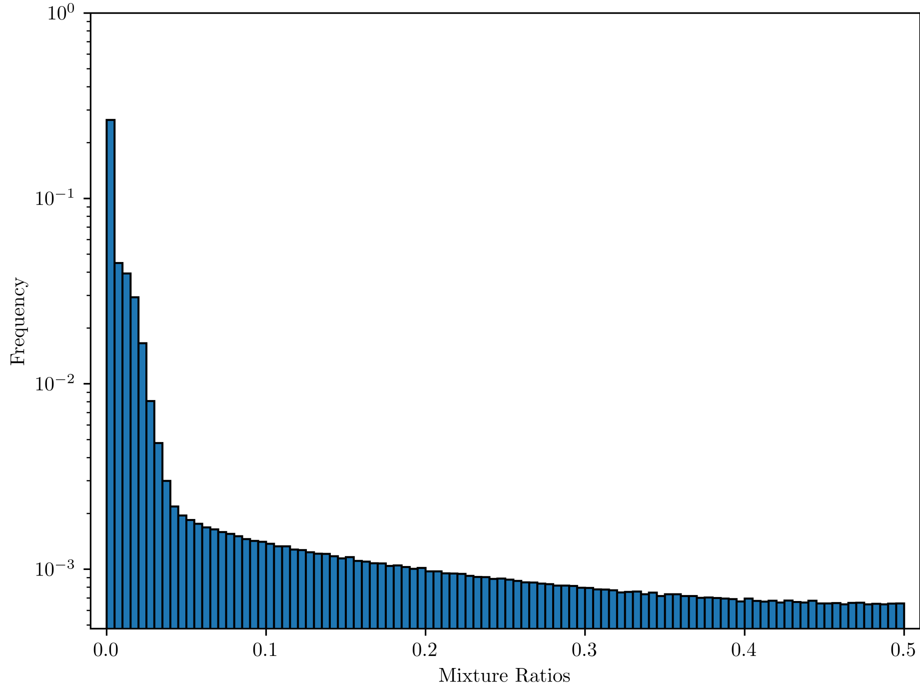


Figure 3.3: Histogram of the DNS data over the interval $[0, 0.5]$. Frequency corresponds to the fraction of grid points with whose f_{DNS} values lie in a bin of width 0.005.

array of voxel contributions according to Eqs. (3.9) and (3.10). Each voxel contribution is weighted by the corresponding coefficient in the quadrature formula and added into the voxel contribution array. For each shell kernel a reduction is performed to sum the entries in the voxel contribution array to produce the total level set contribution to the one-dimensional integral. The level set range is then subdivided and a higher resolution one-dimensional integral is computed. Based on the difference between the low and high resolution one-dimensional integrals, local subdivisions continue until either an error threshold is met or a maximum subdivision depth is reached.

