Implicit modeling for additive manufacturing

Melike Aydinlilar

▶ **To cite this version:**

Melike Aydinlilar. Implicit modeling for additive manufacturing. Computer Science [cs]. Université de Lorraine, 2023. English. NNT : 2023LORR0336 . tel-04584706

HAL Id: tel-04584706
https://hal.univ-lorraine.fr/tel-04584706

Submitted on 23 May 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE LORRAINE | BIBLIOTHÈQUES UNIVERSITAIRES

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact bibliothèque : ddoc-theses-contact@univ-lorraine.fr
(Cette adresse ne permet pas de contacter les auteurs)

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Implicit modeling for additive manufacturing

THÈSE

15 Décembre 2023

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Melike AYDINLILAR

Composition du jury

Président :	Sylvain LAZARD	Université de Lorraine, Loria, Inria
Rapporteurs :	Julie DIGNE	LIRIS, CNRS
	Eric GALIN	LIRIS, Université Lyon 1
Examinateur :	Géraldine MORIN	IRIT, Université de Toulouse
Directeur de thèse :	Sylvain LEFEBVRE	Université de Lorraine, Loria, Inria
	Cédric ZANNI	Université de Lorraine, Loria, Inria

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Acknowledgments

First of all, I would like to thank my advisor Cédric Zanni, with whom we were able to discuss many ideas, build prototypes and overall who was always available for a discussion and clearing up the numerous questions and the preliminary ideas that I brought. I could not be more grateful for my co-advisor Sylvain Lefebvre who in the first place, modeled how to lead by example. I am grateful for the whole MFX team and Loria laboratory for their continuous support. I would also like to thank the committee members for their constructive feedback and insights.

All research works build on efforts of many that came before them who followed their curiosity and worked tirelessly over problems, not for the recognition or the monetary gains, but for the pursuit of knowledge. I am profoundly grateful for the spirit of open science and free dissemination of knowledge that transcends borders and time.

This thesis was supported by the ANR IMPRIMA(ANR-18-CE46-0004) project.

Le savant n'étudie pas la nature parce que cela est utile;
il l'étudie parce qu'il y prend plaisir et il y prend plaisir parce qu'elle est belle.

The scientist does not study nature because it is useful;
he studies it because he delights in it, and he delights in it because it is beautiful.
Henri Poincaré

Contents

List of Figures

Résumé

1 Définition du problème et motivation

La fabrication additive est une méthode de fabrication numérique contrôlée par programme, dans laquelle les objets sont construits couche par couche à l'aide de trajectoires de dépôt précises. Cette précision informatisée permet de répondre à de nombreuses exigences qui ne sont pas facilement satisfaites par les méthodes de production habituelles, qu'il s'agisse de prototypage rapide, de personnalisation ou de fabrication d'équipements scientifiques et médicaux hautement spécifiques. Plus important encore, grâce à la fabrication additive, il est désormais possible de produire des structures géométriques qui n'étaient pas réalisables auparavant. Cela permet d'étendre le domaine des objets pouvant être fabriqués, en faisant entrer les objets du monde numérique et mathématique dans le monde réel.

Que les objets à produire soient des modèles d'objets réels ou qu'ils soient générés de manière procédurale sous des contraintes données, ils doivent d'abord être représentés numériquement. Cette représentation est ensuite visualisée et manipulée au moyen d'une interface visuelle. Enfin, elle est convertie en une autre représentation réalisable par l'équipement de fabrication additive.

La représentation, la visualisation et la manipulation d'objets 3D ont été le principal sujet d'intérêt de l'infographie depuis ses origines. Toutefois, l'accent a été mis sur les représentations de surface, qui sont suffisantes pour la plupart des cas d'utilisation de la manipulation et de la visualisation numériques. Une forme est représentée par sa frontière qui sépare la forme solide du vide extérieur. Cependant, pour la fabrication additive, comme pour de nombreuses autres applications de l'infographie dans le monde réel, les représentations volumétriques ont de nombreux atouts par rapport aux représentations surfaciques.

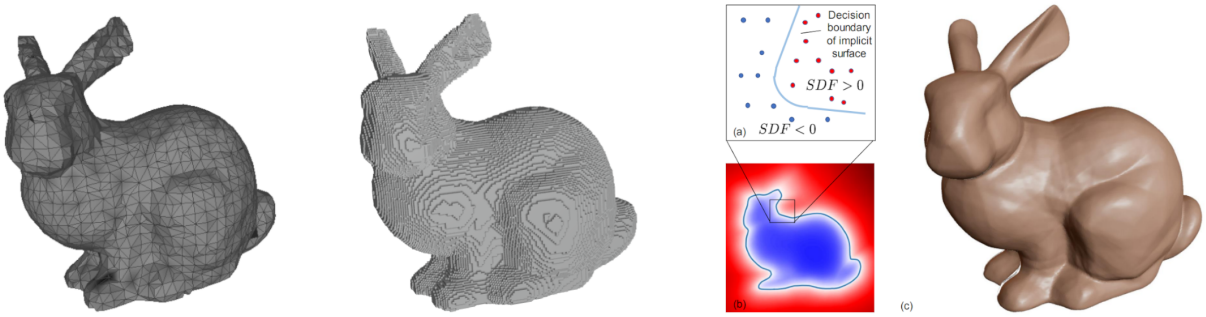


Figure 1: Maillage triangulaire, voxel et représentations neuronales implicites du lapin de Stanford [Sta] (figure de droite extraite de [PFS*19]).

L'objectif principal de cette thèse est la représentation implicite des surfaces pour la fabrication additive. Les représentations implicites sont définies par une fonction de champ et chaque point de l'espace est interrogé pour vérifier s'il se trouve à l'intérieur, à l'extérieur ou sur la surface définie par cette fonction [Blo97]. La définition de la fonction de champ donne une représentation compacte et indépendante de la résolution de la surface, ainsi que des requêtes volumétriques robustes. Cette définition volumétrique robuste permet d'appliquer directement des opérations de géométrie solide constructive (CSG) lors de la combinaison d'objets. Associées

à ces opérations, les définitions implicites de surface constituent un outil efficace pour modéliser une large gamme d'objets pouvant être réalisés par fabrication additive.

Bien qu'elles aient été introduites et aient gagné en popularité dans les années 1990, les représentations implicites ont récemment bénéficié d'un regain d'intérêt. Des représentations neuronales [PFS*19; CZ19; MON*19] aux outils de modélisation compacts [Mag21], les représentations implicites offrent une alternative utile aux représentations explicites en fournissant des représentations de formes lisses et continues pour des tâches spécifiques.

Le principal défi consiste alors à manipuler numériquement ces définitions de surfaces implicites. Notre contribution consiste à visualiser et à trancher (slicing en anglais) efficacement les surfaces implicites. Le tranchage est l'opération qui consiste à extraire les couches à traiter par l'équipement de fabrication additive c'est-à-dire l'imprimante 3D. Dans notre cas, nous utilisons le même cadre pour le rendu et le tranchage des objets [Lef13].

Contrairement aux définitions explicites, telles que les maillages triangulaires, les définitions implicites des surfaces ne donnent pas directement les positions de l'ensemble des points de l'espace définissant la surface. Chaque point de l'espace doit être interrogé pour pouvoir vérifier s'il se trouve sur la surface, à l'intérieur ou à l'extérieur de l'objet. Les définitions de surfaces implicites peuvent être visualisées directement ou en extrayant d'abord une représentation explicite intermédiaire. L'extraction de représentations intermédiaires telles que les maillages triangulaires ou les grilles de voxels est couramment utilisée pour le rendu en temps réel [LC87; JLSW02; CZ21] (Figure 1). Cependant, ces représentations intermédiaires dépendent de la résolution et nécessitent une mémoire supplémentaire pour des représentations de surface par ailleurs compactes. Pour les applications de modélisation en temps réel, une surface intermédiaire mise à jour doit être créée à chaque modification. Le lancer de rayons, quant à lui, fournit une méthode directe de rendu des iso-surfaces en trouvant les intersections entre les rayons de la caméra et la surface.

Le ray-tracing (lancer de rayons) [MS16] est une méthode fondamentale en infographie pour le rendu direct de formes 3D (Figure 2). La scène est rendue en intersectant les rayons de la caméra virtuelle avec les équations définissant les objets de la scène. Il s'agit d'un problème unidimensionnel de recherche de racines, qui est résolu numériquement dans la plupart des cas. Pour un rendu opaque, il suffit de trouver le premier point d'intersection entre les rayons de la caméra et l'équation de la surface. Pour le rendu transparent et le tranchage pour la fabrication additive, où les tranches planes qui seront fabriquées doivent être obtenues, toutes les intersections le long de chaque rayon doivent être enregistrées. Lorsqu'il n'est pas possible d'obtenir des solutions directes sous forme fermée, un point sur le rayon est déplacé à partir de l'origine du rayon ou du point d'intersection avec une boîte englobante, jusqu'à ce qu'il croise la surface. Cette méthode de lancer de rayons itérative avec une taille de pas constante s'est avérée utile avec des fonctions de bruit générées de manière procédurale [PH89]. Cependant, le lancer avec l'utilisation d'une taille de pas constante est inefficace et ne garantit pas de trouver toutes les intersections entre le rayon et la surface. Pour surmonter ces problèmes, le Sphere Tracing [Har96] a été introduit pour avoir une taille de pas adaptive qui garantie de ne pas traverser la surface.

Pour les définitions de champ qui consistent en un grand nombre de primitives de squelette, l'évaluation efficace des fonctions de champ est aussi importante que la recherche efficace des intersections de rayons. Dans le cas du lancer de rayons, seul un petit nombre de primitives intersectent un rayon donné. Les méthodes d'accélération spatiale pour l'élagage des primitives redondantes et de l'espace vide sont essentielles pour un rendu rapide.

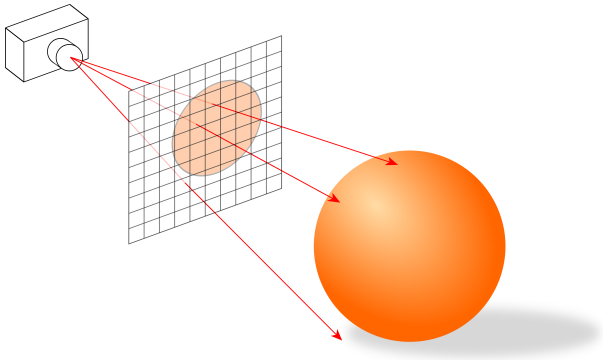


Figure 2: Le ray-tracing calcule l'intersection entre les rayons de la caméra et la surface de l'objet pour le rendu.

2 Les contributions

Dans cette thèse, nous présentons de nouvelles méthodes efficaces pour le rendu et le tranchage de surfaces implicites. Nous élargissons les méthodes existantes de lancer des rayons et d'intersection rayon-surface. Avec des implémentations GPU efficaces, nous visons à étendre l'utilisation des surfaces implicites dans les applications interactives.

Premièrement, nous présentons une nouvelle méthode de visualisation des surfaces intégrales basées sur des squelettes permettant une visualisation en temps réel. Les surfaces intégrales basées sur un squelette sont définies à l'aide de primitives de squelettes de points, de lignes ou de triangles. La surface implicite est ensuite définie en lissant la distance au squelette à l'aide d'un noyau décroissant. Le champ de densité résultant ne peut pas être visualisé directement. Pour visualiser cette surface implicite, les intersections rayon-surface doivent être calculées. Ce problème d'intersection n'a pas de solution de forme fermée et doit donc être résolu numériquement.

En s'appuyant sur une méthode récente d'inversion de champ [Bru19], nous extrayons une estimation de la distance à partir du champ de densité joint qui est exacte au niveau de la surface, par conséquent la résolution de l'intersection rayon-surface sur ce champ inversé donne une visualisation correcte de la surface sans erreurs d'approximation. Nous montrons qu'une interpolation quadratique au lieu d'une interpolation linéaire converge beaucoup plus rapidement. Par conséquent, l'interpolation sur l'estimation de la distance au carré permet un traitement efficace lorsqu'elle est combinée à des structures d'accélération spatiale sur GPU. Nous mettons en œuvre une subdivision des rayons pour l'interpolation afin de capturer la variation du champ inversé et de ne traiter que les primitives du squelette qui influencent un rayon donné. Nous montrons que cette interpolation quadratique itérative avec subdivision des rayons saute rapidement les zones vides et converge en quelques étapes d'interpolation. Par rapport au lancer de sphères [Har96] sur le champ inversé, l'interpolation quadratique pour la recherche de racines accélère le rendu et nous pouvons atteindre des taux interactifs pour des modèles complexes (Figure 3). Nous avons publié ce travail dans *Computer Graphics Forum* [AZ21].

Deuxièmement, en utilisant des méthodes numériques auto-validées, nous proposons une méthode de rendu transparent robuste pour les surfaces intégrales basées sur un squelette. Les méthodes numériques auto-validées, en particulier l'arithmétique d'intervalle [Mit90], fournissent des intersections de rayons robustes en bornant les valeurs de champ le long d'un rayon sur un intervalle donné. Pour réduire les bornes extrêmement étendues générées par l'arithmétique

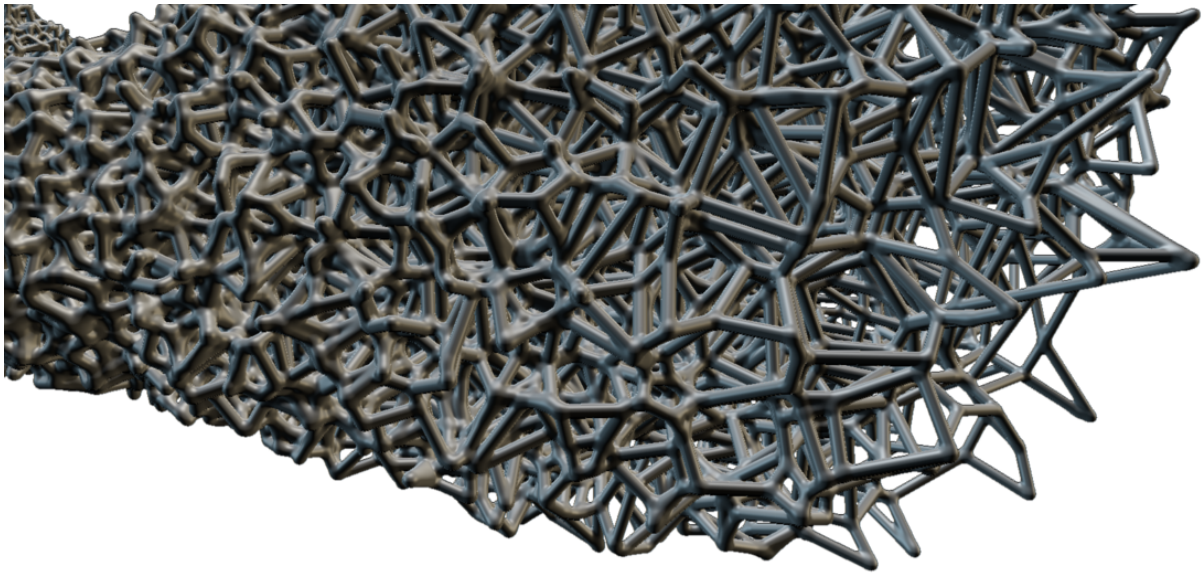


Figure 3: Gros plan sur la mousse procédurale rendue avec notre méthode [AZ21].

d'intervalle pour les expressions complexes, nous nous appuyons sur la connaissance de la monotonicité du champ le long du rayon. En utilisant les méthodes d'arithmétique d'intervalle pour limiter uniquement les opérations de mélange, nous améliorons les bornes calculées. Il en résulte un rendu transparent et un tranchage robustes et efficaces pour la fabrication additive. Ce travail a été publié dans *Eurographics Short Papers* [AZ22].

Figure 4: Surface intégrale basée sur un squelette et rendue transparente avec l'arithmétique d'intervalle par primitive [AZ22].

Enfin, en combinant les méthodes de lancer et les méthodes d'analyse de borne telles que l'arithmétique d'intervalle [Mit90] et l'arithmétique affine [dFS04], nous introduisons des bornes itératives linéaires et quadratiques asymétriques. Nous observons que la méthode Sphere Tracing [Har96] peut être interprétée comme un lancer avec une borne linéaire globale symétrique au point d'interrogation sur le rayon. Par conséquent, les méthodes de lancer qui utilisent une borne de Lipschitz locale [GGPP20] peuvent être étendues aux bornes linéaires asymétriques. Ces bornes linéaires asymétriques peuvent être calculées directement en trouvant les valeurs minimales et maximales de la dérivée de la fonction interrogée sur un intervalle donné. Cela contraste avec le calcul de la valeur maximale absolue de la dérivée lors du calcul des bornes

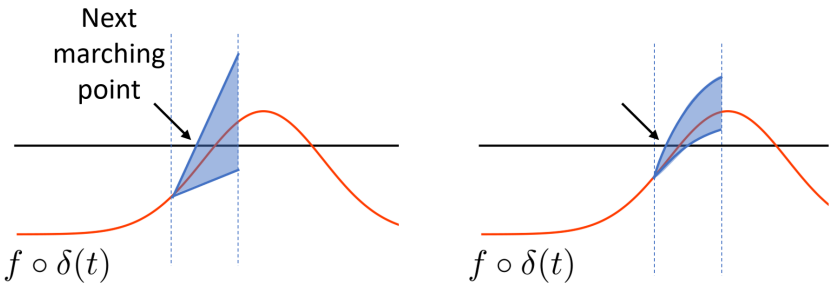


Figure 5: Fonctions d'inclusion à terme linéaires et quadratiques construites sur des fonctions de champ le long des rayons [AZ23].

de Lipschitz. Nous montrons que ces bornes linéaires peuvent être calculées de manière ascendante comme dans l'arithmétique d'intervalle et l'arithmétique affine [dFS04] en commençant par chaque opération simple constituant la fonction interrogée finale. Ces deux méthodes de calcul des bornes linéaires sont utilisées conjointement en fonction de la méthode qui fournit les bornes les plus étroites et de la facilité de calcul des dérivées d'ordre supérieur. Une fois que les bornes sont calculées pour la fonction interrogée le long du rayon sur un intervalle donné, le point sur le rayon est simplement déplacée en trouvant la racine suivante des bornes linéaires.

Nous avons ensuite étendu ces bornes linéaires asymétriques à des bornes quadratiques asymétriques. Les bornes quadratiques fournissent une meilleure approximation et une convergence plus rapide. Elles peuvent être calculées de la même manière que les bornes linéaires, soit en bornant les dérivées d'ordre supérieur, soit de manière ascendante en combinant les bornes quadratiques des opérations primitives. Elles sont ensuite déplacées sur le rayon en utilisant les racines des bornes quadratiques. Ces limites linéaires et quadratiques vers l'avant réunissent deux approches utilisées précédemment pour le traçage de surfaces implicites, à savoir les méthodes de Lipschitz et les méthodes numériques auto-validées. Ces bornes asymétriques sont robustes, flexibles et générales, car de nouvelles opérations arithmétiques peuvent être ajoutées aux opérations prises en charge. La méthode de lancer asymétrique conserve la propriété d'inclusion robuste des méthodes basées sur l'inclusion tout en gardant l'algorithme de recherche de racine simple et itératif. Ce travail a été publié à *Shape Modeling International (SMI) 2023* [AZ23].

Publications

Aydinlilar M., Zanni C.: Fast ray tracing of scale-invariant integral surfaces. *Computer Graphics Forum 40*, 6 (2021), 117–134. `https://doi.org/10.1111/cgf.14208`

Aydinlilar M., Zanni C.: Transparent Rendering and Slicing of Integral Surfaces Using Per-primitive Interval Arithmetic. *In Eurographics 2022 - Short Papers* (2022), The Eurographics Association. `https://doi.org/10.2312/egs.20221027`

Aydinlilar M., Zanni C.: Forward inclusion functions for ray-tracing implicit surfaces. *Computers & Graphics 114 (Special Section on SMI 2023)* (2023), 190–200. `https://doi.org/10.1016/j.cag.2023.05.026`

Abstract

Implicit surfaces provide many useful solutions for computer graphics tasks such as simple in and out queries, resolution independent representation and compact definition. However, rendering them robustly and efficiently provides a challenge especially for surfaces defined with complex field functions. In Part I, we introduce a real-time rendering method for skeleton-based integral surfaces. It relies on dynamically built A-buffers on GPU to discard empty spaces and reduce the number of skeleton primitives evaluation. The root finding is performed using rational quadratic interpolation to limit the number of field evaluations. Part II introduces a per-primitive interval arithmetic for skeleton-based integral surfaces for real-time rendering and slicing, and finally in Part III we introduce a family of robust forward inclusion methods for rendering a wide family of implicit surfaces. Using linear and quadratic inclusion functions, calculated either by bounding the first and second order derivatives, or building them up from the basic algebraic operations that constitute the field function definitions, ray-surface intersections are calculated reliably and efficiently. The problem of creating infinite or invalid bounds are eliminated by reducing the interval sizes and bounding piece-wise defined functions.