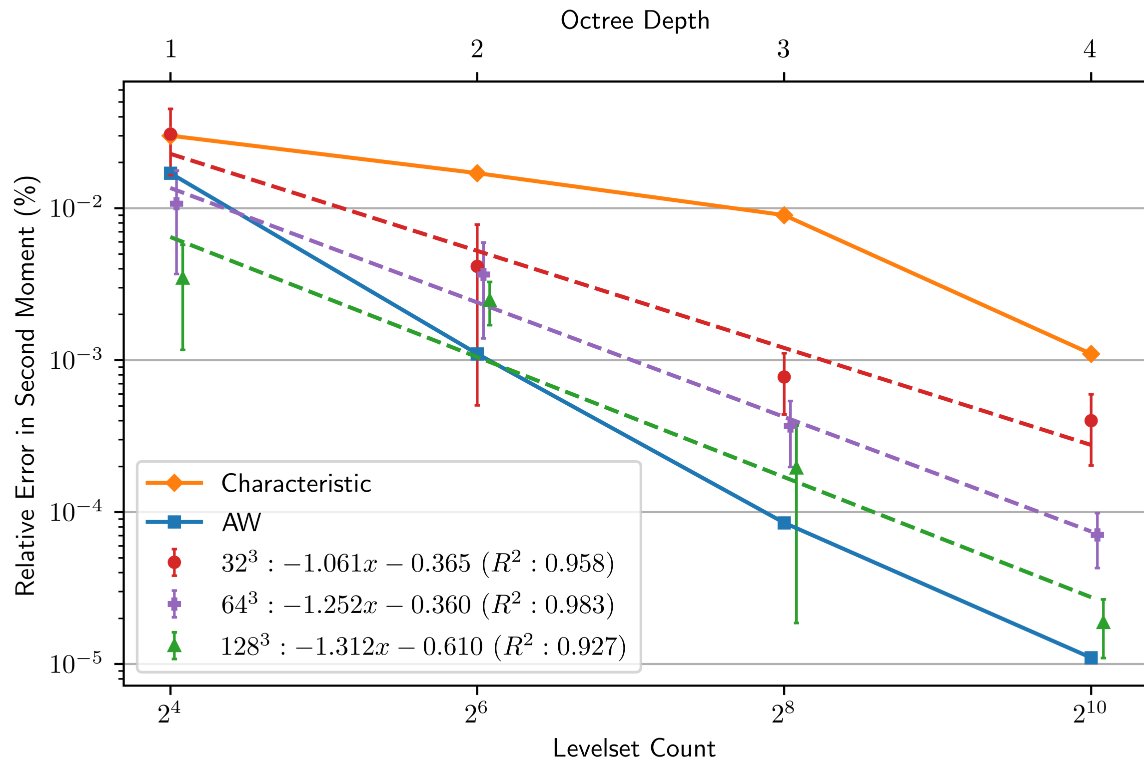


Figure 3.4: The L^2-norm of the residual error in integrating all trivariate polynomials of order up to three ($x^i y^j z^k$ with $i, j, k \geq 0$ and $i + j + k \leq 3$) over a unit sphere. The data for the solid lines is the error for the Characteristic and AW methods (presented in [84]) as a function of octree depth (labeled on the upper horizontal axis). The data for dashed lines is the error for the coarea method averaged over 10 jittered trials with grid size 32^3, 64^3, and 128^3 respectively as a function of level set count (labeled on the lower horizontal axis).

3.5 Results

We begin this section by applying the grid-based coarea integration method to produce results for comparison with examples presented in Thiagarajan and Shapiro [84]. We then proceed to a problems where analytical solutions are not known: an example due to Saye [74] with an open domain and logarithmic integrand and a data set produced from a direct numerical simulation of combustion [7]. Note that the effect of varying stencil size for finite difference evaluation of derivatives was treated in [91], and we proceed maintaining the conclusion that a standard central difference stencil of radius 2 is an appropriate choice. We use Simpson's quadrature rule for all one-dimensional integrals but note again that the coarea method can be employed with any one-dimensional quadrature method.

Thiagarajan and Shapiro presented and established the effectiveness of their adaptively weighted finite cell method (AW) [84] for multi-dimensional quadrature by choosing examples with known analytical results to support error comparison the characteristic function version of FCM presented by Abedian et al. [1]. In particular Thiagarajan and Shapiro considered integrals of polynomials over the unit sphere, and we start by applying the coarea method to these same problems.

Figure 3.4 illustrates convergence results for the grid-based coarea integration method along with results presented by Thiagarajan and Shapiro for the characteristic method and their AW method. For each method, we plot the L^2 norm of the residual error over all trivariate polynomials of order up to three ($x^i y^j z^k$ where $i, j, k \geq 0$ and $i + j + k \leq 3$) integrated over a unit sphere as a function of a refinement measure. For the characteristic and AW methods, the relevant refinement measure is the level of octree subdivision. A second labeling of the horizontal axis shows the number of level set values which is the relevant measure for the coarea method. Because of the dual labeling of the horizontal axis, attempts at one-to-one comparisons should be done with caution. However, several observations can be reliably made: (1) For a given grid refinement, the coarea method converges as the number of level sets increases corresponding to the usual refinement of a one-dimensional

quadrature; and (2) given a sufficiently refined grid of sample value, a number of level sets can be computed to achieve errors smaller than those of the characteristic method and comparable to the errors for the AW method.

It should also be noted that, despite its deterministic nature, the results in Fig. 3.4 for the coarea method are presented with error bars. This is because particular grid alignments produce fortuitously small errors [91]. In order to present more reliable error estimates, the data presented is the mean and standard deviation over 10 trials that are jittered by applying the coordinate transform $\{x, y, z\} \rightarrow \{x - x_c, y - y_c, z - z_c\}$ with $x_c, y_c, z_c \in [-h, h]$.

Thiagarajan and Shapiro chose a particular case, specifically $\int_\Omega x^2 dV$, to provide more detailed error information. We use this case to enable a more direct error comparison. Tables 1 and 2 give the AW error for an octree refinement level, followed by the coarea level set count needed to obtain a comparable error for a given level of grid refinement. The final column gives timing information for the coarea computation performed on a laptop equipped with an NVIDIA GeForce GTX 1650 Max-Q GPU.

Table 3.1 presents results based on a grid of 64^3 sample values. The GPU-parallel implementation of the coarea method produces results at interactive rates (with computation times less than the 16 ms refresh time for a 60 Hz display). At this level of sampling refinement, the errors produced by AW at octree depths 1, 2 and 3 can be achieved by the coarea method with an appropriate number of shells (2^1, 2^4, and 2^8 respectively). For such a coarse grid, the errors in the area integrals that define the shell contributions are too large to enable overall errors comparable to the 2.81×10^{-4} achieved by AW with octree depth 4.

The corresponding results for a 128^3 grid are shown in Table 3.2. At this level of sampling refinement, the error produced by AW with octree refinement depth 1, 2, 3, and 4 can be roughly matched by a coarea computation with shell count 2^1, 2^3, 2^7, and 2^{10} respectively. The more refined grids and improved accuracy cause some increase in computational costs, but the execution times remain below 100 ms.

We now move on to problems for which analytical solutions are not available. The first example, due to Saye, involves integrating over a domain defined as the region within the

Table 3.1: Percent error in computing $\int_\Omega x^2 dV$ for the AW method at varying octree depth levels, the number of level sets needed for corresponding error via coarea on a 64^3 grid, and timing for the coarea computation. Timings were computed using on a laptop equipped with an NVIDIA GeForce GTX 1650 Max-Q GPU.

	AW	Coarea	
Error (%)	Depth	Level sets	Timing (ms)
1.17e+0	1	2^1	8
6.78e-2	2	2^4	8
4.62e-3	3	2^8	15
2.81e-4	4	–	–

box $B = \{(x, y, z) \mid x, y \in [-4.25, 4.25],\ z \in [-2.125, 2.125]\}$ with the following implicit defining function and integrand:

$$f_{saye} = \cos(x)\sin(y) + \cos(y)\sin(z) + \cos(z)\sin(x) < 0 \tag{3.12}$$

$$g_{saye} = \ln((x^2 + y^2 + z^2)/4.25^2 + 3/8) \tag{3.13}$$

for which Saye provides a reference value $I_{saye} = \iiint_\Omega g_{saye}\ dV = 6.26192376...$. This problem provides a challenge even for the integration methods in commercial grade software. For example, we evaluated this integral using the *NIntegrate* method in Mathematica [36]. Treating the integral as an immersed boundary problem, i.e. integrating over the box, B, and restricting the support by including a factor of $\chi(f_{saye})$ in the integrand and applying an adaptive refinement method enables *NIntegrate* to obtain reasonable values for this integral.

The default settings for the global adaptive refinement method produce an error of about 1% in 5.8 seconds. Adjusting an optional parameter value (*MaxErrorIncreases*) can reduce the error at the cost of increased computing times. For example an error of 1.4×10^{-2}% was obtained in 8 seconds. Higher accuracy results were achievable (errors of 3.9×10^{-4}% and

Table 3.2: Percent error in computing $\int_\Omega x^2 dV$ for the AW method at varying octree depth levels, the number of level sets needed for corresponding error via coarea on a 128^3 grid, and timing for the coarea computation.

	AW	Coarea	
Error (%)	Depth	Level sets	Timing (ms)
1.17e+0	1	2^1	28
6.78e-2	2	2^3	28
4.62e-3	3	2^7	34
2.81e-4	4	2^{10}	89

$1.5 \times 10^{-5}\%$) but with computing times (35 seconds and 5.5 minutes respectively) that were far from the interactive range. Applying the coarea method to the problem, a result with $4 \times 10^{-2}\%$ error was obtained in 100 ms.

Next we present examples of volume integrals involving data from direct numerical simulation (DNS) of a fluid flow [70]. In this case, instead of being given the implicit function f_{DNS}, we are given values of f_{DNS} on a regular grid with 256 grid points along each axis of a bounding box corresponding to a unit cube. This is the sort of input data, grids of values rather than functions defined over a continuous domain, that motivates development of the method presented in this paper. The data set corresponds to a local mixture fraction so exact values are restricted to $f_{DNS} \in [0, 1]$.

The DNS is initialized with equal quantities of unmixed fluids so the starting configuration of the data consists largely of 0's and 1's with a median value of 0.5. The data set used occurs very early in the simulation so there are still large regions of unmixed fluid with values very close to 0 and 1, and the median remains close to 0.5. The stoichiometric surface, which corresponds to the $f_{DNS} = 0.35$ level set, is of special interest for combustion studies. The area of this level set was reported in [91]; here we expand to compute the volume of a stoichiometric region defined by mixture ratios $f_{DNS} \in [0.3, 0.4]$ as depicted in Fig. 3.2.

Applying the coarea method, we determined that this region occupies 1.47% of the volume of the simulation domain, and the computation took 150 ms. For comparison, tessellating a single level set using a standard implementation of marching cubes [86] takes 220 ms for this 256^3 data set.

Next we consider the volume integral over a wider range of mixture ratios $f_{DNS} \in [0, 0.5]$. The majority of mixture fraction values in this range are concentrated near 0 as illustrated by the histogram shown in Fig. 3.3. Due to the clustering of values near 0 there are regions where the function is very nearly flat. Ensuring that such flat regions are well sampled by the level sets requires a very small spacing between level set values, and very small level set value spacing across the entire range involves a very large number of level sets and unnecessary computational costs. The standard way of managing allocation of sampling resources is to employ locally adaptive refinement, so we now apply adaptive refinement of the level set values following Algorithm 1 in [26]. Using the adaptive scheme with maximum subdivision depth of 15, the coarea method took 2.0 seconds to compute that the region occupies 49.78% of the total simulation volume.

3.6 Discussion

We introduced a new grid-based coarea formulation for computing volume integrals over implicitly defined domains. The grid-based coarea volume integral formulation benefits from the solid mathematical foundation provided by Federer's coarea formula, the wavelet analysis convergence proof for the grid-based surface integral formulation, and known properties of standard methods for one-dimensional numerical quadrature.