Example surfaces are given with skeleton-based implicits, convolution surfaces, Hermite radial basis implicits for real-time rendering and slicing for additive manufacturing.

Keywords: Additive Manufacturing, Implicit Modeling, Computer graphics.

Part I

Introduction

1 Introduction

1.1 Problem definition and motivation

Additive manufacturing is a programmatically controlled digital manufacturing method where objects are built layer by layer with precise deposition paths. This computerized precision provides a solution for many requirements that are not satisfied easily with regular production methods, from fast prototyping and customization to manufacturing highly specific scientific and medical equipment. Most importantly, using additive manufacturing, previously non-realizable geometric structures are now possible to produce. This gives a way to extend the domain of objects that can be manufactured, bringing the objects of the digital and mathematical world into the real world.

Whether the objects to be produced are models of real world objects, or generated procedurally under given constraints, they first need to be represented digitally. Then this representation would be visualized and manipulated through a visual interface. Finally, it would be converted to yet another representation that is realizable by the additive manufacturing equipment.

Representation, visualization, and manipulation of 3D objects have been the main topic of interest in Computer Graphics since its origins. However, the focus remained on the surface representations which is sufficient for most use cases for digital manipulation and visualization. A shape is represented by its boundary that separates the solid shape and the outside void. For additive manufacturing however, as many other real-world applications of Computer Graphics, volumetric representations are invaluable in addition to the surface representations.

The main focus of this thesis is implicit surface representations for additive manufacturing. Implicit representations are defined with a field function and each point in space is queried to test whether it lies inside, outside or on the surface [Blo97] defined by this function. The field function definition gives a compact, resolution independent surface representation and robust volume queries. This robust volumetric definition allows constructive solid geometry (CSG) operations to be applied directly when combining objects. Powered with these operations, implicit surface definitions provide an effective tool for modeling a wide range of objects that can be realized by additive manufacturing.

Even though they were first introduced and gained popularity in 1990's, recently, implicit representations have been enjoying a renewed interest. From neural representations [PFS*19;

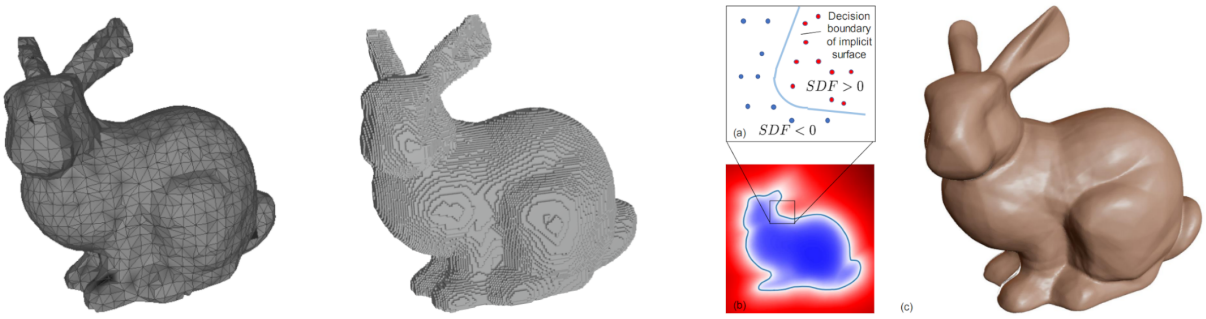


Figure 1.1: Triangle mesh, voxel and neural implicit representations of Stanford bunny [Sta] (rightmost figure taken from [PFS*19]).

CZ19; MON*19] to compact modeling tools [Mag21], implicit representations bring a useful alternative to explicit representations by providing smooth and continuous shape representations for specific tasks.

Then, the main challenge becomes digitally manipulating this implicit surface definitions. Our contributions lie in efficiently rendering and slicing implicit surfaces. Slicing is the operation of extracting layers to be processed by the additive manufacturing equipment/3D printer. In our case, we use the same framework for both rendering and slicing the objects [Lef13].

Unlike explicit definitions, such as triangle meshes, with implicit surface definitions, the positions of the set of points in space defining the surface are not given directly. Each point in space needs to be queried to be able to verify whether this point lies on the surface or is inside or outside of the object.

Implicit surface definitions can be visualized directly, or by first extracting an intermediate explicit representation. Extracting intermediate representations such as triangle meshes or voxel grids are commonly used for real-time rendering [LC87; JLSW02; CZ21] (Figure 1.1). However, these intermediate representations are resolution-dependent and require additional memory for otherwise compact surface representations. For real-time modeling applications, an updated intermediate surface needs to be created with each modification. Ray tracing, on the other hand, provides a direct method for rendering iso-surfaces by finding the intersections between camera rays and the surface.

Ray-tracing [MS16] is a fundamental method in Computer Graphics for directly rendering 3D shapes (Figure 1.2). The scene is rendered by intersecting the virtual camera rays with the equations defining the objects in the scene. This amounts to a one-dimensional root finding problem, which is solved numerically for most cases. Finding the first intersection point with the camera rays and the surface equation is enough for opaque rendering. For transparent rendering and slicing for additive manufacturing where the planar slices that will be manufactured need to be obtained, all intersections along each ray must to be registered. When direct solutions are not possible with closed form solutions, the ray is marched from the ray origin or the intersection point with a bounding box, until it intersects with the surface. This marching method with a constant marching step size has been shown to be useful with procedurally generated noise functions [PH89]. However, marching with a constant step size is inefficient and not guaranteed to find all the intersections between the ray and the surface. To overcome this problems, Sphere Tracing [Har96] have been introduced to have a guaranteed step size that would not cross the surface when marching.

For the field definitions that consist of a high number of skeleton primitives, efficient eval-

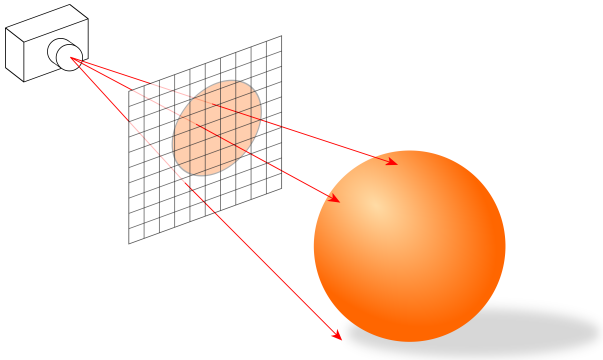


Figure 1.2: Ray-tracing calculates the intersection between camera rays and the object surface for rendering.

uation of the field functions are as important as efficiently finding the ray-intersections. In the case of ray-tracing, only a handful of the primitives intersect with any given ray. Spatial acceleration methods for pruning the redundant primitives and the empty space are essential for fast rendering.

1.2 Contributions

In this thesis, we present new efficient methods for rendering and slicing implicit surfaces. We extend the existing ray marching and related ray-surface intersection methods. With efficient GPU implementations, we aim to extend the usage of implicit surfaces in interactive applications and within the existing rendering frameworks.

Firstly, we introduce a novel method for visualization of skeleton based integral surfaces with real-time rendering rates. Skeleton based integral surfaces are defined with point, line or triangle skeleton primitives. Then the implicit surface is defined by smoothing the distance to the skeleton with a decreasing kernel. The resulting density field is given by an implicit definition, therefore is not rendered directly. To ray trace this implicit surface, the ray-surface intersections need to be calculated. This intersection problem has no closed-form solution, therefore it has to be solved numerically. Building on a recent field inversion method [Bru19], we extract a distance estimation from the blended density field that is exact at the surface, therefore solving the ray-surface intersection on this inverted field gives the correct surface visualization without approximation errors. We show that a quadratic interpolation instead of a linear interpolation converges much faster. Hence, interpolating on the squared distance estimate provides efficient processing when combined with spatial acceleration structures on GPU. We implement a ray-subdivision for the interpolation to capture the variation of the inverted field and only process the skeleton primitives that influence the given ray. We show that this iterative quadratic interpolation with the ray subdivision skips empty areas quickly and converge in a few interpolation steps. Compared to sphere tracing [Har96] on the inverted field, quadratic interpolation for root finding accelerates rendering and we can achieve interactive rates for complex models (Figure 1.3). We have published this work in *Computer Graphics Forum* [AZ21].

Secondly, using self validated numerical methods, we propose a robust transparent rendering method for skeleton based integral surfaces. Self validated numerical methods, in particular interval arithmetic [Mit90] provide robust ray intersections by bounding the field values along a

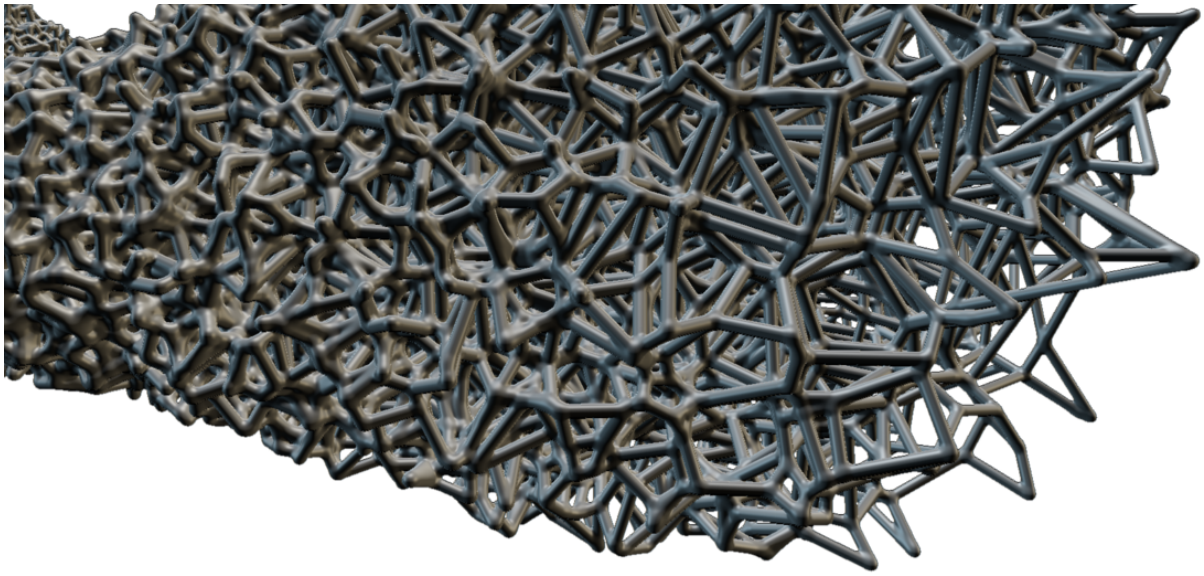


Figure 1.3: Close-up of the procedural foam rendered with our method [AZ21].

ray on a given interval. To reduce the prohibitively large bounds for complex calculations that interval arithmetic provides, we build on the knowledge of the field monotonicity along the ray. Using the interval arithmetic methods for bounding only the blending operations we improve the calculated ranges. This results in robust and efficient transparent rendering and slicing for additive manufacturing. This work has been published in *Eurographics Short Papers* [AZ22].

Figure 1.4: Skeleton-based integral surface rendered transparently with per-primitive Interval Arithmetic [AZ22].

And finally, combining marching and range analysis methods such as Interval Arithmetic [Mit90] and Affine Arithmetic [dFS04], we introduce asymmetric linear and quadratic marching bounds. We observe that, the Sphere Tracing [Har96] method can be interpreted as marching with a symmetric global linear bound at the query point on the ray. Therefore, the marching methods that use a local Lipschitz bound [GGPP20] can be extended to asymmetric linear bounds. This asymmetric linear bounds can be calculated directly by finding the minimum and maximum values of the derivative of the queried function on a given interval. This is in contrast with calculating the absolute maximum value of the derivative when calculating the Lipschitz bounds. We show that, these linear bounds can be calculated in a bottom-up fashion

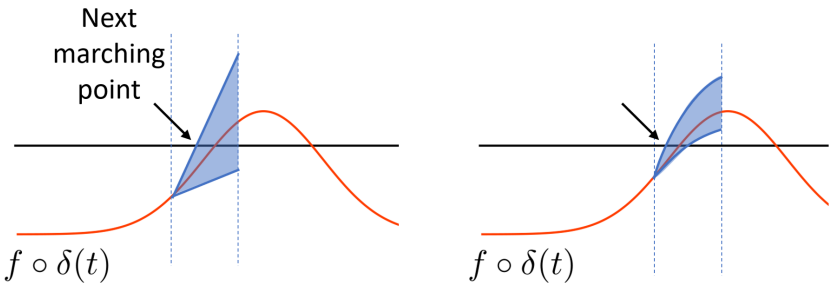


Figure 1.5: Linear and quadratic forward inclusion functions built on field functions along the rays [AZ23].

as in interval arithmetic and Affine Arithmetic [dFS04] starting from each simple operation constituting the final queried function. These two methods of calculating the linear bounds are used together depending on which method provides the tighter bounds and whether the higher order derivatives are easily calculated. Once the bounds are calculated for the queried function along the ray on a given interval, it is simply marched by finding the next root of the linear bounds.

We have then extended these asymmetric linear bounds to asymmetric quadratic bounds. Quadratic bounds provide better approximation and faster convergence. They can be calculated similarly to the linear bounds, either by bounding the higher order derivatives, or in a bottom-up fashion by combining the quadratic bounds of the primitive operations. They are then marched on the ray using the roots of the quadratic bounds. These linear and quadratic forward bounds bring together two approaches that were previously used for ray-tracing implicit surfaces, namely Lipschitz methods and self-validated numerical methods. These asymmetric bounds are robust, flexible and general as new arithmetic operations can be added to the supported operations as desired. The asymmetric marching method keeps the robust inclusion property of the inclusion based methods while keeping the root finding algorithm simple and iterative. This work has been published at *Shape Modeling International (SMI) 2023* [AZ23].

Publications

Aydinlilar M., Zanni C.: Fast ray tracing of scale-invariant integral surfaces. *Computer Graphics Forum 40*, 6 (2021), 117–134. `https://doi.org/10.1111/cgf.14208`

Aydinlilar M., Zanni C.: Transparent Rendering and Slicing of Integral Surfaces Using Per-primitive Interval Arithmetic. *In Eurographics 2022 - Short Papers* (2022), The Eurographics Association. `https://doi.org/10.2312/egs.20221027`

Aydinlilar M., Zanni C.: Forward inclusion functions for ray-tracing implicit surfaces. *Computers & Graphics 114 (Special Section on SMI 2023)* (2023), 190–200. `https://doi.org/10.1016/j.cag.2023.05.026`

Part II

Ray-Tracing Implicit Surfaces: State of the Art

A display connected to a digital computer gives us a chance to gain familiarity with concepts not realizable in the physical world. It is a looking glass into a mathematical wonderland.

Ivan E. Sutherland

1 Implicit Surfaces

Implicit surfaces represent smooth shapes with arbitrary topology and provide well-defined volumes for applications such as modeling, animation as well as additive manufacturing. This compact and resolution independent representation supports smooth blending of shapes by field compositions and provides direct inside-outside testing [Blo97].

Being a smooth and compact representation for complex shapes, implicit representations provide many useful applications for modeling and shape processing. They have been initially introduced in Computer Graphics for molecular rendering [Bli82] and extended to skeletal modeling [BS91]. They have been shown to be useful for soft contact modeling [Can93] and different primitives have been structured with different combination and deformation operations [WGG97].

Implicit surfaces have been further explored in several research areas such as producing microstructures for meta-material design for fabrication [PFV*11; MDL16], skinning for animation [VBG*13] and terrain modeling [PGGM09; PGP*19]. Recently, neural implicits [PFS*19; CZ19; MON*19] have been used for machine learning tasks as differentiable surface definitions are invaluable for optimization. Recent commercial software also uses the advantages of implicit surfaces in additive manufacturing such as nTopology [nTo15] and Cognitive Design Systems [Cog21], games such as Clay [Sec18] and Dreams [Med20] and modeling such as MagicaCSG [Mag21] and Womp [Wom22] as well as the popular procedural modeling sandbox platform Shadertoy [QJ13] (Figure 1.1).

Implicit surfaces are defined with a field function f and an iso-value c. The surface is defined where the function values are equal to the iso-value, and it robustly encloses a volume in 3D space, therefore providing straightforward inside-outside queries. Depending on the use case, there are different field function definitions. Signed distance fields (SDFs) are defined by the minimum distance to the surface at each point. They also encode information for further rendering tasks like ambient occlusion. In practice, SDFs can also be stored on a grid structure for fast field queries. While providing fast field queries for rendering, SDF grids are stored and later processed at a finite resolution which limit the capability of representing sharp edges. This also prevents the application of methods requiring the knowledge of the expression itself such as closed-form field gradient and range queries.

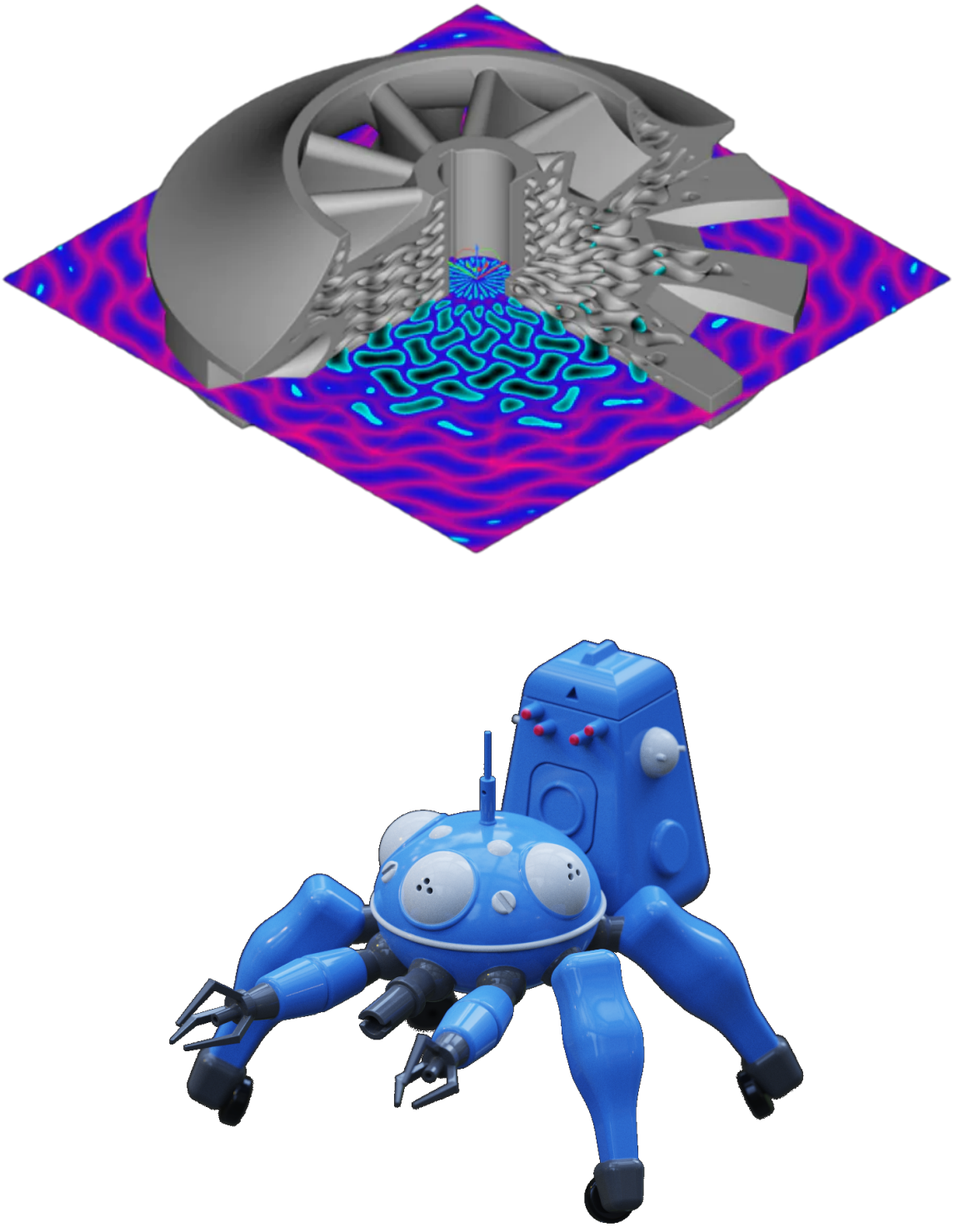


Figure 1.1: Implicit surface examples from nTopology [nTo15] showing the usage of microstructures for additive manufacturing (top) and a primitive-based artistic model built in MagicaCSG [Mag21](bottom).