To provide initial evidence for the effectiveness of the new grid-based coarea volume integral formulation, in particular we applied it to compute known examples presented by [84] involving integrals of polynomials over the unit sphere. The coarea results show reliable convergence, and increasing the grid refinement and quadrature shell count produces reliable error reduction. Comparison with results of the AW method presented by Thiagarajan and Shapiro [84] demonstrates that, given a sufficiently refined grid of input data and a sufficient

level set count, the coarea method can achieve comparable error results at interactive speeds.

Following the initial convergence study, we consider problems that lack analytical solutions: Saye's example and computation of volumes associated with specified ranges of mixture ratios produced by direct numerical simulation of fluid mixing. Saye's example provides analytic expressions for both the defining function, f, and the integrand, g; however it still provides challenges because the integrand is non-trivial and the defining function is a trigonometric polynomial which would define an open region without specification of a bounding box. Even using commercial grade software, it may take a significant amount of time to compute accurate results for this problem. Using the coarea method we computed the value of Saye's integral to an accuracy of $4 \times 10^{-2}\%$ in 100 ms.

The final examples involve integrals based on data from direct numerical simulation of fluid mixing. No analytic expression is given for the implicit defining function; instead the data consists of a grid of computed values of the mixture fraction, f_{DNS}. The grid-based coarea method is motivated precisely by this type of problem and succeeded in computing the volume of a stoichiometric region in 150 ms. We then proceeded to compute the volume for a mixture fraction range including 0 which is an accumulation value, so the f_{DNS} data set includes large regions where the function values are nearly constant. The coarea method can be successfully applied, and implementation of a standard adaptive scheme for quadrature refinement can provide significant reduction in computation time for this problem.

The grid-based coarea method presented here provides an alternative approach to computing volume integrals of well-behaved functions on implicitly-defined domains. (For links into the separate literature on integration of singular functions typically associated with potential theory, see the work of Greengard and collaborators [68, 52].) Like previously published grid-based methods [91], the grid-based coarea volume integral formulation presented in this paper applies directly in cases where only a grid of sampled values is provided for the geometric defining function and/or the integrand. Provided that the grids of sampled values are sufficiently refined, the coarea method can often achieve accuracy comparable to existing state-of-the-art methods for computing volume integrals. Perhaps most importantly,

the coarea method is also highly parallelizable and in many cases produces integral results at interactive rates.

BIBLIOGRAPHY

[1] ALIREZA Abedian et al. "PERFORMANCE OF DIFFERENT INTEGRATION SCHEMES IN FACING DISCONTINUITIES IN THE FINITE CELL METHOD". In: *International Journal of Computational Methods* 10.03 (2013), p. 1350002. DOI: 10.1142/S0219876213500023. eprint: https://doi.org/10.1142/S0219876213500023. URL: https://doi.org/10.1142/S0219876213500023.

[2] Mitko Aleksandrov, Sisi Zlatanova, and David J. Heslop. "Voxelisation Algorithms and Data Structures: A Review". In: *Sensors* 21.24 (2021). ISSN: 1424-8220. DOI: 10.3390/s21248241. URL: https://www.mdpi.com/1424-8220/21/24/8241.

[3] B. R. de Araújo et al. "A Survey on Implicit Surface Polygonization". In: *ACM Comput. Surv.* 47.4 (May 2015). ISSN: 0360-0300. DOI: 10.1145/2732197. URL: https://doi.org/10.1145/2732197.

[4] Christoph Bader et al. "Making data matter: Voxel printing for the digital fabrication of data across scales and domains". In: *Science Advances* 4.5 (2018). DOI: 10.1126/sciadv.aas8652. URL: https://www.science.org/doi/abs/10.1126/sciadv.aas8652.

[5] Johanna Beyer, Markus Hadwiger, and Hanspeter Pfister. "State-of-the-Art in GPU-Based Large-Scale Volume Visualization". In: *Comput. Graph. Forum* 34.8 (Dec. 2015), pp. 13–37. ISSN: 0167-7055. DOI: 10.1111/cgf.12605. URL: https://doi.org/10.1111/cgf.12605.

[6] Adrian Biran and Rubén López-Pulido. "Numerical Integration in Naval Architecture". In: Dec. 2014, pp. 77–96. ISBN: 9780080982878. DOI: 10.1016/B978-0-08-098287-8.00003-7.

[7] B. C. Blakeley et al. “On the kinematics of scalar iso-surfaces in turbulent flow”. In: *APS Meeting Abstracts*. Nov. 2017, L29.001, p. L29.001.

[8] Jules Bloomenthal and Brian Wyvill. “Introduction to Implicit Surfaces”. In: 1997.

[9] Shane Cook. *CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 9780124159334.

[10] Cyril Crassin et al. “GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering”. In: *I3D ’09*. 2009.