With a point $\mathbf{p}$ in $\mathbb{R}^3$ and iso-value zero, the boundary surface is given by:

$$\left\{\mathbf{p} \in \mathbb{R}^3 \mid f(\mathbf{p}) = 0\right\} \tag{1.1}$$

and for the points on the surface, the normal is equal to the normalized field gradient.

There are several different conventions around the values that the field functions take. The functional representations (F-Rep) [PASS95] are usually close to a *signed-distance field* where a volume is defined by $f(\mathbf{p}) \leq 0$, and the field f tends toward infinity when moving away from the surface.

On the other hand *density fields* define metaball-like surfaces [Bli82; WMW86; NHK*85] with a volume defined by $f(\mathbf{p}) \geq c$ (usually with $c = 1/2$ or $c = 1$), where field values tend towards zero away from the object (Figure 1.2).

Different surfaces can be blended by combining their fields' contributions. Such combinations are often performed hierarchically in a Blobtree data structure [WGG99], where each node of the tree represents a composition operator and each leaf is an implicit primitive.

In this thesis we primarily focus on rendering and slicing skeleton-based surfaces with density fields. Therefore more detailed definitions will be given in the remainder of this chapter.

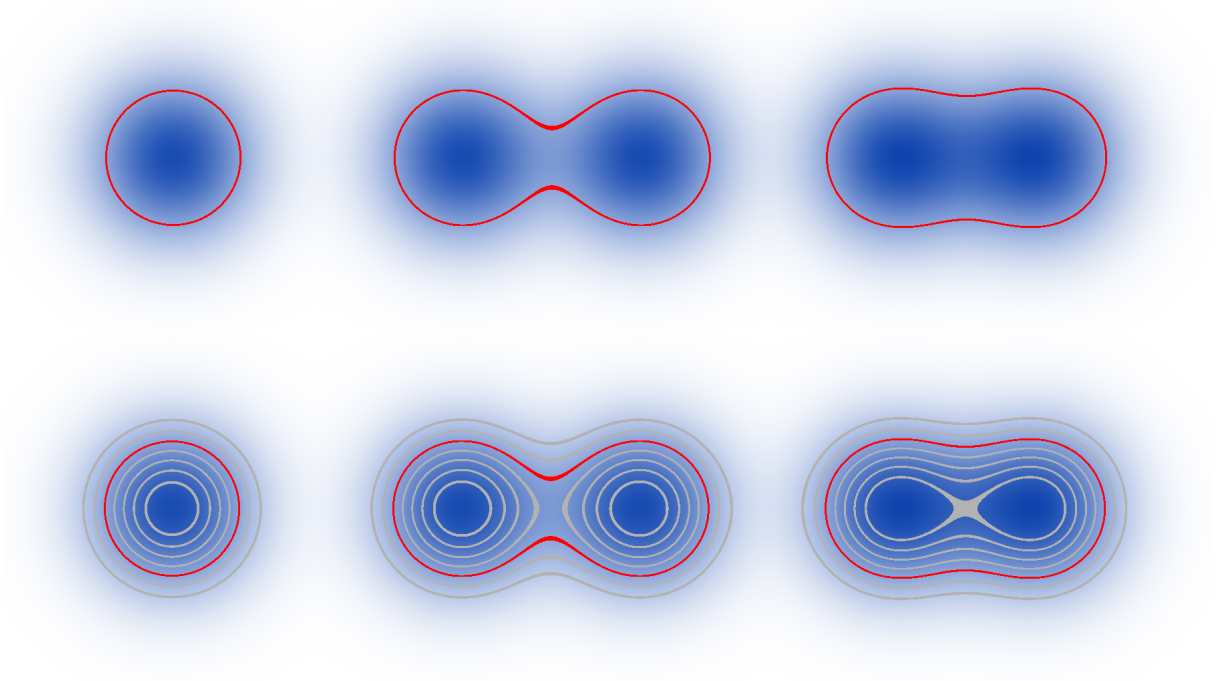


Figure 1.2: Density field for point primitives in 2D and the iso-lines with $c = 0.5$ in red.

1.1 Skeleton-Based Implicit Surfaces

The main family of implicit surfaces we study is skeleton-based implicit surfaces. Skeletons provide a natural way of handling the objects when defining and modeling implicit surfaces. Skeletons can be made of points, lines and even surfaces, we will start by explaining the earlier work done on points, then move to the more general form in Section 1.1.1. Point skeletons were

initially used in Computer Graphics for molecular rendering [Bli82]. These surfaces are defined by point primitives and the distances to the points are smoothed with a decreasing Gaussian kernel, building a density field. Several neighboring density fields then can be combined by summing the field values and create a smooth blended surface.

With a Gaussian kernel and S as the set of point centers:

$$f(S, \mathbf{p}) = \sum_{s \in S} \exp(-\|\mathbf{s} - \mathbf{p}\|^2) \tag{1.2}$$

gives the definition of these surfaces which can be further parameterized for the sphere radius and desired blobbiness factor.

With this definition, a single point generates a sphere. When multiple points gets close to each other, the summed density field creates a smooth blended surface. This smooth blended surface provides a convenient modeling tool. Since the density field is defined also in the inside of the surface, it gives a robust volume definition, therefore constructive solid geometry (CSG) operations can be applied directly on the combined field values. This further improves the modeling capabilities.

This density field formulation has been extended to polynomial field functions [WMW86; NHK*85] for simpler computation and field evaluation localization in space.

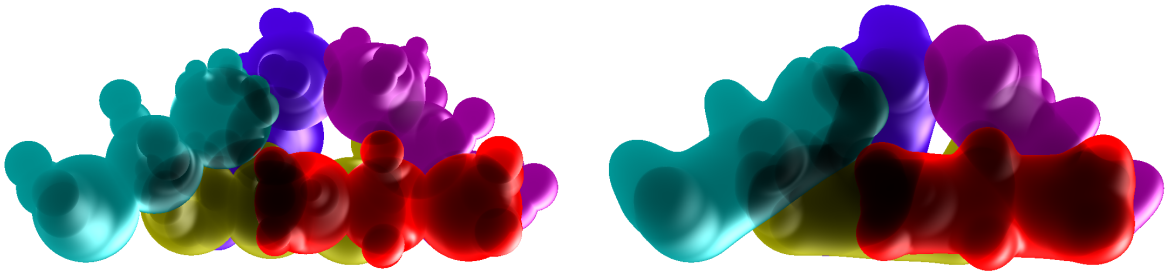


Figure 1.3: *Left:* Point primitives with radius information. *Right:* Blended Surface with addition and union operators.

1.1.1 Convolution Surfaces

Point primitives can be generalized into higher dimensional skeletons such as line segments and triangles. Distance surfaces are isosurfaces of the function value of d with a point p and skeleton S:

$$d(\mathbf{S}, \mathbf{p}) = \min_{s \in S} \|\mathbf{s} - \mathbf{p}\| \tag{1.3}$$

And with the smoothing kernel k:

$$f(\mathbf{S}, \mathbf{p}) = k\left(\min_{s \in S} \|\mathbf{s} - \mathbf{p}\|\right) \tag{1.4}$$

Direct summation of distance surfaces creates bulges where sub-parts of the skeletons overlap (Figure 1.4). To fix this problem, convolution surfaces have been introduced [BS91] where the skeleton can be considered as infinite number of point primitives.

Figure 1.4: Skeleton sub-division creates bulging effect at the junctions. (Image from [ZBQC13])

Given a skeleton S, and a point $\mathbf{p}$ in space and k as the exponential function, their contribution can be written as in Equation 1.2 or, with integration as:

$$f(S, \mathbf{p}) = \int_S \exp\left(-||s - \mathbf{p}||^2\right) \tag{1.5}$$

Closed-form solutions have been provided [MS98] with polynomial kernels. There are several kernel families with compact support or different polynomial representations [She99b]. A brief listing is given below. Further closed form solutions for convolution surfaces with varying radius and weighted skeletons for different kernels have been extensively studied [JTFP01; JT02b; JT02a; HC12; Hub12] for line segment skeletons with varying radius and triangle skeletons.

Kernels *Gaussian Kernel*: Given the distance d to the skeleton, and a radius r, Gaussian kernels [Bli82] are given in the form:

$$k_s(d) = \frac{1}{N} e^{-s\left(\frac{d}{r}\right)^2} \tag{1.6}$$

where s denote the kernel scale and N is a constant normalizing factor for achieving the radius r.

The main kernel families used to define skeleton-based implicit surfaces are Cauchy, Inverse and Compact Polynomial kernels. They form families of kernels parametrized by a degree parameter n, defining the smoothness of the surface for the compact polynomial kernel, and a scale parameter σ, defining the extent of blends during composition.
Compact Polynomial kernels [WGG99] are defined as:

$$k_{n,\sigma}(d) = \begin{cases} \left(1 - \left(\frac{d}{\sigma}\right)^2\right)^{\frac{n}{2}} & if\ d < \sigma\ , \\ 0 & otherwise. \end{cases} \tag{1.7}$$

The *Cauchy* kernels [MS98] are defined as:

$$k_{n,\sigma}(d) = \frac{1}{\left(1 + \left(\frac{d}{\sigma}\right)^2\right)^{\frac{n}{2}}}. \tag{1.8}$$

And the Inverse kernel [HAC03; JT02a]:

$$k_{n,\sigma}(d) = \frac{1}{\left(\frac{d}{\sigma}\right)^n}. \tag{1.9}$$

We mostly focus on the *Compact Polynomial* kernel due to its desirable properties: local support – required for efficient field evaluation on large skeletons – and efficient closed form evaluation of Equation (1.14) for line segment primitives (with linearly interpolated prescribed

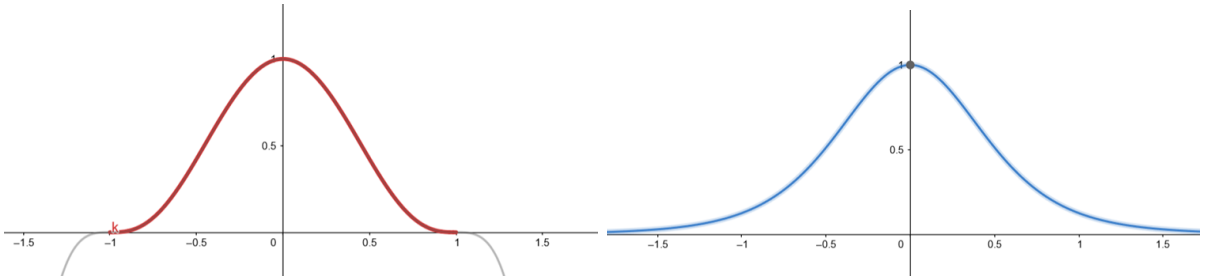


Figure 1.5: *Left:* Compact Polynomial kernel. *Right:* Cauchy kernel.

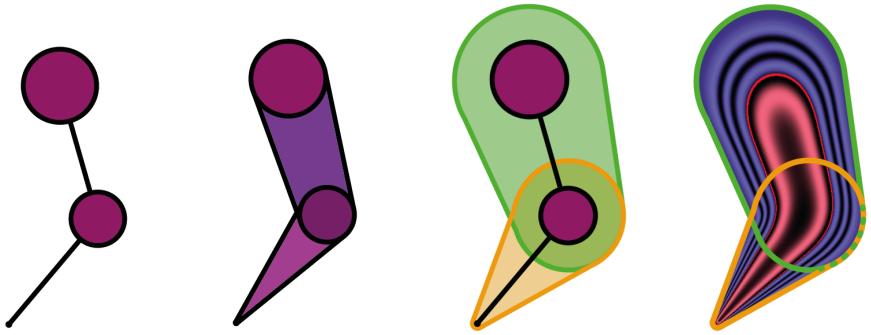


Figure 1.6: *Leftmost:* line segment primitives with prescribed radius at vertices. *Left:* associated sphere-cones. *Right:* Primitive supports when using a compact kernel. The supports are also sphere-cones and their radii are proportional to the radii τ_{S_i} prescribed on the skeleton. *Rightmost:* a slice of the SCALIS scalar field, the points that are inside of the volume defined by $f(\mathbf{p}) > c$ are depicted in red, the points outside are depicted in blue. The points outside of the supports of all primitives have a null field value and are depicted in white.

thickness τ). In this case, the support of an individual primitive (e.g. the volume for which the field value is not zero) is also a sphere-cone that is defined by scaling the two primitive endpoints' radius by a factor σ (see Figure 1.6 which also describes the convention for the display of the fields).

Scale Invariant Integral Surfaces

Chapters 1 and 2 focus on rendering Scale-Invariant Integral Surfaces (SCALIS) [ZBQC13]. This surface definition builds on the previous convolution surface definition [BS91] and extends it for simpler radius control and preserving sharp details as well as controlling the local blending behaviour.

A skeleton is defined as a set of line segment primitives S_i with prescribed radius information at the endpoints (or vertices - see Figure 1.7, left). For each point $\mathbf{q}$ on the line segment, a radius $\tau_{S_i}(\mathbf{q})$ can be defined as the linear interpolation of the radii of the two endpoints. Hence, for each such line segment primitive S_i, a *sphere-cone* can be defined as the infinite union of balls defined by the skeleton points $\mathbf{q}$ belonging to the segment with associated radius $\tau_{S_i}(\mathbf{q})$ (see Figure 1.7, middle).

Line segment primitives are defined in terms of distance between a point $\mathbf{p}$ in space to a

Figure 1.7: *Left:* a skeleton with prescribed radius on its vertices, *Middle:* infinite union of spheres defined by linear interpolation of prescribed radius along skeleton edges (union of sphere-cones), *Right:* resulting scale-invariant integral surface which can be seen as a smoothing of the sphere-cone union.

point $\mathbf{q}$ on the line segment:

$$d(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\| \tag{1.10}$$

which, divided with the radius $\tau_{S_i}(\mathbf{q})$ at the skeleton point $\mathbf{q}$, defines the *homothetic distance* to a single skeleton point:

$$h(\mathbf{p},\mathbf{q}) = \frac{d(\mathbf{p},\mathbf{q})}{\tau_{S_i}(\mathbf{q})} \tag{1.11}$$

or to line segment primitives :

$$h_{S_i}(\mathbf{p}) = \min_{\mathbf{q}\in S_i} h(\mathbf{p},\mathbf{q}) \tag{1.12}$$

Given a kernel k, this allows to define a density function for the line segment S_i:

$$f_i(\mathbf{p}) = \max_{\mathbf{q}\in S_i} k \circ h_{S_i}(\mathbf{p},\mathbf{q}) \tag{1.13}$$

with $\circ$, the composition operator.

The SCALIS field is defined as:

$$f_i(\mathbf{p}) = \frac{1}{N_{k,c}} \int_{S_i} \frac{k \circ h(\mathbf{p},\mathbf{q})}{\tau_{S_i}(\mathbf{q})}\, \mathrm{d}\mathbf{q} \tag{1.14}$$

where the normalization factor $N_{k,c}$, which depends on the chosen isovalue c and kernel k, is used in order to achieve the prescribed radius around the skeleton.

Due to the additive property of the integral, similar to convolution surfaces [BS91], SCALIS provides an independence from skeleton subdivision when using blending by summation. Visually, the resulting isosurface appears as a smoothing of the sphere-cones associated to the prescribed radii (see Figure 1.7, right).

In addition to a direct radius control, this formulation simplifies modeling by providing scale-invariance properties: scaling both the skeleton geometry and the associated radii with a factor s results in a scaling of the isosurface of interest by the same factor, e.g. blending behavior is independent from the scale. This scale invariance property also reduces the blurring of details when blended in larger shapes.

1.2 Further implicit function definitions

There is a vast literature dedicated to the development of further primitives and operators for implicit modeling applications. Blobtrees [WGG97] provide a way to process primitives and operations on them in a tree structure for both density fields and SDFs. Moreover, when using distance functions, smoothed versions of CSG operations (e.g. union, intersection and difference operations) [DOG04] are commonly used. Gradient based blending [GBC*13] prevents smoothing of details and unwanted blending at distance. Deformations such as warps, twists, tapers etc. [WO97; WGG97] provides further control over the implicit shapes. Sweeping volumes [SAJ21] creates implicit shapes by sweeping over paths. Procedural hypertextures [PH89] and microstructures [MDL16] allow creating complex structures on the fly. In Chapter 3 we show our additional results on Hermite Radial Basis Functions Implicits (HRBF's) [Wen05; MGV11] which are defined with point in space and their normal information.

2
Rendering Implicit Surfaces

One of the main difficulties while working with implicit surfaces is providing an efficient visualization. During modeling and final rendering, visualization may have different requirements in terms of time and precision. For interactive modeling it is important to have a method that requires no pre-computations and no transformations of the native representation. In this thesis, we have studied direct and efficient rendering for implicit surfaces.

2.1 Meshing and Voxelization

In practice implicit surfaces are mainly rendered by either meshing or voxelization. Efficient GPU implementations have been suggested for polygonization with the challenges of sharp edge reconstruction, a guarantee on water-tightness and adaptive meshing. For voxelization, the main challenge is the cubic growth of the number of voxels as the resolution is increased, becomes rapidly unmanageable.

For real-time rendering, extracting explicit representations are commonly used. From marching cubes [LC87] and dual contouring [JLSW02] and their more recent neural counterparts [CZ21; CTFZ22] to voxelization [CNLE09] this problem is still an active research topic [SMH*23]. An overview of the polygonization methods are given in the survey [dALJ*15].

2.2 Ray Tracing Implicit Surfaces

Ray tracing is a direct method for rendering iso-surfaces by finding the intersections between camera rays and the surface. To ray trace an implicit surface, the ray equation with origin $\mathbf{o}$ and direction $\mathbf{u}$:

$$\delta(t) = \mathbf{o} + t\mathbf{u} \tag{2.1}$$

is plugged into the surface equation (1.1), resulting in a one-dimensional root finding problem:

$$f \circ \delta(t) = 0 \tag{2.2}$$

This way, ray tracing implicit surfaces comes down to solving the one-dimensional root-finding equation (2.2) for the ray parameter t. For most field functions, closed-form solutions

are not available. Several numerical methods have been suggested for solving this ray-surface intersection equation. They differ in their robustness, performance, computational complexity, and universality.

In earlier methods, Blinn [Bli82] have suggested using *regula falsi* with Newton root refinement for the blobby molecular models (1). Later, robust root isolation has been achieved by using Lipschitz constants on first and second derivatives to drive the bisection methods [KB89]. This way, it was guaranteed to not to lose the small details.

The number of field evaluations performed along the rays is usually the driving complexity factor with typically challenging areas such as grazing rays. In addition, transparent rendering amplifies numerical robustness and complexity issues.

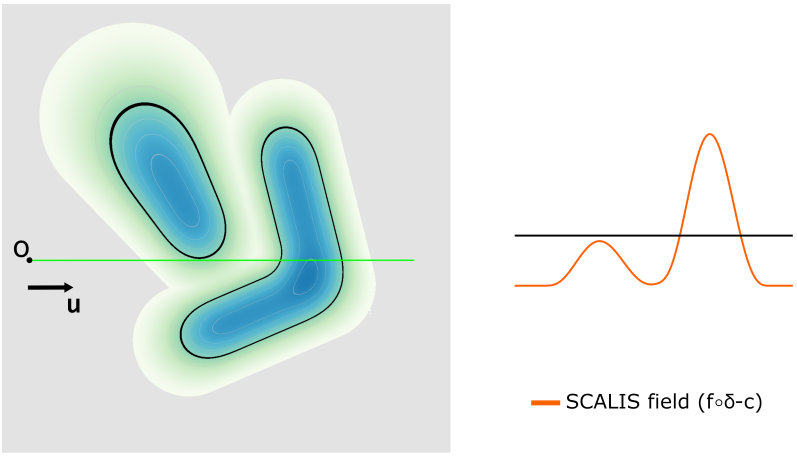


Figure 2.1: Field values for SCALIS field with compact kernel along a given ray.

2.2.1 Polynomial Approximation

Several ray tracing algorithms rely on polynomial root finding algorithms to locate the iso-surface along the ray, either because the field itself is defined by polynomials or because a polynomial approximation of the field is built along the ray.

For point primitives defined with the compact polynomial kernel (Equation (1.7)), the field function along the ray is a piece-wise polynomial. For small degree (up to degree 4), this property can be used to compute a closed-form expression of the root [GPP*10]. For higher degree polynomials, Laguerre's method [WT90] and Bézier clipping [NN94; BJ07; LZLW09] have been used for iteratively converging toward the first root of the polynomial or to rapidly reject the root-free intervals. Bézier clipping uses the inclusion property of the Bézier control polygons to iteratively converge towards the first root and discard grazing rays more rapidly. While using higher degree approximations is possible, root computations are more expensive and can easily get numerically unstable [LZLW09].

For convolution surfaces, polynomial approximations of the field value can be built along the ray [She99a]. During rendering, each primitive can be approximated by one or several polynomials on the interval defined by the intersection between the ray and the primitive support. The interval is uniformly subdivided and polynomial approximations are calculated from Hermite data (field value and derivative) sampled at sub interval boundaries (see Figure 1.2). The roots of resulting cubic polynomial approximations can be computed analytically. When primitives

are combined with the summation operator, new polynomials are defined by summation of polynomials on the intersection of primitive sub intervals.

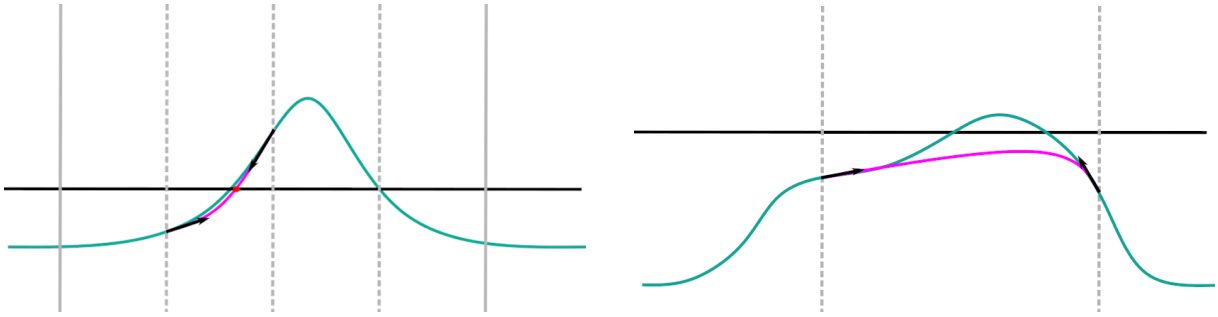


Figure 2.2: Polynomial approximation can introduce approximation error or discard roots.

Choosing an adequate level of subdivision is problematic: on one hand, small number of intervals can result in poor approximation depending on both the viewpoint and the skeleton tessellation even more so in the context of SCALIS as it becomes more difficult to capture the maximal field contribution of a given primitive. On the other hand, higher number of intervals results in higher computational time due to additional field evaluations. Furthermore, when performing computation on GPU, the incoherence of the generated subintervals require additional data management to avoid the computation of the additional field values.

An alternative to interval subdivision is to rely on a higher degree approximation defined from additional sampling positions [She99a; JTZ09]. However, such an approach prevents the use of closed form expressions and can also be subject to increased fitting error due to introduction of oscillations in a higher degree polynomial approximation due to Runge's phenomenon.

For arbitrary field definition, Taylor polynomial approximations were also used to perform root isolation [SWR18]. Adaptive intervals are defined along the ray using a gradient based heuristics, each interval is analyzed using two Taylor approximations of degree two defined from both end-points of the interval, intervals are subdivided until the Taylor approximations find a single root. A bounded Newton method is then used on the resulting interval. Such an approach requires the computation of a higher order derivative and a well chosen initial step-size (either user defined or based on global property of the Hessian) in order for the root isolation to be guaranteed.

2.2.2 Self-Validated Numerical Methods

Interval arithmetic was introduced by Moore [Moo66] in the 1960s to overcome measurement errors in scientific computing. With interval arithmetic, every arithmetic operation is replaced with its interval equivalent. Hence, given an input interval, an output interval including all the possible function values can be constructed by simply evaluating the expression.

For an arbitrary function-based field definition, relying on interval arithmetic is a robust approach for ray tracing [Mit90; CHMS00; Kee20]. Interval arithmetic computes a bound on the field variations on a given interval and each time the estimated bound contains the zero iso-value the interval is subdivided into two until a root is registered.

The main drawback of interval arithmetic is the overestimation of the bounds caused by the dependency problem. Higher-degree forms have been introduced to address this problem, such as Centered forms and Taylor forms [JKDW01; Neu02; MT06; Rat97].

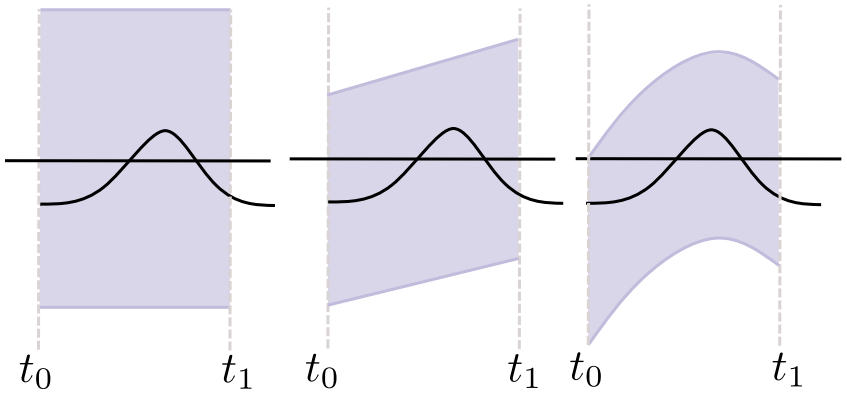


Figure 2.3: Behaviour of Interval Arithmetic (left), Revised Affine Arithmetic [FPC10] (middle) and a quadratic (right) inclusion function.

These different interval analysis methods can be formulated in terms of inclusion functions [JKDW01]. Figure 2.3 illustrates the example behaviour.

Inclusion functions: For each t on the ray interval $[t_0, t_1]$, the inclusion function $[f]$ is defined as the points containing all possible values of $f(t)$. $[f]$ consists of two functions f_- and f_+, where $f_-(t) \leq f(t)$ and $f_+(t) \geq f(t)$.

For a given function $f : \mathbb{R} \to \mathbb{R}$, $[f]$ is an *inclusion function* on the interval $[t_0, t_1]$ if:

$$\begin{gathered} \forall t \in [t_0, t_1], \quad f([t_0, t_1]) \subseteq [f]([t_0, t_1]), \quad [f] = \{f_-, f_+\} \text{ , where} \\ f_-(t) \leq f(t) \quad \text{and} \quad f_+(t) \geq f(t). \end{gathered} \tag{2.3}$$

$f_-(t)$ and $f_+(t)$ are the *lower* and *upper bounds* of the inclusion function $[f]$. For a detailed reference on inclusion functions, related literature can be consulted [JKDW01].

Taylor Inclusion If f is one-dimensional: nth order Taylor inclusion can be given as:

$$[f]_T(t) = f(m) + f'(m)(t-m) + \cdots + f^{n-1}(m)\frac{(t-m)^{n-1}}{(n-1)!} + [f^n](t)\frac{(t-m)^n}{n!} \tag{2.4}$$

around the query point m.

To illustrate the overestimation problem in interval arithmetic, take the simple example below:

Given two intervals $[X] = [\underline{x}, \overline{x}]$ and $[Y] = [\underline{y}, \overline{y}]$ the multiplication operation is defined by:

$$[X * Y] = [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}]$$

.

In case we have the x^2 function defined as $x^2 = x * x$, the above formulation would overestimate the bounds as interval arithmetic does not keep track of the dependent variables in the expression. In this case it is easy to define the same equation in interval forms as:

$$[X^2] = [0, max\{\underline{x}\underline{x}, \overline{x}\overline{x}\}].$$

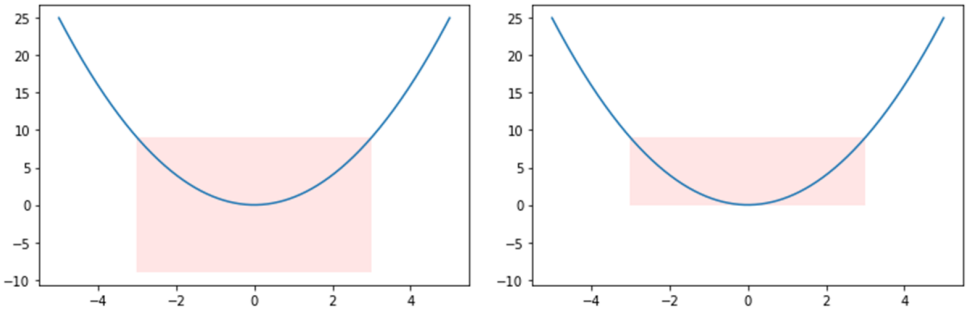


Figure 2.4: The overestimated bounds (left) and the tight bounds (right) for the example expression x^2 illustrating the dependency problem. When $[X] = [-3, 3]$ the two different evaluations $[X * X]$ and $[X^2]$ give different results with the bounds $[-9, 9]$ and $[0, 9]$ respectively.

This is illustrated in Figure 2.4. However, in a large expression with many operations, this property would create overestimated bounds which would not be automatically resolved.

Affine Arithmetic [dFS04] computes bottom-up inclusion functions – similar to interval arithmetic – while keeping track of dependent variables. This provides tighter bounds. Simplified versions have been suggested for ray tracing implicit surfaces [FPC10; KHK*09; SJ22], which can be considered linear inclusion functions similar to first-order Taylor forms.

Revised Affine Arithmetic [FPC10] provides a compact representation compared to Affine Arithmetic [dFS04] as it does not keep track of all error parameters separately but accumulates them to a single error parameter instead. While increasing the error for each interval, the compact representation is shown to be much more efficient as it uses less data to represent each expression [SJ22].

Interval Arithmetic and Affine Arithmetic can also be used to query over tiles in 2D [Kee20] and subfrustrum in 3D [GFN11; SJ22] to quickly discard large empty areas for rendering implicits.

2.2.3 Lipschitz Methods

The principle of Lipschitz continuity was first used by [KB89] for guaranteed localization of the intersections between a ray and an implicit surface. The method uses Lipschitz bounds of both the field function and its gradient along the ray direction in order to build an octree space partitioning used to prune empty areas. The roots are then isolated by investigating change in the gradient and the sign of the field function. Such approach requires the existence and calculability of the first and the second derivatives of the field function.

The existence of a global Lipschitz bound for signed distance field has been used in [Har96] to define a robust ray tracing algorithm : Sphere Tracing. This algorithm belongs to the ray-marching family. Each step size is computed as a function of the current field value and global Lipschitz constant guaranteeing that the isosurface is never missed. This approach is widely used for direct content creation in shaders [QJ13]. A large number of extensions to this approach have been proposed, such as safe overrelaxation [KSK*14], locally defined Lipschitz constant [GGP*15], optimal overrelaxation for planar surfaces [BV18] and visualization of implicit surfaces under derformation [SJNJ19].

Lipschitz constant measures how fast can a given function change. For a function $f : \mathbb{R}^n \to \mathbb{R}$,

a Lipschitz constant is a positive value L verifying:

$$|f(\mathbf{x}) - f(\mathbf{y})| < L\|\mathbf{x} - \mathbf{y}\| \ , \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^n.$$

In dimension three, if the function is differentiable, it can be defined as:

$$L = \max_{\mathbf{p} \in \mathbb{R}^3} \|\nabla f(\mathbf{p})\|$$

Sphere Tracing [Har96] shows that it is possible to march along the ray with the following formula:

$$stepsize = \frac{f \circ \delta(t)}{L}. \tag{2.5}$$

with t the current parameter along the ray. This formulation provides an easy-to-implement ray-marching algorithm, provided that the computational cost of calculating a practical Lipschitz bound is not high. SDFs present a useful special case of this approach, where $L = 1$. This way, the marching step size is simply equal to the field function evaluation value.

As stated in the original work, this formulation is similar to Newton-Raphson iterations. It uses the steepest possible slope, which is guaranteed not to cross the first intersection. Therefore Sphere Tracing converges linearly, and quadratically if the function is steepest at the first root [Har96]. The main limitation of this family of algorithms is the arbitrarily large number of steps required for grazing rays. Several successful attempts have been made to reduce the number of steps required by Sphere Tracing using various over-relaxation strategies [KSK*14; BV18]. These methods provide improvements on the marching step size for the overly conservative Lipschitz bounds in specific cases.

When applied to a density field instead of a distance field, the algorithm also suffers from the shape of the kernel function (null gradient near kernel support boundary) and the difficulty to compute a tight Lipschitz bound for an N-ary summation operator. For instance if there are large differences of radius for SCALIS primitives, the Lipschitz constant per primitive is inversely proportional to the prescribed radius therefore creating unnecessarily small steps globally. To overcome this problem, Segment Tracing [GGPP20] uses *local* and *directional* Lipschitz bounds: an upper bound of the absolute value of the directional derivative of the field function along the ray is computed on the ray sub-intervals to be processed. This approach drastically decreases the number of steps required for compactly supported primitives combined in a Blobtree [WGG99].

However, due to the usage of a bound on the *absolute value* of the derivative, it does not distinguish the local increasing/decreasing behavior of the field function. By considering the local monotonicity, we can further improve the processing of grazing rays and transparent rendering. By contrast, Bruckner [Bru19] proposes to map the density fields to approximate signed distance fields to render large-scale molecular models. Then, it is possible to render the scene efficiently using sphere tracing on the resulting normalized field. However, this method is specific to spherical molecular primitives with a unique radius in the scene. Furthermore, field normalization also requires a limited number of local primitive interactions. Our method in Chapter 1 relies on the same type of field normalization.

2.3 Acceleration Strategies

Implicit surfaces are defined with field evaluations that include every object in the scene, whether or not they would be visible for a given view-point or affect the final field computation in a given point in space. Several strategies has been proposed to reduce the unnecessary evaluations. For

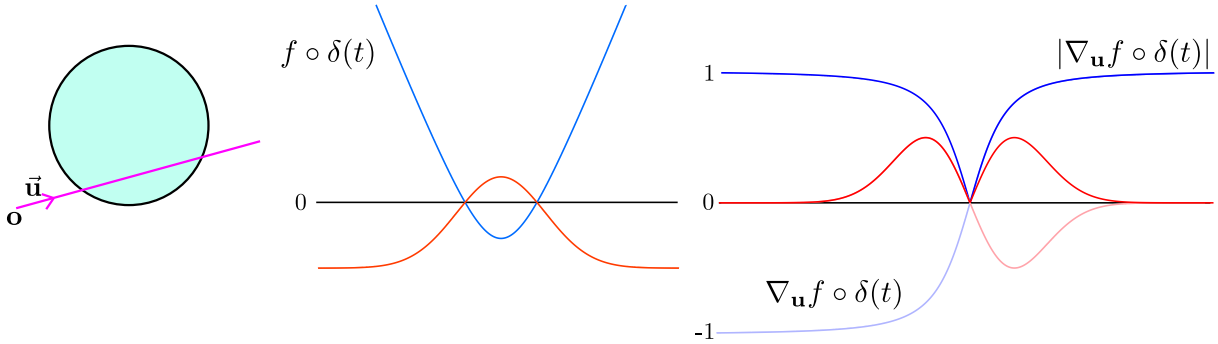


Figure 2.5: Left: Ray/primitive configuration. Middle: SDF (blue) vs density field with Gaussian kernel (red) along the example ray. Right: The directional derivative of the SDF (blue) remains nearly constant on a large proportion of the ray (i.e. everywhere except in a small area around the argminimum of the distance).

large skeletons – or more generally BlobTrees – computational times are highly correlated to the time required to iterate over all the primitives to find the ones that influence a given point along the ray (e.g. during field evaluation).

Tree pruning was introduced for efficient rendering Blobtree representations by evaluating a reduced tree by spatial subdivision and culling the primitives that cannot be intersected with the given rays [FGW01]. In [GPP*10] a bounding volume hierarchy (adapted for the management of the blending operation) was used for efficient evaluation. [GDW*16] proposed efficient tree traversal on GPU architecture by storing the tree information in a linear memory pattern and using axis aligned binary space partition trees for space pruning. Caching of field computations has also been proposed [SWG05; RLD*12] and recently suggested for animation where the field is only recomputed for the areas that go through substantial changes [JK23]. Recently for molecular rendering [Bru19], a dynamic data structure that is efficiently rebuilt at each frame by using the GPU rendering pipeline is suggested: for each ray, a linked list of intersection points with support of each primitive is computed relying on a GPU based quadric visualization algorithm [SWBG06]. In Chapter 1 we extend this approach for better management of segment primitives with varying radius (primitives whose support are also sphere cones). [LLZ*21] also suggested a similar strategy by keeping an ordered list of primitives for slicing.

[Kee20] suggest *tape shortening* on expression level to reduce the unnecessary calculations in the field evaluation by using Interval Arithmetic to discard unambiguous areas within a progressively subdivided grid.

Part III

Contributions

1

Fast ray-tracing of scale-invariant integral surfaces

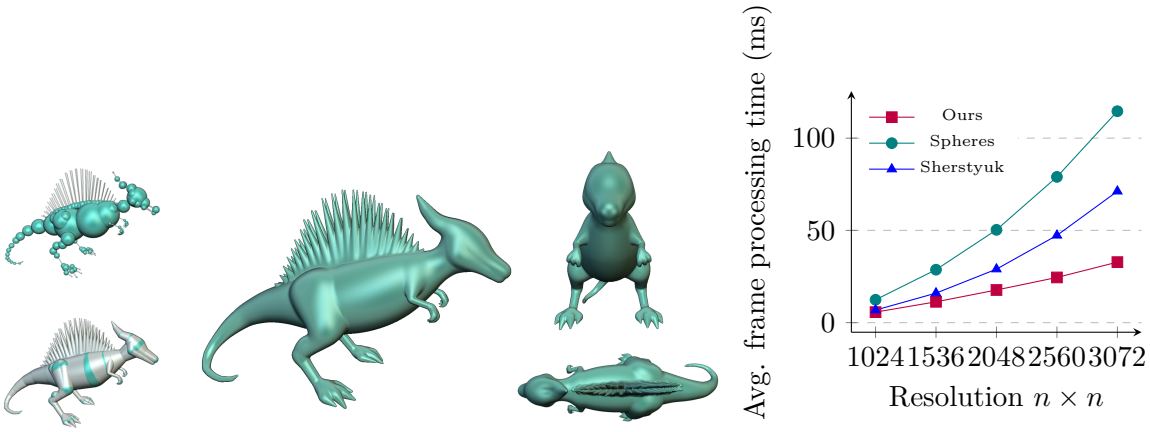


Figure 1.1: Scale-invariant integral surfaces (SCALIS) provide a way to define smooth surfaces from skeletons with prescribed radii defined at their vertices (with linearly interpolated radii along the skeleton edges). The generated surface can be seen as a smoothed version of an infinite union of spheres centered on the skeleton edges. We propose a new rendering pipeline allowing to visualize such surfaces in real-time. We provide comparison to revisited state of the art techniques on a large range of skeleton types for a variety of GPUs. Our method provides improvements for various resolutions on all combination of tested models and GPU hardware (see right graph. More comparisons in the results section).

Obtaining a robust computation of the closest ray/iso-surface intersection along the ray is the first challenge when ray tracing implicit surfaces.

In the case where we target integral surfaces, such as SCALIS, there are two additional problems that have to be tackled. Firstly, in order to perform the field evaluation, one needs to determine which primitives from the scene definition influences a given location in space. For large skeletons with many primitives, searching through the primitives to find the ones that influence a given location produces the main bottleneck. Secondly, since the SCALIS

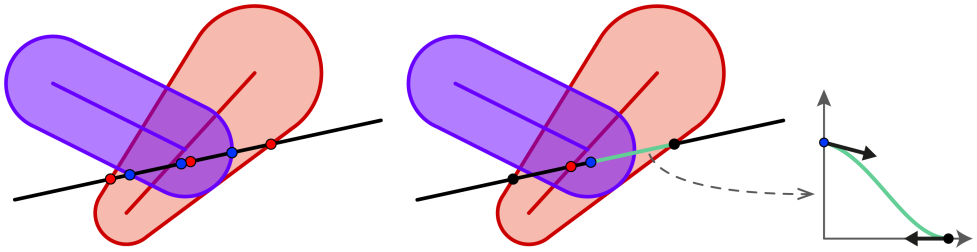


Figure 1.2: *Left:* In [She99a], boundaries of the primitive supports are used to subdivide the ray, in order to compute the local polynomial approximations of the field values. The approximations are then used to compute an approximation of the iso-surface position. To increase the approximation precision, supports are uniformly subdivided before generating the final segmentation which can result in intervals of highly incoherent size. *Middle:* our ray segmentation only relies on the estimated maximal influence of the individual primitives to define an initial segmentation with a small number of subintervals. We then rely on a dynamic subdivision in order to increase precision of approximation wherever required. *Right:* Hermite data defined at interval's end-points and associated polynomial interpolation.

field evaluations for individual primitives are relatively expensive to compute, a brute force approach that require a large number of field evaluations is not a viable option for an interactive application.