[11] Bas Dado et al. “Geometry and Attribute Compression for Voxel Scenes”. In: *Computer Graphics Forum* 35.2 (2016), pp. 397–407. DOI: `https://doi.org/10.1111/cgf.12841`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12841`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12841`.

[12] P.J. Davis, P. Rabinowitz, and W. Rheinbolt. *Methods of Numerical Integration*. Computer Science and Applied Mathematics. Elsevier Science, 2014. ISBN: 9781483264288. URL: `https://books.google.com/books?id=mbLiBQAAQBAJ`.

[13] A. De Cusatis, L.H. De Figueiredo, and M. Gattass. “Interval methods for ray casting implicit surfaces with affine arithmetic”. In: *XII Brazilian Symposium on Computer Graphics and Image Processing (Cat. No.PR00481)*. 1999, pp. 65–71. DOI: `10.1109/SIBGRA.1999.805711`.

[14] Dan Dolonius et al. “Compressing Color Data for Voxelized Surface Geometry”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.2 (2019), pp. 1270–1282. DOI: `10.1109/TVCG.2017.2741480`.

[15] Tom Duff. “Interval Arithmetic Recursive Subdivision for Implicit Functions and Constructive Solid Geometry”. In: *SIGGRAPH Comput. Graph.* 26.2 (July 1992), pp. 131–138. ISSN: 0097-8930. DOI: `10.1145/142920.134027`. URL: `https://doi.org/10.1145/142920.134027`.

[16] H. Engels. *Numerical quadrature and cubature*. Computational mathematics and applications. Academic Press, 1980. ISBN: 9780122388507. URL: https://books.google.com/books?id=vyvvAAAAMAAJ.

[17] Mark T Ensz, Duane W Storti, and Mark A Ganter. "Implicit methods for geometry creation". In: *International Journal of Computational Geometry & Applications* 8.05n06 (1998), pp. 509–536.

[18] Mark T. Ensz, Duane W. Storti, and Mark A. Ganter. "Implicit Methods for Geometry Creation". In: *International Journal of Computational Geometry & Applications* 08.05n06 (1998), pp. 509–536. DOI: 10.1142/S0218195998000266. eprint: https://doi.org/10.1142/S0218195998000266. URL: https://doi.org/10.1142/S0218195998000266.

[19] Luiz Henrique de Figueiredo and Jorge Stolfi. "Affine Arithmetic: Concepts and Applications". In: *Numerical Algorithms* 37 (2004), pp. 147–158.

[20] Oleg Fryazinov and Alexander Pasko. "GPU-based real time FRep ray casting". In: *GraphiCon 2007 - International Conference on Computer Graphics and Vision, Proceedings* (Jan. 2007).

[21] Eric Galin et al. "Segment Tracing Using Local Lipschitz Bounds". In: *Computer Graphics Forum* 39.2 (2020), pp. 545–554. DOI: https://doi.org/10.1111/cgf.13951. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13951. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13951.

[22] Ming Gao et al. "GPU Optimization of Material Point Methods". In: *ACM Trans. Graph.* 37.6 (Dec. 2018). ISSN: 0730-0301. DOI: 10.1145/3272127.3275044. URL: https://doi.org/10.1145/3272127.3275044.

[23] Jean-David Génevaux et al. "Terrain Modeling from Feature Primitives". In: *Computer Graphics Forum* 34 (May 2015). DOI: 10.1111/cgf.12530.

[24] John Geweke. "Monte carlo simulation and numerical integration". In: *Handbook of Computational Economics*. Ed. by H. M. Amman, D. A. Kendrick, and J. Rust. Vol. 1. Handbook of Computational Economics. Elsevier, Sept. 1996. Chap. 15, pp. 731–800. URL: https://ideas.repec.org/h/eee/hecchp/1-15.html.

[25] Enrico Gobbetti, Fabio Marton, and José Iglesias Guitián. "A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets". In: *The Visual Computer* 24 (July 2008), pp. 797–806. DOI: 10.1007/s00371-008-0261-9.

[26] Pedro Gonnet. "A Review of Error Estimation in Adaptive Quadrature". In: *ACM Comput. Surv.* 44.4 (Sept. 2012). ISSN: 0360-0300. DOI: 10.1145/2333112.2333117. URL: https://doi.org/10.1145/2333112.2333117.

[27] Erwin Groot and Brian Wyvill. "Rayskip: faster ray tracing of implicit surface animations". In: Jan. 2005, pp. 31–36. DOI: 10.1145/1101389.1101395.

[28] Markus Hadwiger et al. "Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces". In: *Computer Graphics Forum* (2005). ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2005.00855.x.

[29] Markus Hadwiger et al. *Real-Time Volume Graphics*. USA: A. K. Peters, Ltd., 2006. ISBN: 1568812663.

[30] Pat Hanrahan. "Ray Tracing Algebraic Surfaces". In: *SIGGRAPH Comput. Graph.* 17.3 (July 1983), pp. 83–90. ISSN: 0097-8930. DOI: 10.1145/964967.801136. URL: https://doi.org/10.1145/964967.801136.