In order to overcome these challenges and provide a fast and precise ray-tracing-based integral surface visualization, our strategy combines four main components; 1) We introduce, during rendering, a segmentation of rays in intervals that captures the main field variations while generating only a small number of sub-intervals to be processed. 2) We transform the field values to a space that allows good quadratic polynomial approximation of the SCALIS field on each of the sub-intervals defined in the previous step along the ray. 3) Fast quadratic roots computation allows iterative building of polynomial interpolations that rapidly converge toward the real field function. And 4) in order to limit the computational overhead, selecting the primitives whose supports overlap a given interval along the ray is performed using a dynamic view-dependent acceleration structure. The last component is built on the fly relying on fast GPU construction of A–buffers which have been previously employed for metaballs rendering [Bru19] and transparency [Thi11; MCTB11; MCTB12].

This results in a method that goes beyond Sherstyuk's fast ray-tracing algorithm for convolution surfaces [She99a], notably allowing a better support of large radius variation while being less sensitive to skeleton tessellation.

1.1 Method

The shapes we study are defined as skeletons consisting of line segment primitives with the SCALIS field definition(see Equation (1.14)). Since SCALIS field evaluations are expensive to compute, brute force ray-marching approaches are not feasible. We therefore propose to rely on *local* field interpolations in order to reduce the number of field evaluations required for calculating the ray-surface intersections. To achieve high fidelity interpolations with lower degree polynomials, we propose to subdivide the ray into intervals that capture the main field variations along the ray. A different interpolation is used and refined in each interval, locally producing an accurate approximation. The overall idea and the processing of an example ray

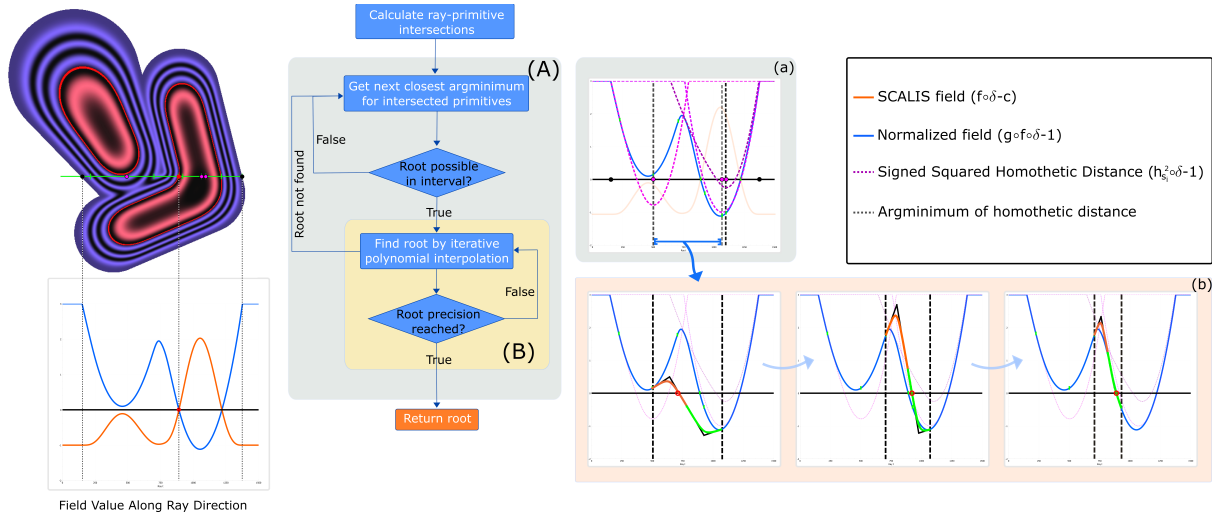


Figure 1.3: General pipeline of our method. *Top Left:* A slice of the SCALIS field consisting of three primitives and a ray (green) to be processed. Below it, the SCALIS field variation along the ray $f \circ \delta$ (orange) and the normalized field values (in blue) are given. We are interested in locating the zero-crossing (root of the Equation (2.2)) marked with the red disk. In the middle the two main loops of the algorithm are denoted as (A) and (B) and on the right these are shown respectively for the given ray: dividing the ray into intervals (a) and iterative root refinement by polynomial interpolation (b). The normalized field exhibits limited variations on sub-intervals defined by cutting the ray at local minima of the homothetic squared distance to individual primitives (purple curves in (a)).

can be seen on Figure 1.3.

Our approach relies on a bijective mapping of the SCALIS field to an approximate smooth homothetic distance field (squared) which has the exact same iso-surface of interest as the SCALIS field. All processing is done on this *normalized* field. This new field exhibits similar variations as the homothetic distance fields (squared) to individual primitives ${h_{S_i}}^2$ (see Equation (1.12)).

We use the correspondence between the normalized field and the ${h_{S_i}}^2$ fields in order to define our ray-subdivision strategy: the intervals are defined by cutting the ray at the minima of ${h_{S_i}}^2$ for each individual primitive S_i (purple curves in Figure 1.3 (a)). We expect such cuts to limit the number of oscillations of the normalized field in a given subinterval, a property required to perform field analysis through low degree polynomial interpolation. As can be observed in Figure 1.3 (a), the normalized field has this desirable property on each sub-intervals. By cutting the ray at the minima of ${h_{S_i}}^2$, the likelihood of finding a point within the implicit surface is increased, such points allows isolating a root by detecting a sign change. While the local minima of the homothetic distance field along the ray provides only an approximation to the local minima of the SCALIS field, this approach behaves well in practice (see section 1.8).

Our ray-tracing algorithm generates and processes intervals on the fly in depth-order until an iso-value crossing is detected. This corresponds to the main loop (A) in the diagram in Figure 1.3. We process a subinterval by iteratively refining a local polynomial interpolation. The interpolation is first initialized using the Hermite data sampled at interval end-points. Then, it is refined by reducing the interval extent, cutting it at the root of the polynomial which interpolates the field on the reduced interval (inner loop (B) in the diagram and graph of Figure 1.3). Provided that the polynomial interpolation has a root within the sub-interval, the

root of the successive interpolations will converge toward the real iso-surface as the interval gets iteratively smaller. We use smooth piecewise quadratic polynomials to obtain fast and stable root computations.

We first discuss the mapping from density field to homothetic squared distance field in section 1.2. Then, in section 1.3 we describe our ray subdivision strategy and in section 1.4 we present the approach used to perform the processing of a ray's subinterval. Finally, in section 1.5, we describe our GPU implementation relying on a dynamic data structure.

1.2 Approximated squared homothetic distance

Scale-invariant integral surfaces are defined from the homothetic distance to skeleton points as described in section 1.1.1. For most of the kernels used in practice, it is actually defined from a *squared* homothetic distance. We use this property to introduce a mapping that allows computing more precise quadratic interpolations. We first present the specific configurations where a quadratic polynomial can be fitted exactly to the mapped values, then discuss the general case.

1.2.1 Remapping SCALIS field values

In previous works on the integral surfaces, the study of infinite primitives has proven its usefulness for thickness control around skeletons [ZBQC13] and blending control [ZGC15]. We use it to analyze the homothetic distance to the iso-surface and provide better polynomial interpolation. We first introduce a field mapping for infinite line primitives, then generalize it for arbitrary skeletons.

Infinite line primitive Let us consider an infinite line primitive with a constant prescribed radius τ_0. For the main kernel families used to define skeleton-based implicit surfaces, the SCALIS field for such primitive in isolation can be defined as a function of distance d to the line by using a kernel $\tilde{k}$ of the same family with a different degree. Hence, the field can be defined as :

$$f_{line,\tau_0}(d) = f_{line,1}\left(\frac{d}{\tau_0}\right) = \lambda\,\tilde{k}\left(\frac{d}{\tau_0}\right) \tag{1.1}$$

where $\lambda = c/\tilde{k}(1)$. For the *Compact Polynomial* kernel of order i, we have $\tilde{k} = k_{i+1,\sigma}$ and for the *Cauchy* kernel, we have $\tilde{k} = k_{i-1,\sigma}$.

We generalize the approach of [Bru19] from metaballs to full skeletons with varying radius. The *homothetic squared* distance (instead of a Euclidian distance) to the *line* primitive (instead of point primitive) can be computed at any given location in space from the field value by applying the inverse of the function defined in Equation (1.1) followed by squaring the result :

$$g(f) = \left({f_{line,1}}^{-1}(f)\right)^2 = \left(\tilde{k}^{-1}\left(\frac{f}{\lambda}\right)\right)^2 = \left(\frac{d}{\tau_0}\right)^2 \tag{1.2}$$

For the *Compact Polynomial* kernel, this develops into:

$$g(f) = \left(\frac{d}{\tau_0}\right)^2 = \sigma^2 \begin{cases} ((1-(1-\frac{1}{\sigma^2})(\frac{f}{c})^{\frac{2}{i+1}})) \text{ if } f > 0\ , \\ 1 \quad \text{otherwise.} \end{cases} \tag{1.3}$$

while for the *Cauchy* kernel, this develops into:

$$g(f) = (1+\sigma^2)\left(\frac{c}{f}\right)^{\frac{2}{i-1}} - \sigma^2 \tag{1.4}$$

with the scale parameter σ and the order i of the kernel as defined in Equation (1.7,1.8). Note that in the case of radius $\tau_0 = 1$, this is also the Euclidean squared distance to the line.

Similarly, as $g(c) = 1$, a *signed* squared homothetic distance to the iso-surface of interest can be computed:

$$g(f) - 1 \tag{1.5}$$

In the remainder of the text, we call the field resulting from the application of Equation (1.5) to the SCALIS field the *normalized* field.

For a line with constant radius in isolation (or a long enough segment for the compact polynomial kernel), it is important to note that for a given ray, the squared distance from the ray to the line is a degree two polynomial. Hence, for such a configuration, the squared homothetic signed distance can be *exactly* interpolated by a quadratic polynomial defined by Hermite data sampled at an arbitrary position along the ray (i.e. by evaluating $g \circ f \circ \delta - 1$ and its derivative at the ray parameter t).

Other special configurations For a few other configurations, the function $g \circ f \circ \delta - 1$ is also a quadratic polynomial, hence it can be exactly interpolated by a quadratic polynomial defined by any Hermite data sampled along the ray. Such configurations are: any rays for two equal line primitives, all rays belonging to the bisecting plane between two parallel line primitives, the three rays that correspond to the equidistant lines for two crossing line primitives and the two rays corresponding to equidistant lines for two arbitrary line primitives. Indeed in all those cases, the global field is defined by $2f_{line,\tau_0}(d)$, hence a constant can be factored out from Equation (1.3).

Generalization Interestingly, Equation (1.3) can be used directly on the SCALIS field generated from a more general skeleton in order to compute a smooth approximation of the homothetic squared distance to the surface. Indeed, since g is a strictly decreasing function on $\mathbb{R}^+$, hence injective, the *normalized* field leaves all the level set unchanged (in terms of geometry), including the one corresponding to the iso-surface of interest (which now corresponds to an iso-value of 0 instead of c). And, outside of the blending area (e.g. branching) and the neighborhood of skeleton end-points, the field behavior will tend toward one of an infinite line before the application of g, hence toward a squared homothetic distance after application of g.

We designed the normalized field to have good properties for our algorithm: it does not create new oscillations in the field and is less dependent on the kernel used. The mapping has the beneficial property of removing the inflexion points of $f \circ \delta - c$ that are due to the Cauchy and Compact polynomial kernel shape (see Figure 1.4).

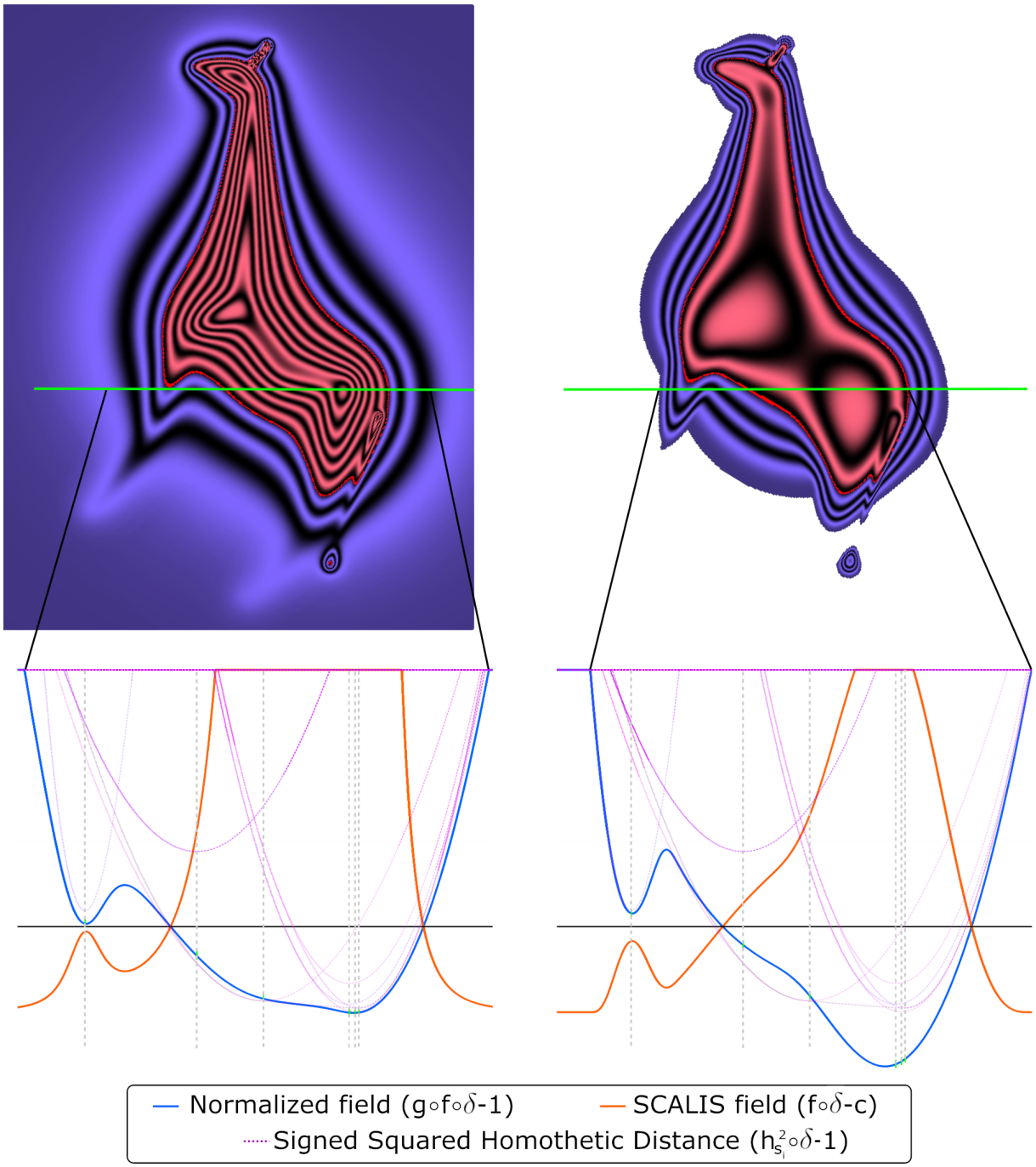


Figure 1.4: A sample ray on the skeleton defined in Figure 1.7. *Left:* Cauchy Kernel. *Right:* Compact Polynomial Kernel. The field variation along the ray $f \circ \delta$ (orange curve) exhibits limited variations on sub-intervals defined by cutting the ray at local minima of homothetic (squared) distance to individual primitives (purple curves). A high correlation between these values can also be observed, and is even more noticeable after mapping $f \circ \delta$ to approximate homothetic squared distances (blue curve).

1.3 Ray subdivision

In order to simplify the processing of a given ray, we want to define intervals with limited number of oscillations (typically either one or two local minima of $g \circ f \circ \delta - 1$ exists in the defined intervals). This provides better interpolations in configurations where the normalized field along the ray is not exactly a quadratic function.

We further use the fact that the normalized field can be seen as a smooth approximation of the homothetic distance (squared) to the surface $\min_i(h^2_{S_i}) - 1$ in order to simplify the processing of a given ray.

1.3.1 Correlation with $h^2_{S_i} - 1$

In order to ease the analysis of the field, we take inspiration from the Linderberg principle of absence of local extrema creation by convolution with a specific family of kernel [Lin91]. While we are not exactly in this context (Cauchy and Gaussian verify exactly the condition, compact kernel is only close to it, SCALIS is not a convolution when using varying radii), we experimentally observe a *similar* behavior. Furthermore, the mapping g is decreasing, it cannot create new local extrema, which allows a similar observation on the normalized field (see Figure 1.3 and 1.4). Similarly, we expect the local normalized field behavior to directly correlate to homothetic squared distance $h^2_{S_i}$ to individual primitives including in presence of varying radius. Indeed, it is defined by blending (i.e. smoothing) contributions of all skeleton-points based on their proximity (i.e. with sharper kernels the correlation between $g \circ f \circ \delta - 1$ and $h^2_{S_i} \circ \delta - 1$ increases).

The aforementioned evidence hints that a good sampling strategy is achievable by computing the arguments of the minima of the homothetic distance to the individual primitives $h_{S_i} \circ \delta$.

1.3.2 Argminimum of homothetic distance along a ray

In order to generate our ray subdivision on the fly, we need to find efficiently the argument of the minimum of the homothetic distance from a segment primitive S_i with linearly varying radius (Equation (1.12)) to the ray δ, i.e. we are trying to solve:

$$\underset{t}{\operatorname{argmin}}\ h_{S_i} \circ \delta(t) \tag{1.6}$$

Let us define the segment primitive S_i, parametrized by:

$$\mathbf{p}(s) = \mathbf{p}_0 + s\,\mathbf{u} \ \text{ and } \ \tau(s) = \tau_0 + s\ \Delta\tau \tag{1.7}$$

with $\mathbf{u}$ a unit vector and the parameter s in the range $[0, L]$, L being the length of the segment. For a skeleton point $(\mathbf{p}(s), \tau(s))$ in isolation, the Euclidean distance and the homothetic distance have the same argument of the minimum along the ray, the minima only differing by a factor $1/\tau(s)$. For the Euclidian distance, the minima (squared) have a closed form expression :

$${d_\delta}^2(\mathbf{p}(s)) = \|(\mathbf{m} + s\mathbf{u}) - (\mathbf{m} + s\mathbf{u})^T \mathbf{d}\,\mathbf{d}\|^2 \tag{1.8}$$

with $\mathbf{m} = \mathbf{p}_0 - \mathbf{o}$, where o is the ray origin and $\mathbf{d}$ is the ray direction.

Therefore, in order to solve Equation (1.6), we can invert the order in which minima are computed and we can instead study:

$$\underset{s\in[0,L]}{\operatorname{argmin}}\ \frac{d^2_\delta(\mathbf{p}(s))}{\tau(s)^2} \tag{1.9}$$

This gives the skeleton parameter $s_{\min}$ for which the minimum is reached. The ray parameter $t_{\min}$ can easily be derived from it (e.g. by computing the orthogonal projection of the skeleton point $\mathbf{p}(s_{\min})$ onto the ray).

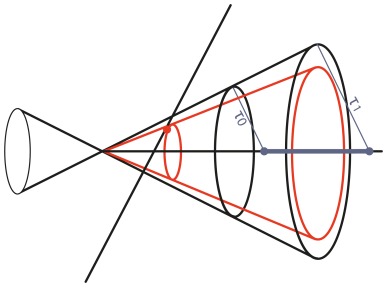
We therefore have to study a rational function whose poles are not in the range of interest (i.e. in the range $[0, L]$), thus the minimum of the function is either reached at the boundary of the interval $[0, L]$ or where the derivative cancels. By the cancellation of the derivative, we get a linear equation in the form $bx + c = 0$ with:

$$b = \tau_0 L^2(1 - (\mathbf{u}^T\mathbf{d})^2) - \Delta\tau L\mathbf{u}^T(\mathbf{m} - (\mathbf{m}^T\mathbf{d})\mathbf{d})$$

and

$$c = \Delta\tau\|\mathbf{m} - (\mathbf{m}^T\mathbf{d})\mathbf{d}\|^2 + \tau_0 L\mathbf{u}^T(\mathbf{m} - (\mathbf{m}^T\mathbf{d})\mathbf{d})$$

Note that Equation (1.6) could be solved directly. Recall that the sphere-union associated to a given primitive includes a cone section. Depending on the ray/cone configuration, it amounts to finding the smallest scaling factor to apply to the segment's radii such that there exists a single intersection between the ray and the scaled sphere-cone, see Figure inset. We find that our approach requires less computations and is easier to compute in a numerically stable way.