[31] Mark Harris. *GPGPU: Beyond Graphics*. 2004. URL: http://download.nvidia.com/developer/presentations/GDC_2004/GDC2004_OpenGL_GPGPU_04.pdf.

[32] J. C. Hart, D. J. Sandin, and L. H. Kauffman. "Ray Tracing Deterministic 3-D Fractals". In: *SIGGRAPH Comput. Graph.* 23.3 (July 1989), pp. 289–296. ISSN: 0097-8930. DOI: 10.1145/74334.74363. URL: https://doi.org/10.1145/74334.74363.

[33] John Hart. “Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces”. In: *The Visual Computer* 12 (June 1995). DOI: 10.1007/s003710050084.

[34] M. Hazewinkel. *Encyclopaedia of Mathematics (1)*. Encyclopaedia of Mathematics: An Updated and Annotated Translation of the Soviet ”Mathematical Encyclopaedia”. Springer, 1987. ISBN: 9781556080005. URL: https://books.google.com/books?id=WzbzxytdAuUC.

[35] Rama Karl Hoetzlein. “GVDB: Raytracing Sparse Voxel Database Structures on the GPU”. In: *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. Ed. by Ulf Assarsson and Warren Hunt. The Eurographics Association, 2016. ISBN: 978-3-03868-008-6. DOI: 10.2312/hpg.20161197.

[36] Wolfram Research Inc. *Mathematica, Version 12.0*. Champaign, IL, 2019. URL: https://www.wolfram.com/mathematica.

[37] Tao Ju et al. “Dual Contouring of Hermite Data”. In: *ACM Trans. Graph.* 21.3 (July 2002), pp. 339–346. ISSN: 0730-0301. DOI: 10.1145/566654.566586. URL: https://doi.org/10.1145/566654.566586.

[38] Devendra Kalra and Alan H Barr. “Guaranteed ray intersections with implicit surfaces”. In: *ACM SIGGRAPH Computer Graphics* 23.3 (1989), pp. 297–306.

[39] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. “High Resolution Sparse Voxel DAGs”. In: *ACM Trans. Graph.* 32.4 (July 2013). ISSN: 0730-0301. DOI: 10.1145/2461912.2462024. URL: https://doi.org/10.1145/2461912.2462024.

[40] Yoshihiro Kanamori, Zoltan Szego, and Tomoyuki Nishita. “GPU-based Fast Ray Casting for a Large Number of Metaballs”. In: *Computer Graphics Forum* 27.2 (2008), pp. 351–360. DOI: https://doi.org/10.1111/j.1467-8659.2008.01132.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2008.

01132.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2008.01132.x.

[41] Matthew Keeter. *Libfive*. 2019. URL: https://libfive.com/.

[42] Matthew J. Keeter. “Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces”. In: *ACM Trans. Graph.* 39.4 (July 2020). ISSN: 0730-0301. DOI: 10.1145/3386569.3392429. URL: https://doi.org/10.1145/3386569.3392429.

[43] Benjamin Keinert et al. “Enhanced Sphere Tracing”. In: *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. Ed. by Andrea Giachetti. The Eurographics Association, 2014. ISBN: 978-3-905674-72-9. DOI: 10.2312/stag.20141233.

[44] Doyub Kim, Minjae Lee, and Ken Museth. *NeuralVDB: High-resolution Sparse Volume Representation using Hierarchical Neural Networks*. 2022. arXiv: 2208.04448 [cs.LG].

[45] A. Knoll et al. “Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic”. In: *Computer Graphics Forum* 28.1 (2009), pp. 26–40. DOI: https://doi.org/10.1111/j.1467-8659.2008.01189.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2008.01189.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2008.01189.x.

[46] Aaron Knoll, Ingo Wald, and Charles Hansen. “Coherent Multiresolution Isosurface Ray Tracing”. In: *The Visual Computer* 25 (July 2009), pp. 209–225. DOI: 10.1007/s00371-008-0215-2.

[47] Aaron Knoll et al. “Interactive Isosurface Ray Tracing of Large Octree Volumes”. In: *2006 IEEE Symposium on Interactive Ray Tracing*. 2006, pp. 115–124. DOI: 10.1109/RT.2006.280222.

[48] Aaron Knoll et al. “Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic”. In: *2007 IEEE Symposium on Interactive Ray Tracing*. 2007, pp. 11–18. DOI: 10.1109/RT.2007.4342585.

[49] Samuli Laine and Tero Karras. "Efficient Sparse Voxel Octrees". In: *IEEE Transactions on Visualization and Computer Graphics* 17.8 (2011), pp. 1048–1059. DOI: 10.1109/TVCG.2010.240.

[50] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A LLVM-based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: ACM, 2015, 7:1–7:6. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833162. URL: http://doi.acm.org/10.1145/2833157.2833162.

[51] Charles Loop and Jim Blinn. "Real-Time GPU Rendering of Piecewise Algebraic Surfaces". In: *ACM Trans. Graph.* 25.3 (July 2006), pp. 664–670. ISSN: 0730-0301. DOI: 10.1145/1141911.1141939. URL: https://doi.org/10.1145/1141911.1141939.