End-point correctors The minima of $h_{S_i} \circ \delta$ are highly correlated to the maximal field contributions for a given primitive. However, due to the integral nature of the SCALIS field definition, field variations diverge from this behavior near primitive end-points. While this is not problematic in general (due to blending by summation of adjacent primitives), this could become a problem near skeleton end-points. This problem is related to the problem of radius shrinkage at skeleton end-point observed in [ZBQC13]. When applying the *end-point correctors*, we force an additional sampling position near skeleton end-points, increasing the correlation between the argument of the minima of $h_{S_i} \circ \delta$ for individual primitives and the ones of $g \circ f \circ \delta - 1$. Note that in the absence of correctors, a heuristic presented in section 1.4.2 can compensate this shortcoming.

1.4 Ray processing

The ray subdivision strategy presented in the previous section not only provides intervals with limited number of oscillations, it also limits the initial number of intervals along a given ray. Smaller number of intervals result in smaller number of compulsory field evaluations while processing a given ray.

We also subdivide the ray when switching from an area not overlapped by any primitive support to an area overlapped by at least one primitive support (and vice-versa). This way, we can avoid processing large empty areas. This is easily achieved with the data structure presented in section 1.5.

We first present the quadratic polynomial interpolation used to define local field interpolation within a given subinterval, then we present the processing of a given subinterval.

1.4.1 Quadratic polynomial interpolation

Working with quadratic polynomial interpolation has the advantage of much more efficient, simpler and numerically stable root computation. As discussed in section 1.2.1, thanks to the

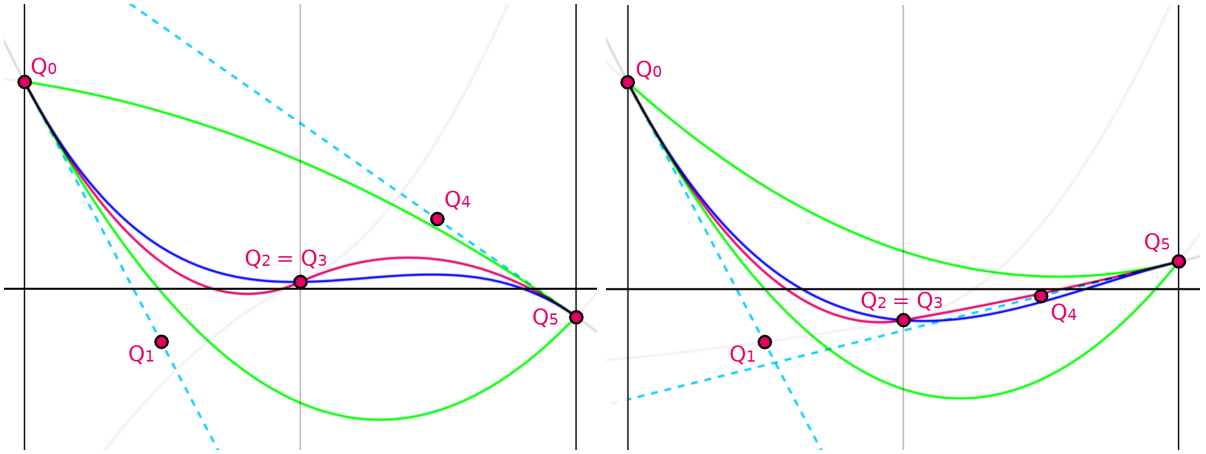


Figure 1.5: Two Hermite data configurations and associated polynomial interpolations: Hermite cubic interpolation in blue, quadratic interpolation by part in red and quadratic Hermite-Birkhoff interpolation in green.

field normalization, it also provides exact interpolation, in specific cases, independently of the kernel used. However, some configurations of Hermite data at interval's end-points cannot be realized by a quadratic interpolation (four constraints for only three degrees of freedom, see Figure 1.5). In such incompatible configuration, a possible approach would be to rely on Hermite-Birkhoff data, ignoring one of the two derivatives at interval end-points (see green curve in Figure 1.5). Instead, we propose an alternative solution that allows to respect the Hermite data at both end-points by relying on two piecewise quadratic polynomials, and without computing any additional field values. For simplicity we cut the initial interval in two sub-parts of equal size. Our interpolant is defined as a one dimensional Bézier curve on each part. For such curves, the control polygons are constrained such that the control points are equally spaced in abscissa. The ordinates of control points are chosen to verify Hermite data and interpolation smoothness. From the six control points $Q_{i=0..5}$ (three for each of the Bézier curves), the two first control points $Q_{0,1}$ of the first interval (respectively, last points $Q_{4,5}$ for the second intervals) are constrained by Hermite data. Remaining control points on both sub-intervals should have the same value $Q_2 = Q_3$ to ensure continuity and they should lie on the line joining the two adjacent control points Q_1 and Q_4 to ensure smoothness, which leads to the following ordinates:

$$\frac{1}{2}(h_1 + h_2) + \frac{L}{8}(h_1' - h_2'), \tag{1.10}$$

where (h_i, h_i') is the Hermite data at each end of the interval. Note that this value is also equal to the value returned by the cubic polynomial interpolation at the middle of the range (see Figure 1.5) as well as the average of the two Hermite-Birkhoff interpolations and still provide perfect interpolation when Hermite data are compatible.

In the case of the compact polynomial kernel, for general configurations, we experimentally observe a similar convergence rate (function of the interval size) for the quadratic interpolation on the normalized field $g \circ f \circ \delta - 1$ and cubic interpolation on the original field $f \circ \delta - c$ (see Figure 1.6, right). As expected, better results are obtained for primitives with limited radius variation and blending. For the Cauchy kernel, we obtain much better interpolations with the normalized field for small numbers of interval's subdivision, as such kernel is badly suited for low degree polynomial interpolation on larger intervals.

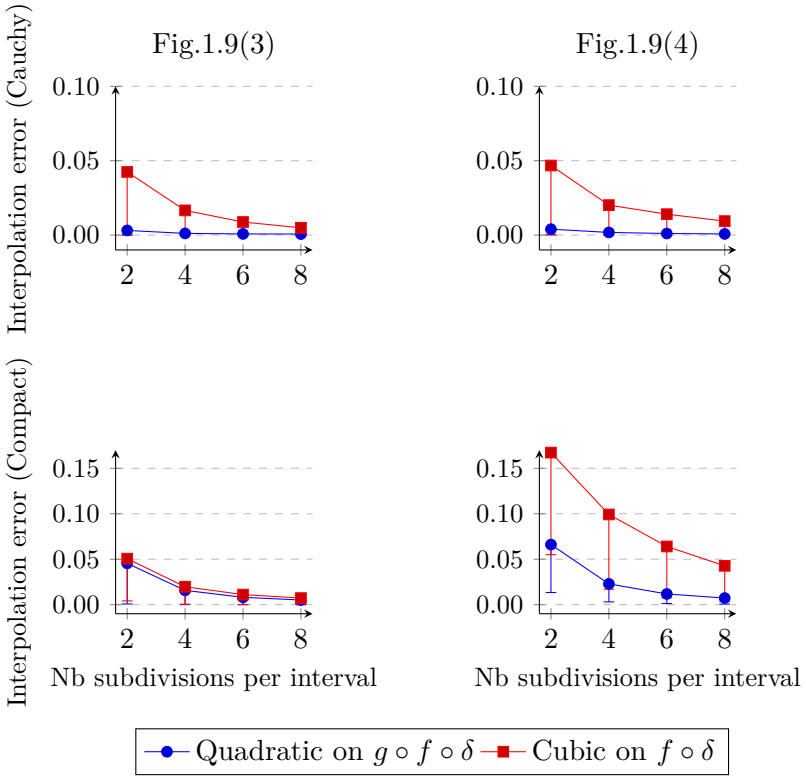


Figure 1.6: Comparison of convergence between the quadratic interpolation of $g \circ f \circ \delta - 1$ and the cubic interpolation of $f \circ \delta - c$. Both averages (main curve) and medians are computed over the intervals defined from our segmentation strategy with different level of subdivision of those initial intervals. Note that for the Cauchy kernel we ignore intervals for which all field values are below 0.015. This allows minimizing the impact of the kernel clipping required due to the infinite support of the Cauchy kernel. For each graph, 5k rays are launched in the scene of Figure 1.9 (middle and far right respectively).

1.4.2 Processing of an interval

We generate intervals iteratively and process them on the fly. For each interval we apply a routine in order to check if the iso-surface is crossed. If so, we return the position of the iso-surface along the ray, else we move to the next interval (see section 1.5).

Let us define $[t_{begin}, t_{end}]$ as a sub-interval to process. We use the quadratic interpolation of Hermite data presented in Section 1.4.1 in order to estimate the iso-surface position if it exists. The estimated position t_{root} is defined as the first root of the interpolation defined from Hermite data evaluated at t_{begin} and t_{end}.

The routine iteratively reduces the size of the interval in order to define Hermite interpolations of increasing precision (see Figure 1.3). By doing so, the estimate t_{root} converges toward the exact iso-surface location.

The interval size is reduced by moving one of its endpoints to the estimate t_{root} depending on the Hermite data evaluated in t_{root}. If no clear choice is possible (e.g. no sign change is detected in normalized field value and neither of the sub-intervals $[t_{begin}, t_{t_{root}}]$ and $[t_{root}, t_{end}]$ has a constant sign for their directional derivative, the first sub-interval is chosen and the second interval is stored to allow one step of backtracking.

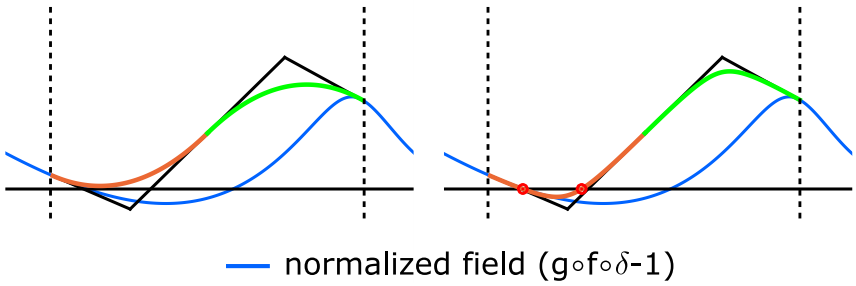


Figure 1.7: A polynomial interpolation can present smaller variations than the actual field function, resulting in possible missed iso-values. Until a root is isolated, we rely on rational Bézier curves to address this problem.

The iteration continues until the presence of the iso-surface can be rejected or the iso-surface has been located within a given error threshold. The structure of the interval processing loop is given in Algorithm 1.

Heuristic/Oracle On intervals presenting large radius variations (or in absence of end-point correctors) the polynomial interpolation might overestimate normalized field values $g \circ f \circ \delta - 1$, opening the possibility of missing the iso-surface. One solution to this problem would be to uniformly subdivide intervals in order to obtain better initial polynomial interpolations. However this comes at a large increase in the number of intervals to be processed. In order to avoid this, our algorithm uses a heuristic to limit the need for extra subdivision. We modify the oracle estimating the existence of a root (and its position if it exists) until an iso-surface crossing has been isolated (i.e. until a sign change exist between interval end-points).

We rely on rational Bézier curves until a root is guaranteed. Such curves allow having an interpolation that is arbitrarily closer to the control polygon hence increasing the chance to detect an iso-value crossing (see Figure 1.7). Note that with rational functions, root computation remains the same as long as the pole is not in the range of interest. In our implementation, a weight equal to 3 on the middle control point of the interpolation curve was sufficient on all tested examples.

With the modified oracle, even without any interval subdivision, failure cases are rare (see section 1.8), especially compared with the behavior of the sphere-tracing algorithm that requires arbitrarily large number of steps therefore creating holes in the surface, when working with limited number of steps for rapid rendering (see again section 1.8).

1.5 Dynamic data structure : efficient GPU implementation

When evaluating the SCALIS field on large skeletons with compact kernels, the computation time is mostly driven by the selection of primitives whose supports overlap the evaluation point. Similarly to [Bru19], our GPU implementation works in two steps, first, building a dynamic data structure where entry and exit points in primitives supports are stored per ray, then processing the ray. We rely on the stored data to serve as an acceleration data structure to efficiently retrieve the primitives whose supports overlap the given interval of the ray. When using the Cauchy kernel, we clip the support of a given primitive by a sphere-cone whose scaling $\sigma_{clipping}$ is computed to guarantee a maximal error $\epsilon_{clipping}$. This allows us to use the same approach for both kernel families.

Algorithm 1 Processing interval (t_{begin}, h_b, t_{end}, h_e).

```
// h_b, h_e are tuples containing Hermite data
n ← 0, saved ← None
while do++n < 32
    t_root ← Oracle(t_begin, t_end, i)
    if then t_root == ∞ and !saved return false, ∞
    else if then t_root < ∞
        h_r ← HermiteData(t_root)
        if then |h_r.val| < ε return true, t_root
        end if
        b_prim ← h_r.prim > 0 and h_b.prim < 0
        if !saved and h_r.val ≥ 0 and h_e.val < 0 and b_prim then
            saved ← [(t_root, h_r), (t_end, h_e)]
        end if
        if h_r.val < 0 or b_prim then
            t_end, h_e ← t_root, h_r
        else
            t_begin, h_b ← t_root, h_r
        end if
    else if saved then
        (t_begin, h_b), (t_end, h_e) ← saved
        saved ← None
    end if
end whilereturn false, ∞
```

The first step relies on a fast GPU construction of A–buffers [Thi11; MCTB11; MCTB12]. A geometry shader is used to generate a quad per skeleton segment such that the quad covers the projected segment primitive support. As the support of a given primitive is defined by intersecting some quadrics (a sphere-cone), we compute the quad using [SWBG06]. We adapt the proposed approach to compute segment-aligned quads to limit the number of generated fragments for which ray/sphere-cone intersections are computed (in particular for primitives whose maximal radii are comparatively small to the segment length). The ray-supports intersections are computed in a fragment shader. Entry and exit points in the sphere-cones, as well as the associated primitive id, are inserted in the A–buffer linked-list (see Figure 1.8) and sorted on the fly according to the depth of the entry [LHL14]. Note that contrary to [Bru19], the use of a compact support kernel avoids the need to cut-off primitives influences in this first step.

Of course, any variant of A–buffer creation could be used to accelerate this first step, for instance postponing the sorting of fragments until the ray processing step [Bru19] or using advanced memory management [SF14].

Once the first step of the processing is done, the linked-lists of the A–Buffer define intervals with a fixed set of segment primitives. We now face a choice. Subdividing the ray based on those intervals would minimize the maximum number of segment primitive to be stored per fragment shader call. However, it would also increase the number of field computations required. Increasing the interval size too much might require more memory to store all segment primitives whose supports overlap each intervals. On the opposite, subdividing the ray only when switching from an area not overlapped by any primitive support to an area overlapped by at least one primitive support (and vice-versa) would require storage of a large number of segment primitives

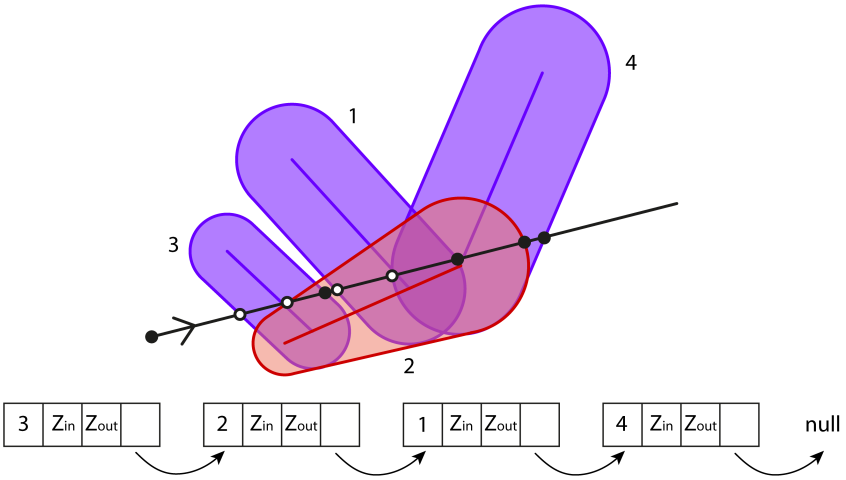


Figure 1.8: A-buffer linked-list associated to a given pixel after rasterization of all skeleton's primitives. Note that entry depth z_{in} and exit depth z_{out} in the support of a given primitive are stored in the same linked-list node. This linked-list is the main entry to the ray processing algorithm and allows efficiently retrieving the primitives whose supports overlap a given ray's subinterval.

at once depending on the ray/skeleton configuration, hence increasing memory usage of each individual thread.

We choose to combine this second subdivision strategy to our contribution presented in section 1.1. This presents a nice trade-off: it limits the number of segment primitives to be stored at once while generating long enough intervals, hence limiting the number of compulsory field evaluation. The fragment shader that renders the view generates and processes the intervals on-the-fly using algorithm 1 until an intersection with the iso-surface is found. The resulting linked-list processing loop is given in Algorithm 2.

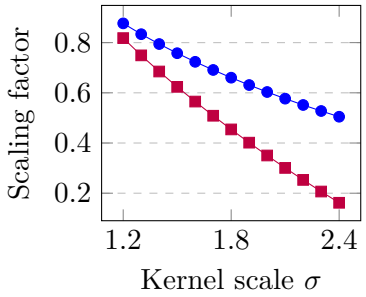
Due to possible numerical instability in the normalized field derivative computation (the latter being non continuous when the field switch from null to non null), we apply one step of sphere tracing to avoid those areas, note that this require no field evaluation as the field is null at the support limit (see comment in Algorithm 2).

Toward output sensitivity Similarly to Bruckner's approach to achieve output sensitivity [Bru19], it is possible to render an occluding depth buffer in order to limit the number of primitives that will be registered in the linked-lists. The depth buffer is initialized with depth values corresponding to positions that are guaranteed to be in the volume. Then all primitives whose support entries are beyond the occluding depth can be safely ignored. For efficiency those depth values need to be computed in a first render path on a per primitive basis. Due to the integral formulation of the field, the minimal radius around primitives does not necessarily correspond to the prescribed radius (e.g. typically for bent skeletons). In order to overcome this difficulty, we rely on two assumptions: usage of the skeleton *end-point correctors* and bounded radius variations: none of the two spheres at a primitive end-point completely encompass the other - e.g. each vertex should have an influence on the geometry of the union of sphere. Note that we do not render occluders for primitives corresponding to end-point correctors. In this context, thanks to the scale invariant property of the SCALIS representation, it is possible to analyze the worst skeleton configuration to derive a scaling factor for the sphere

Algorithm 2 Processing linked list.

GenerateRay(i,j)
primitives ← emptyHeap()
argmins ← emptyHeap()
frags ← InitFragList(i,j)
$t_{\text{argmin}} \leftarrow \infty$
$t_{\text{entry}} \leftarrow$ frags.currZIn()
while !(frags.empty() **and** primitives.empty()) **do**
  $t_b, h_b \leftarrow t_e, h_e$
  $t_{\text{exit}} \leftarrow$ primitives.back().val
  if primitives.empty() **or** $t_{\text{argmin}} < t_{\text{exit}}$ **or** $t_{\text{entry}} < t_{\text{exit}}$ **then**
    if primitives.empty() **then**
      $t_b, h_b \leftarrow t_{\text{entry}}, (g(0), \cdots)$
    end if
    while $t_{\text{argmin}} > t_{\text{entry}}$ and !frags.empty() **do**
      $t_{\text{prim}} \leftarrow \text{Argmin}(\sigma, \text{frags.currSegId}())$
      $t_{\text{out}} \leftarrow$ frags.currZOut()
      primitives.insert($\{t_{\text{out}}$, frags.currSegId()$\}$)
      argmins.insert(t_{prim})
      $t_{\text{argmin}} \leftarrow$ argmins.front()
      frags.next()
      $t_{\text{entry}} \leftarrow$ (!frags.empty()) ? frags.currZIn() : ∞
    end while
    $t_e, h_e = t_{\text{argmin}}, \text{HermiteData}(t_{\text{argmin}})$
  else
    $t_e, h_e \leftarrow t_{exit}, (g(0), \cdots)$
  end if
  // see text for management of entry/exit of non-null field area
  if $t_b < t_e$ **then**
    $f, t \leftarrow \text{processInterval}(t_b, h_b, t_e, h_e)$
    if f **then**
      GenerateIso-surface(t) **return**
    end if
  end if
  primitives.popElements($[](x)\{x \leq t_e\}$)
  argmins.popElements($[](x)\{x \leq t_e\}$)
  $t_{\text{argmin}} \leftarrow$ (!argmins.empty()) ? argmins.front() : ∞
end while

cones' radii such that the implicit surface is guaranteed to enclose the modified primitives. Computation of this scaling factor is discussed in Section 1.6.