[52] Dhairya Malhotra and George Biros. "PVFMM: A Parallel Kernel Independent FMM for Particle and Volume Potentials". English (US). In: *Communications in Computational Physics* 18.3 (Jan. 2015), pp. 808–830. ISSN: 1815-2406. DOI: 10.4208/cicp.020215.150515sw.

[53] Josiah Manson and Scott Schaefer. "Isosurfaces Over Simplicial Partitions of Multiresolution Grids". In: *Computer Graphics Forum* 29.2 (2010), pp. 377–385. DOI: https://doi.org/10.1111/j.1467-8659.2009.01607.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01607.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01607.x.

[54] C. Maple. "Geometric design and space planning using the marching squares and marching cube algorithms". In: *2003 International Conference on Geometric Modeling and Graphics, 2003. Proceedings.* 2003, pp. 90–95. DOI: 10.1109/GMAG.2003.1219671.

[55] D. P. Mitchell. "Robust Ray Intersection with Interval Arithmetic". In: *Proceedings on Graphics Interface '90*. Halifax, Nova Scotia: Canadian Information Processing Society, 1990, pp. 68–74.

[56] F.C. Moon. *Applied dynamics: with applications to multibody and mechatronic systems.* Physics textbook. Wiley, 1998. ISBN: 9780471138280. URL: https://books.google.com/books?id=OUrvAAAAMAAJ.

[57] Ramon E Moore. *Interval analysis.* Vol. 4. Prentice-Hall Englewood Cliffs, 1966.

[58] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis.* USA: Society for Industrial and Applied Mathematics, 2009. ISBN: 0898716691.

[59] Ken Museth. "Hierarchical Digital Differential Analyzer for Efficient Ray-Marching in OpenVDB". In: *ACM SIGGRAPH 2014 Talks.* SIGGRAPH '14. Vancouver, Canada: Association for Computing Machinery, 2014. ISBN: 9781450329606. DOI: 10.1145/2614106.2614136. URL: https://doi.org/10.1145/2614106.2614136.

[60] Ken Museth. "NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation". In: *ACM SIGGRAPH 2021 Talks.* SIGGRAPH '21. Virtual Event, USA: Association for Computing Machinery, 2021. ISBN: 9781450383738. DOI: 10.1145/3450623.3464653. URL: https://doi.org/10.1145/3450623.3464653.

[61] Ken Museth. "VDB: High-Resolution Sparse Volumes with Dynamic Topology". In: *ACM Trans. Graph.* 32.3 (July 2013). ISSN: 0730-0301. DOI: 10.1145/2487228.2487235. URL: https://doi.org/10.1145/2487228.2487235.

[62] Tomoyuki Nishita and Eihachiro Nakamae. "A Method for Displaying Metaballs by using Bézier Clipping". In: *Computer Graphics Forum* 13.3 (1994), pp. 271–280. DOI: https://doi.org/10.1111/1467-8659.1330271. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.1330271. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1330271.

[63] Alexander Pasko et al. "Function representation in geometric modeling: concepts, implementation and applications". In: *The Visual Computer* 11 (Aug. 1995), pp. 429–446. DOI: 10.1007/BF02464333.

[64] Alexander Pasko et al. "Function representation in geometric modeling: concepts, implementation and applications". In: *The Visual Computer* 11.8 (1995), pp. 429–446.

[65] Alexander Pasko et al. "Procedural function-based modelling of volumetric microstructures". In: *Graphical Models* 73 (Sept. 2011), pp. 165–181. DOI: 10.1016/j.gmod.2011.03.001.

[66] K. Prashant and S. M. de Bruyn Kops. *Area of Scalar Isosurfaces in Isotropic Homogeneous Turbulence as a Function of Reynolds and Schmidt Numbers, J. Fluid Mech.*

[67] William H. Press et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* 3rd ed. New York, NY, USA: Cambridge University Press, 2007. ISBN: 0521880688.

[68] M. Rachh and L. Greengard. "Integral Equation Methods for Elastance and Mobility Problems in Two Dimensions". In: *SIAM Journal on Numerical Analysis* 54.5 (2016), pp. 2889–2909. DOI: 10.1137/15M103251X. eprint: https://doi.org/10.1137/15M103251X. URL: https://doi.org/10.1137/15M103251X.

[69] Howard L. Resnikoff and Raymond O. Wells Jr. *Wavelet Analysis: The Scalable Structure of Information.* Berlin, Heidelberg: Springer-Verlag, 1998. ISBN: 0-387-98383-X.

[70] James Riley. *On the kinematics of scalar iso-surfaces in a turbulent flow.* Author affiliation: University of Washington. May 2012. DOI: http://dx.doi.org/10.14288/1.0043050. URL: https://open.library.ubc.ca/cIRcle/collections/48630/items/1.0043050.