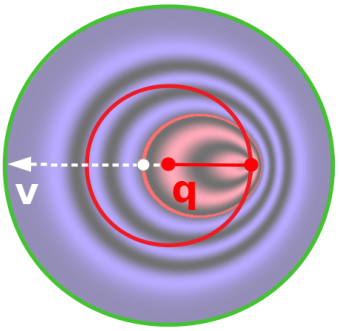
The quality of the minimal volume is related to the kernel scale parameter, the larger the scale the smaller the guaranteed volume is. For instance, for typical Compact polynomial kernel scale σ ranging from 1.5 to 2, the scaling factor range from ≈ 0.62 to ≈ 0.35 (see Figure inset - the purple curve correspond to our bounded radius variation). Similarly, for more constrained radius variations, better bounds can be computed (see Figure inset, the blue curve corresponds to constant radius). In section 1.8, we discuss rendering times with and without this optimization. In absence of end-point correctors, an alternative strategy would be to rely on the field behavior of primitives in isolation. Indeed, for long enough primitives with constant radius, there is a portion of the segment primitives for which the prescribed radius is guaranteed. While we do not use this property in our implementation, it could be used to improve the occluders for some specific skeleton configurations.

1.6 Radii scaling for sphere-cones enclosure

We present here the worst case skeleton configuration used to compute the scaling factor of the sphere-cones' radii that is used for computing the occluding depth buffer.

Usage of the end-point correctors guarantees the existence of a minimal length of skeleton around any skeleton point $\mathbf{q}$ with radius τ (i.e. the length of the correctors on both sides of $\mathbf{q}$) given that $\mathbf{q}$ is not part of an end-point corrector.

We want to find the minimal distance between the iso-surface and the point $\mathbf{q}$. Hence, we should study the field variation along arbitrary directions starting from $\mathbf{q}$. For an arbitrary direction $\mathbf{v}$, the minimal distance will be reached for the fastest decreasing field. Such field is obtained if the guaranteed length of skeleton is leaving $\mathbf{q}$ in the direction $-\mathbf{v}$ (kernels are decreasing functions). Similarly the influence of a given skeleton point is decreasing more rapidly for smaller prescribed radii. Hence, the worst case scenario is reached for two segment primitives in the direction $-\mathbf{v}$ with radius τ in $\mathbf{q}$ and the fastest authorized radius variation (see Figure inset). Thanks to the scale invariance of the surface, we only need to compute the scaling factor for a unit radius which we perform numerically (which corresponds to the distance between $\mathbf{q}$ and the iso-surface in the direction $\mathbf{v}$ - see Figure inset). Note that this worst case configuration is actually independent from the actual skeleton configuration, no additional skeleton processing is required.



1.7 Lipschitz constant computation

When applying sphere-tracing, computing the Lipschitz constant per primitive (e.g. ignoring the ray/primitive configuration) can have a large impact in presence of varying radius along primitives. Indeed, for the studied implicit surfaces the Lipschitz constant is inversely proportional to the smallest radius. In order to alleviate this problem in our sphere-tracing implementation,

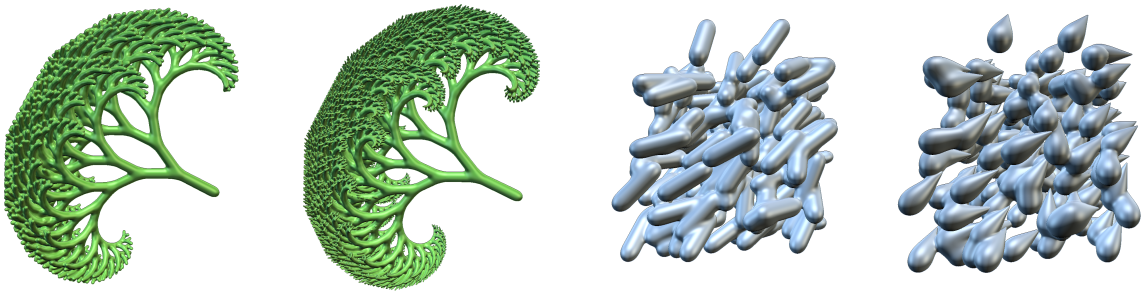


Figure 1.9: *Left:* Procedural trees. *Right:* Random skeletons with constant and varying radii used to test the resilience of our algorithm.

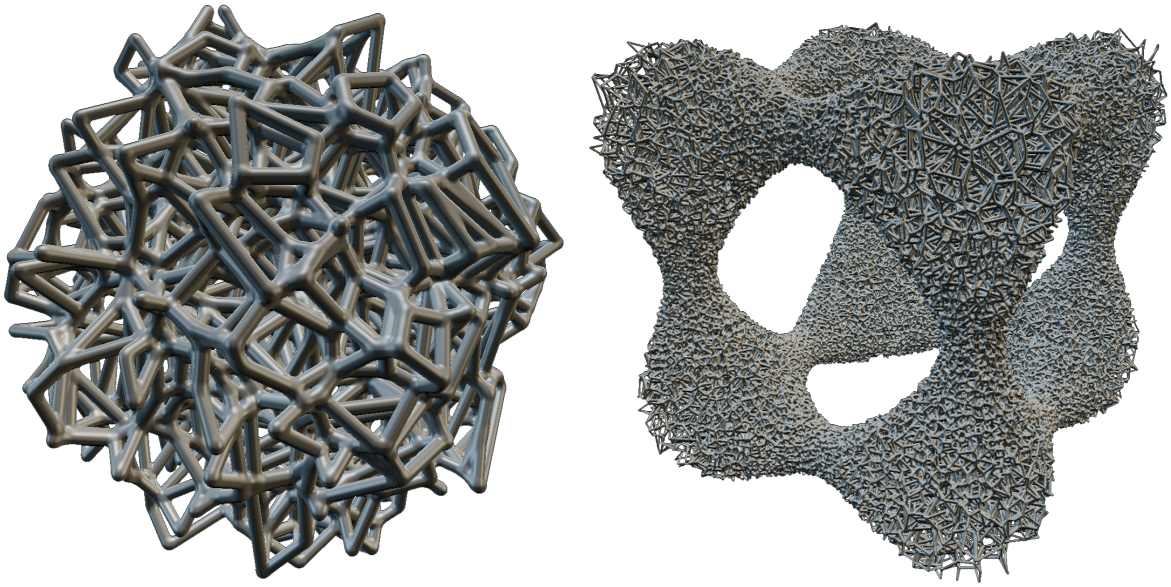


Figure 1.10: Procedural foams whose skeletons are defined as edges of a voronoi diagram of points.

we compute the Lipschitz constant both per primitive and per ray. This can be done by computing the primitive point with the smallest radius whose kernel support is intersected by the ray. Based on Equation (1.8), the kernel support associated to a skeleton point is intersected by the ray if and only if:

$$\|(\mathbf{m} - s\mathbf{u}) - (\mathbf{m} - s\mathbf{u})^T \mathbf{d}\,\mathbf{d}\|^2 - \sigma^2(\tau_0 + s\,\Delta\tau)^2 \leq 0$$

where σ is the kernel scale. The smallest radius can then be deduced from these polynomial roots.

1.8 Results

Our method was implemented in C++ using the OpenGL library and shaders were programmed in GLSL. We have tested our implementation on a large range of objects including artistic shapes (see Figures 1.1, 1.7, 1.11, 1.13), truss structures (see Figure 1.14), branching structures (see

Figure 1.11: Real-time rendering allows users to efficiently explore kernel scale parameters for a given skeleton.

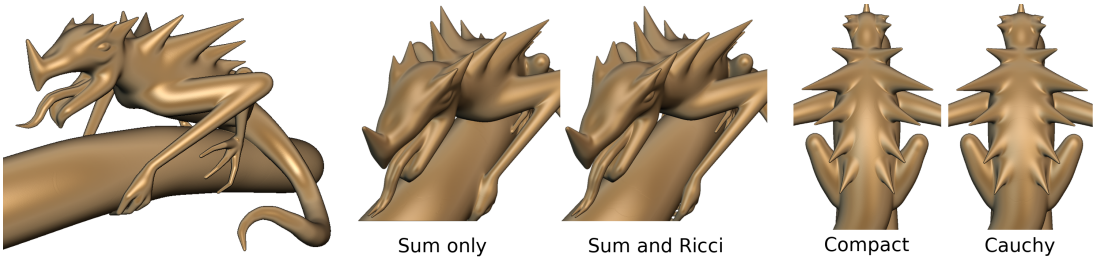


Figure 1.12: *Left:* Model generated using a Cauchy kernel and Ricci blendings of subcomponents (main body, eyes, tongue, legs and branch), *Middle:* Comparison with summation-only blending, *Right:* Comparison with compact polynomial kernel.

Figure 1.9), procedural foam structures (see Figure 1.10) and random skeletons (see Figure 1.9) using both low-end and high-end graphic cards (namely an Intel UHD Graphics 630, an nVidia Quadro P1000 and an nVidia GeForce 2080 RTX). Examples have been chosen to present a large variety of skeletons in terms of number of segments, local density of primitives and whether or not the radii change. In order to assess the efficiency of our method, we compare it to several state of the art techniques: sphere-tracing on the field $g^{\frac{1}{2}} \circ f \circ \delta - 1$ (similar in spirit to [Bru19], see Section 1.7 for the computation of the Lipschitz constant), Shertyuk fast ray-tracing [She99a] and segment-tracing [GGPP20] on $f \circ \delta - c$. Comparison includes both GPU rendering time and statistics per ray (computed on a CPU implementation). For Sherstyuk fast ray-tracing, we only compare rendering time as field evaluations are not performed in the same way. Comparisons with the segment-tracing algorithm are done in the context of homothetic distance primitives (e.g. Equation (1.13)) instead of SCALIS primitives (e.g. Equation (1.14)) as we do not have a computation routine for the local directional Lipschitz bounds on the latter equation. It only includes ray statistics. Our study mostly focuses on the compact polynomial kernel, we explicitly specify in the text when experiments are run with the Cauchy kernel.

Note that in practice, it is is also possible to blend different skeletons with a blending sharper than the summation blending, e.g. by relying on Ricci's blending [Ric73] (see Figure 1.12). Indeed, such blending keeps the correlation of the field with the homothetic squared distance to individual primitives by defining a blending behavior parametrized between a summation blend and a maximum blend.

GPU rendering time We record the average rendering time for a set of viewpoints around our test objects. In order to provide fair comparisons, the GPU implementation of the sphere-tracing and the one of the Shertyuk ray tracing both rely on the A-buffer. In our implementation

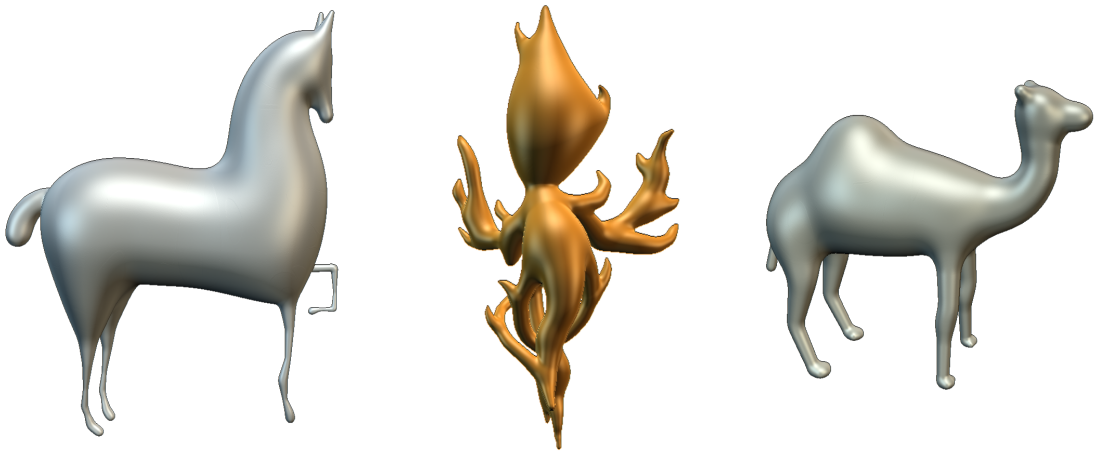


Figure 1.13: Skeleton-based free form modeling.

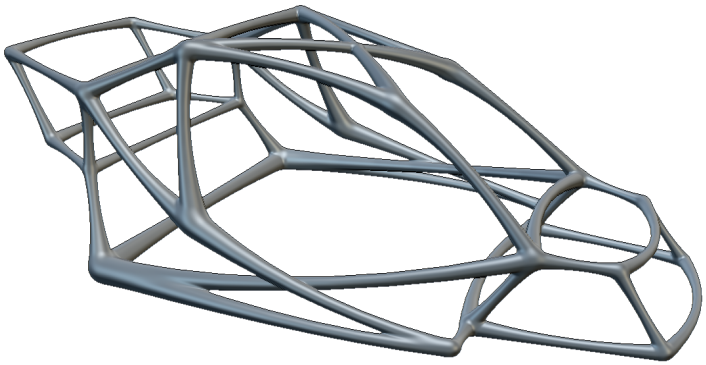


Figure 1.14: Truss structure.

of Sherstyuk's method, we use four subdivisions per primitive, which correspond to the coarsest subdivision allowing to obtain visually acceptable results on primitives with constant radius. Our implementation of the sphere-tracing routine also uses a ray subdivision strategy in order to discard space outside of any primitive support as well as to limit the maximal number of segments to be stored and processed at the same time during the ray processing loop. On each subinterval the number of sphere-tracing steps is bounded (typically 256 in our implementation).

For the compact polynomial kernel, all renderings are done at a resolution of 2048×2048 and rendering times are provided in Table 1.1. On all tested examples, we observe improvements of the global rendering time with our method in comparison to the most competitive one (which is dependent on the model). Rendering time reduction ranges from -34.2% to -48.7% on the Intel card, from -51.5% to -78.0% on the nVidia Quadro card and from -35.3% to -72.1% on the nVidia RTX card. In addition, both sphere-tracing and Sherstyuk's fast ray tracing can present visual artifacts (see Figure 1.15, note that those artifacts can be reduced at the expense of additional computation time). It is also important to note that in presence of a large amount of blending - for instance if a large number of segments connect in one vertex (such as in Figure 1.10) - the sphere-tracing algorithms would require usage of under-relaxation to properly

find the surface.

We also have tested our method at higher resolutions where we observe similar improvements (see Figure 1.18). In order to perform a stress case on the nVidia RTX, we rendered a foam structure consisting of 526k segments using the compact polynomial kernel, running at 15 frames per second (see Figure 1.10, right). Finally, we provide individual timing for the A-buffer creation step (with and without occluders) and the ray processing steps. As expected, usage of occluders can have a large impact in terms of rendering time (as well as memory consumption) depending on the nature of the skeleton.

For the Cauchy kernel, we mainly tested our method on the nVidia RTX card with a lower resolution of 1536×1536 in order to account for the higher number of primitive overlaps generated by the larger clipping sphere-cone (e.g. $\sigma_{clipping} = 4$ for $\sigma = 0.3$ and $\epsilon_{clipping} = 0.005$), otherwise this would lead to an A-buffer overflow. The use of a kernel with a larger footprint results in an increase of computational time. With our method, timings range from 7 to 54 milliseconds (excluding Fig.1.10(Middle)). With respect to sphere tracing, this corresponds to a rendering time reduction ranging from -46.2% to -81.3%. We observed similar improvements on the two other GPUs when run at a resolution of 1024×1024. Note that we do not compare runtime with the Sherstyuk method as it produces numerous artifacts due to poor polynomial interpolation.

Interactive modeling The rendering time achieved by our method allows interactive manipulation of a large range of skeletons. Both, random and procedural skeletons (see Figure 1.9), can be generated interactively. In our implementation skeletons are generated on the CPU at each parameter change. Fast rendering also allows real-time exploration of kernel parameter as depicted in Figure 1.11.

Statistics In order to compute per ray statistics, we have launched sets of rays in six directions (three main axis with both positive and negative orientations). For each model, we have computed the axis-aligned bounding-box from sphere-cone supports. In each direction, we sample the ray origins uniformly in the associated bounding-box's face such that a total of 10 millions rays are launched per model. For each ray, we measure the average, median and maximal number of steps required to process the ray. Note that the maximal number of rays are more important for SIMD architectures. Statistics are provided in Table 1.2 and 1.3. It is important to recall that our method relies on gradient computation for the evaluation of directional derivative. As described in [ZBQC13], both field and gradient can be computed for less

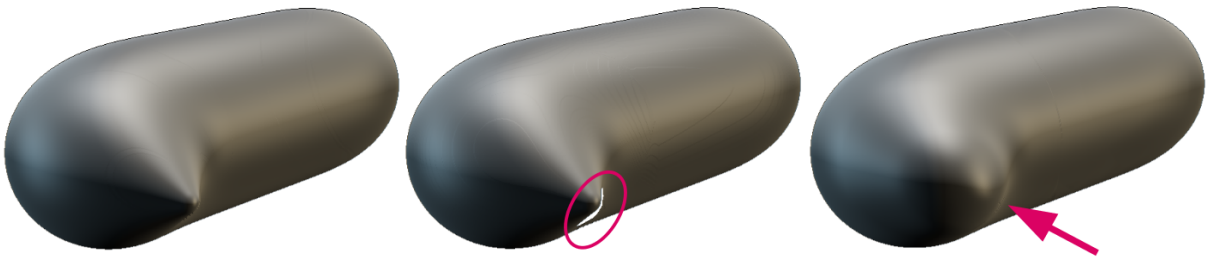


Figure 1.15: *Left:* our method presents minimal errors in iso-surface intersections, *Middle:* when using sphere tracing with a number of steps limited to a few hundreds (in order to limit runtime cost) holes can appear in the surface,*Right:* Sherstyuk fast ray tracing can produce large deformation of the surface depending on the viewpoint.

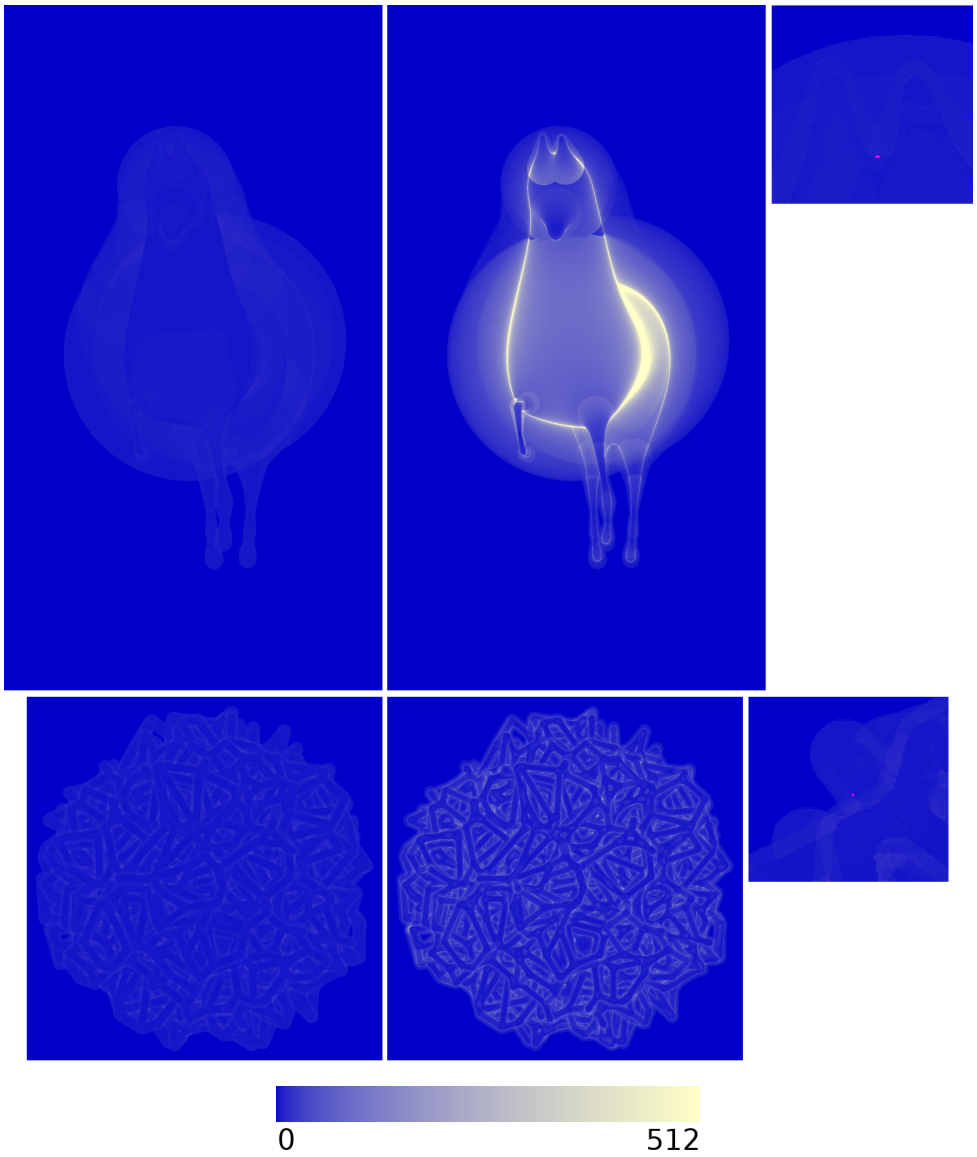


Figure 1.16: Number of field value computations used during the processing of a ray. The red pixels correspond to iso-surface crossing missed by our method. *Left:* Sphere-tracing *Right:* Our method. The rightmost figures show close-ups from the areas with missed roots in purple. These occur at grazing angles.