[71] D. Ruijters and Anna Vilanova. "Optimizing GPU Volume Rendering". In: *WSCG - Winter School of Computer Graphics.* Vol. 14. Feb. 2006, pp. 9–16. URL: http://graphics.tudelft.nl/Publications-new/2006/RV06.

[72] Andy Ruina and Rudra Pratap. *Introduction to statics and dynamics, E-Publishing Inc.* 2010.

[73] Juan Francisco Sanjuan-Estrada, Leocadio G. Casado, and Inmaculada García. “Reliable algorithms for ray intersection in computer graphics based on interval arithmetic”. In: *16th Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI 2003)* (2003), pp. 35–42.

[74] R. I. Saye. “High-Order Quadrature Methods for Implicitly Defined Surfaces and Volumes in Hyperrectangles”. In: *SIAM J. Scientific Computing* 37 (2015).

[75] C. E. Scheidegger et al. “Revisiting Histograms and Isosurface Statistics”. In: *IEEE Transactions on Visualization and Computer Graphics* 14.6 (Nov. 2008), pp. 1659–1666. ISSN: 1077-2626. DOI: `10.1109/TVCG.2008.160`.

[76] Rajsekhar Setaluri et al. “SPGrid: A Sparse Paged Grid Structure Applied to Adaptive Smoke Simulation”. In: *ACM Trans. Graph.* 33.6 (Nov. 2014). ISSN: 0730-0301. DOI: `10.1145/2661229.2661269`. URL: `https://doi.org/10.1145/2661229.2661269`.

[77] Dario Seyb et al. “Non-linear sphere tracing for rendering deformed signed distance fields”. In: *ACM Transactions on Graphics (TOG)* 38 (2019), pp. 1–12.

[78] Vadim Shapiro. “Real functions for representation of rigid solids”. In: *Computer Aided Geometric Design* 11.2 (1994), pp. 153–175.

[79] Vadim Shapiro. “Semi-analytic geometry with R-functions”. In: *ACTA numerica* 16 (2007), pp. 239–303.

[80] Andrei Sherstyuk. “Fast Ray Tracing of Implicit Surfaces”. In: *Computer Graphics Forum* 18.2 (1999), pp. 139–147. DOI: `https://doi.org/10.1111/1467-8659.00364`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.00364`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.00364`.

[81] Jag Mohan Singh and P. J. Narayanan. “Real-Time Ray Tracing of Implicit Surfaces on the GPU”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.2 (2010), pp. 261–272. DOI: `10.1109/TVCG.2009.41`.

[82] *Solid Modeling*. `http://solidmodeling.org/`. Accessed: 2022-05-13.

[83] Jorge Stolfi and Luiz Henrique De Figueiredo. *Self-Validated Numerical Methods and Applications*. 1997.

[84] Vaidyanathan Thiagarajan and Vadim Shapiro. "Adaptively Weighted Numerical Integration in the Finite Cell Method". In: *Computer Methods in Applied Mechanics and Engineering* 311 (2016), pp. 250–279. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2016.08.021. URL: http://www.sciencedirect.com/science/article/pii/S0045782516309823.

[85] Vaidyanathan Thiagarajan and Vadim Shapiro. "Adaptively weighted numerical integration over arbitrary domains". In: *Computers & Mathematics with Applications* 67.9 (2014), pp. 1682–1702. ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2014.03.001. URL: http://www.sciencedirect.com/science/article/pii/S0898122114000984.

[86] Stefan Van der Walt et al. "scikit-image: image processing in Python". In: *PeerJ* 2 (2014), e453.

[87] Alberto Jaspe Villanueva, Fabio Marton, and Enrico Gobetti. "Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes". In: *Journal of Computer Graphics Techniques (JCGT)* 6.2 (May 2017), pp. 1–30. ISSN: 2331-7418. URL: http://jcgt.org/published/0006/02/01/.

[88] Jarke J. van Wijk. "Ray Tracing Objects Defined by Sweeping Planar Cubic Splines". In: *ACM Trans. Graph.* 3.3 (July 1984), pp. 223–237. ISSN: 0730-0301. DOI: 10.1145/3870.3875. URL: https://doi.org/10.1145/3870.3875.

[89] Kui Wu et al. "Fast Fluid Simulations with Sparse Volumes on the GPU". In: *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2018)* 37.2 (2018), pp. 157–167. DOI: 10.1111/cgf.13350.

[90] G. Wyvill and A. Trotman. "Ray-Tracing Soft Objects". In: *Proceedings of the Eighth International Conference of the Computer Graphics Society on CG International '90:*

Computer Graphics around the World. CG International '90. Institute of Systems Science, Singapore: Springer-Verlag, 1990, pp. 469–476. ISBN: 0387700625.

[91] Mete Yurtoglu, Molly Carton, and Duane W. Storti. "Treat All Integrals as Volume Integrals: A Unified, Parallel, Grid-Based Method for Evaluation of Volume, Surface, and Path Integrals on Implicitly Defined Domains". In: *Journal of Computing and Information Science in Engineering* 18 (Mar. 2018). DOI: 10.1115/1.4039639.