Object	Intel UHD Graphics 630			nVidia Quadro P1000			nVidia GeForce 2080 RTX		
	Ours	Sphere	Sherstyuk	Ours	Sphere	Sherstyuk	Ours	Sphere	Sherstyuk
Fig.1.9(4)	290.4	551.9	565.9	84.2	300.4	262.5	7.9	28.2	18.4
Fig.1.9(3)	284.4	374.4	526.2	80.0	169.0	230.3	7.9	16.1	17.0
Fig.1.9(2)	243.0	388.0	384.1	127.8	302.0	268.8	12.0	27.2	22.5
Fig.1.9(1)	169.5	244.6	289.6	74.6	169.0	182.1	7.3	15.6	15.3
Fig.1.11(Middle)	121.3	204.9	178.4	31.8	120.0	79.9	6.2	12.9	7.7
Fig.1.7	116.5	169.8	177.1	23.8	81.9	70.8	6.3	12.9	7.7
Fig.1.13(Middle)	159.3	272.0	265.4	39.9	151.0	115.4	6.6	15.1	10.0
Fig.1.1	129.7	247.8	202.4	33.6	152.7	93.3	6.7	22.0	13.2
Fig.1.13(Left)	207.7	366.2	383.0	49.2	182.0	151.1	6.9	18.9	11.3
Fig.1.13(Right)	117.5	187.1	181.1	29.1	106.8	76.3	5.9	10.4	7.0
Fig.1.14	90.5	120.6	150.7	22.5	69.1	76.0	5.5	8.4	7.5
Fig.1.10(Left)	294.5	390.0	447.8	119.4	236.0	246.0	11.3	25.6	22.7
Fig.1.10(Middle)	-	-	-	-	-	-	62.5	259.9	97.3

Table 1.1: Rendering Times (in milliseconds) averaged over full rotations around the object, at 2048x2048.

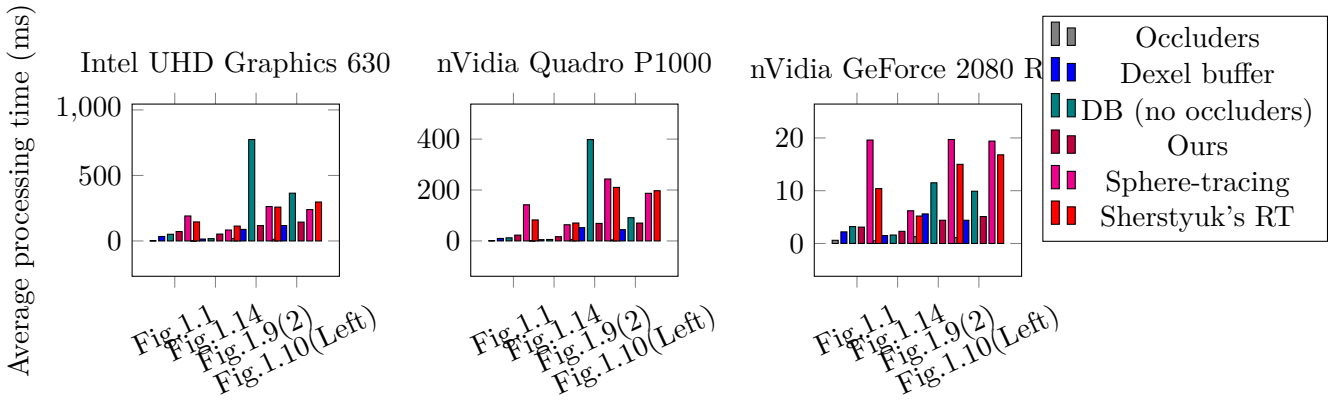


Figure 1.17: Individual times for each substep of the methods (occluding depth buffer, dexel buffer creation with and without occluders and ray processing). For models presenting a large number of depth layers, occluders provide non negligible improvement of rendering time.

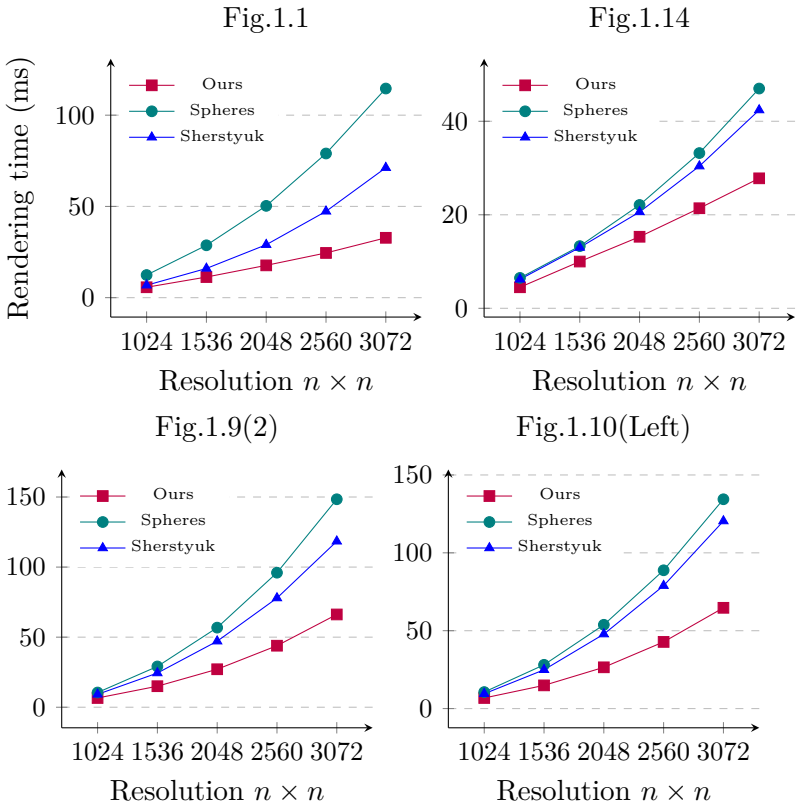


Figure 1.18: Average frame processing time versus rendering resolution on a nVidia GeForce 2080 RTX.

Object	# of Skeletons	Our Method			Sphere Tracing		
		Avg	Median	Max	Avg	Median	Max
Fig.1.10(Left)	4722	5	4	37	16	11	24580
Fig.1.9(1)	255	5	4	80	18	13	2389
Fig.1.9(3)	201	6	6	84	19	14	2124
Fig.1.13(Right)	127	5	5	27	36	24	2980
Fig.1.1	213	5	5	132	68	22	42214
Fig.1.13(Left)	55	5	5	23	73	48	8298
Fig.1.9(4)	201	8	7	84	413	330	21770
Fig.1.11(Middle)	95	5	4	23	31	23	4866
Fig.1.13(Middle)	55	5	5	21	37	20	7559
Fig.1.7	36	4	4	34	28	18	4372
Fig.1.14	608	5	4	46	16	10	8345

Table 1.2: Statistics calculated for our method and sphere tracing for Compact Polynomial. Similar results are calculated for the Cauchy kernel with decrease in median number of steps in the range between 36.3% and 97.9%

Object	# of Skeletons	Our Method			Segment Tracing		
		Avg	Median	Max	Avg	Median	Max
Fig.1.10(Left)	4722	5	4	53	9	7	383
Fig.1.9(1)	255	5	5	55	9	8	549
Fig.1.9(3)	201	5	4	24	9	7	340
Fig.1.13(Right)	127	4	5	20	16	14	413
Fig.1.1	213	5	4	65	15	11	736
Fig.1.13(Left)	55	5	5	16	23	23	302
Fig.1.9(4)	201	6	4	25	177	197	1680
Fig.1.11(Middle)	95	4	4	18	15	14	277
Fig.1.13(Middle)	55	4	6	18	13	16	531
Fig.1.7	36	4	5	16	12	12	524
Fig.1.14	608	5	6	74	8	7	348

Table 1.3: Statistics calculated for segment tracing [PGMG09] and our method on blob primitives.

than twice the evaluation cost of a single field value. Furthermore, as both values are computed at once, segment parameters are only fetched once from memory.

We observe a reduction for all statistics when using our method (see Table 1.2), including when primitives present small radius variations (e.g. the radius is either approximately constant over the full skeleton or is approximately constant by part on the skeleton). Larger reductions are observed in presence of varying radii.

However, the main benefit is achieved along grazing rays that present the maximal number of steps (a well-known problem of the sphere-tracing algorithm - see Figures 1.8 and 1.16).

When using distance primitives instead of SCALIS primitives, this limitation of the sphere-tracing algorithm can be alleviated by the use of directional Lipschitz bounds (e.g. the segment-tracing algorithms). In this context, our method still provides reduction of the maximal number of steps required along grazing rays (see Table 1.3).

In order to validate the robustness of our method, we have also measured errors along rays. In practice, less than $10^{-3}\%$ of the rays are missing an iso-surface crossing, all of them are grazing rays (see Figure 1.16 and 1.8). Those few errors are due to Hermite data configurations for which control polygons of the interpolant do not cross the iso-value 0. This limits the efficiency of our rational interpolation heuristic. In order to mitigate this shortcoming, it could be interesting to investigate 2D quadratic Bézier curves which would allow increasing the amplitude of the control polygon by moving the abscissa of the control points.

Limitation Our algorithm and its implementation present a few limitations. First, the heap used to store segment primitives during the ray processing loop should be large enough to accommodate all primitives whose supports overlap the initial intervals defined by the argument of the minimum of homothetic distance. Observe that this limitation could be mitigated by adding an additional break condition in Algorithm 2 based on the current heap size. Secondly, output invariance could be improved by combining our current strategy with occluders generated from primitives considered in isolation. Finally, the A-buffer can become expensive to build for finely tessellated curves (due to overlaps between primitive supports). Two main directions could be investigated : curve primitives [FSHZ19] and curved simplification based on local radius and

kernel scale. Similarly, finely tessellated skeleton also increase the number of argument of the minimum of homothetic distance along the ray.

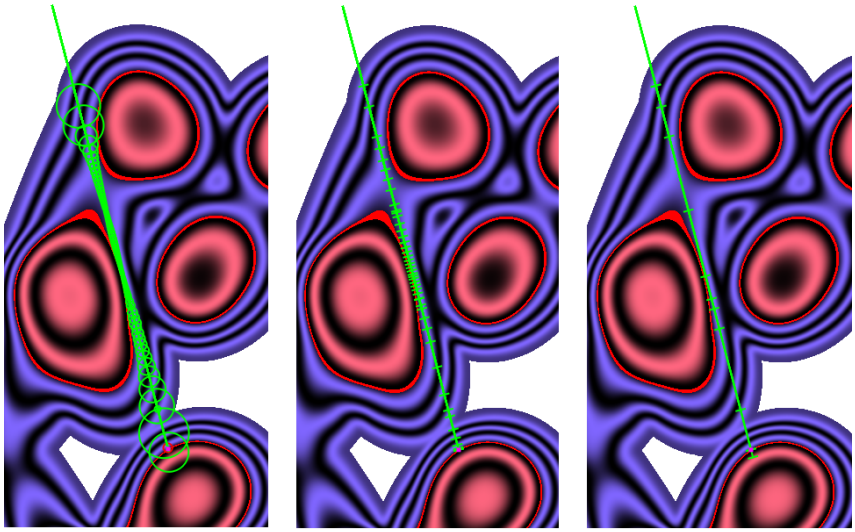


Figure 1.19: Comparison of grazing rays in a slice of the scene of Figure 1.9 (far left) for sphere-tracing, segment-tracing and our method. Note that homothetic distance primitives (Equation (1.13)) are used for this figure in order to allow direct comparison of the three methods.

1.9 Conclusion

We have introduced a new rendering algorithm for scale-invariant integral surfaces. While not providing a guarantee for robustness of root localization, our approach presents no visible artifacts while decreasing computation time by up to 70% in comparison to revisited previous work. Our key contributions are 1) the usage of a new initial sampling strategy of the rays based on homothetic distance to segment primitives, 2) the introduction of a new field mapping combined to quadratic polynomial interpolation, and 3) an efficient GPU implementation relying on a A-buffer data structure. We believe this technique could help in further study of a larger range of integral surfaces.

Among future directions of research, we can mention management of triangle primitives (these share the main properties used in our method) and transparent rendering (relying on depth slabing to limit A-buffer creation cost and memory usage). Finally, a combination of an A-buffer acceleration structure and/or a new sampling strategy to existing rendering techniques for more general implicit surfaces such as [GGPP20; Kee20] could provide interesting optimizations.

The rational interpolating for root finding as described in this section is mostly adapted for opaque rendering. While not guaranteed to isolate all the roots, it does not lead to visual artifacts. In the next section, we focus on robust root isolation for transparent rendering and slicing. The acceleration structures defined in this section can also be applied for slicing and transparency, as long as the depth occluders are not used.

2

Per-Primitive Interval Arithmetic

Applications such as slicing for additive manufacturing and transparent rendering require robust root finding. Self-validated numerical methods provide robust bounds on query intervals and can be used for guaranteed root-finding. However, the error bounds around these query intervals rapidly grow on complex expressions such as the implicit surface definition of integral surfaces with varying radius (Section 1.1.1) that we use.

In this section, the objective is to efficiently use integral surface segment primitives with linearly varying radius inside existing robust methods such as interval arithmetic and segment-tracing. We first present a simple and efficient way to compute field function bounds on a per-primitive basis for using these bounds in an interval arithmetic framework. Then we study directional Lipschitz bounds that improve our bounds in specific interval configurations and provide a way to use integral surface within the segment-tracing algorithm. Finally, as an application of our method, we present a transparent rendering algorithm that can be used for rendering and slicing.

2.1 Per-primitive field bounds for interval arithmetic

In order to use integral surfaces inside interval arithmetic-based algorithms, we need to derive field value bounds given an interval $[t_0, t_1]$ along a ray, e.g., finding an interval $[f_-, f_+]$ such that:

$$f \circ \delta([t_0, t_1]) \in [f_-, f_+].$$

For the line segment primitives with linearly varying radius the field is always monotonously increasing and then decreasing on the given input ray direction.

Monotonous intervals can be detected by checking the sign of the derivatives at the interval end-points. With this observation, for monotonous intervals, the interval inclusion is trivial (see Figure 2.2 (Left)). We can directly use the endpoint values as bounds:

$$[f_-, f_+] = [\min(f \circ \delta(t_0), f \circ \delta(t_1)), \max(f \circ \delta(t_0), f \circ \delta(t_1))]$$

For ambiguous intervals where we have both positive and negative derivatives (increasing and

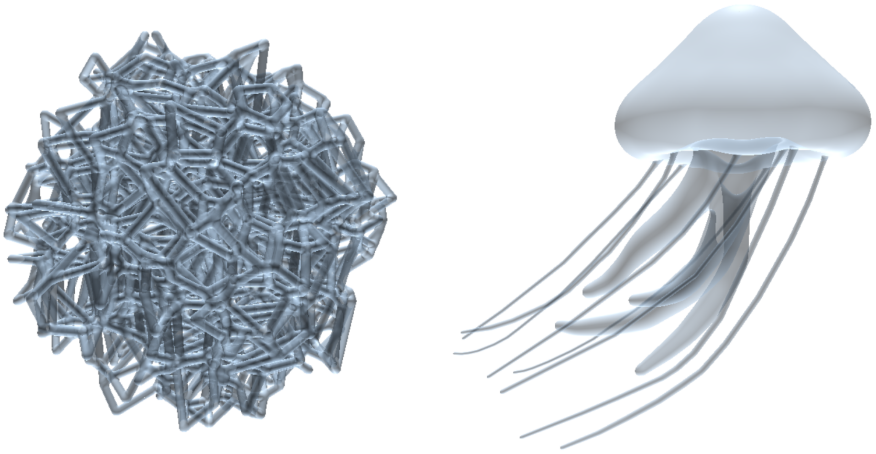


Figure 2.1: Transparent rendering results for microstructures (left) and an artistic model (right).

decreasing values), we can either use infinite values as upper bound :

$$[f_-, f_+] = [\min(f \circ \delta(t_0), f \circ \delta(t_1)), +\infty] \ ,$$

or calculate a bound on the absolute value of directional derivative and extrapolate field values based on the maximal derivative and the field values at the interval endpoints. We can then compute the upper bound as the intersection between two lines (see Figure 2.2 (Right)). We describe the calculation of Lipschitz bound in the next section. This calculation allows bounding the field values robustly. The resulting per-primitive intervals can then be used in classical interval arithmetic evaluation.

2.1.1 Directional Lipschitz bound calculation

In order to provide tighter bounds on the intervals containing a maximum, we derive a directional Lipschitz bound on a per primitive basis. To do so, we study the absolute value of the derivative of $f \circ \delta$, which is defined by :

$$|\nabla f(\mathbf{p})^T \mathbf{u}| = \left| \int_{\mathbf{q} \in S} \frac{1}{\tau_S^2(\mathbf{q})} k' \left(\frac{\|\mathbf{p}-\mathbf{q}\|}{\tau_S(\mathbf{q})} \right) \mathbf{u}^T \frac{\mathbf{p}-\mathbf{q}}{\|\mathbf{p}-\mathbf{q}\|} \, \mathrm{d}\mathbf{q} \right| \tag{2.1}$$

This formula could be bounded by studying the integral of the absolute value. However, we can obtain a tighter bound by separating the integral into two parts, the one where the

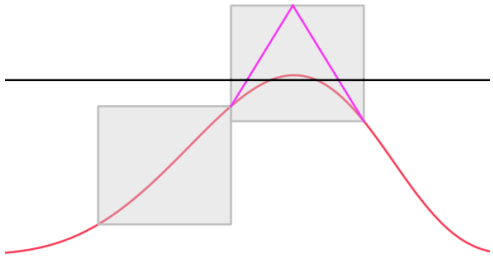


Figure 2.2: Ranges on two intervals, monotonous (left) and ambiguous (right).

integrand is positive and the one where it is negative, then taking the sub-integral with the largest absolute value. The segmentation is defined by a linear inequality $(\mathbf{p}-\mathbf{q})^T\mathbf{u} > 0$ (see Figure 2.3).

As described in [AZ21], we can compute the range $[s_0, s_1]$ of skeleton points which support intersects the ray, as well as the argmin of $\|\mathbf{p}-\mathbf{q}\|/\tau_S(\mathbf{q})$. Using this information, we can bound the first two terms within the integral. As τ_S appears in the denominator of the integrand, we bound it by the minimum of $\tau_S(s_0)$ and $\tau_S(s_1)$. For the term using k', we adopt a similar strategy as in [GGPP20]. We use knowledge on the kernel to select either the global bound of k' if it is in range :

$$\frac{n}{\sqrt{n-1}}\left(1-\frac{1}{n-1}\right)^{\frac{n}{2}-1}\frac{N}{\sigma},$$

otherwise, the maxima that is reached at one of the interval endpoints. Then, we only have to compute a bound on a simpler integral, e.g. for the positive domain:

$$\int_{\mathbf{q}\in S/(\mathbf{p}-\mathbf{q})^T\mathbf{u}>0} \mathbf{u}^T\frac{\mathbf{p}-\mathbf{q}}{\|\mathbf{p}-\mathbf{q}\|}\,\mathrm{d}\mathbf{q} \tag{2.2}$$

By bounding the positive and negative domains based on both the range of interest $[t_0, t_1]$ and the range $[s_0, s_1] \cap [0, 1]$(see Figure 2.3), it is possible to show that upper bound (positive domain) and lower bound (negative domain) can be computed by evaluating the integral with the newly defined larger domain at the end of the range (e.g., in t_0 and t_1 respectively).

We can then obtain the global bound by taking the maximum of both absolute values. While the integral admit a closed-form expression, the formula is numerically instable when the minimal distance between the ray line and segment line tend toward zero. We instead bound the integrand on the domain of interest which also has the advantage of being less costly to evaluate.

Both the per-primitive field bounds and the directional Lipschitz bound can be applied to other kernels whose derivative k' admit a unique maximum. For instance, among the kernels which admit closed-form expression of the integral f, Cauchy kernel defined in Equation 1.8 admits a unique maximum value for k' in $d = \frac{\sigma}{\sqrt{n+1}}$.

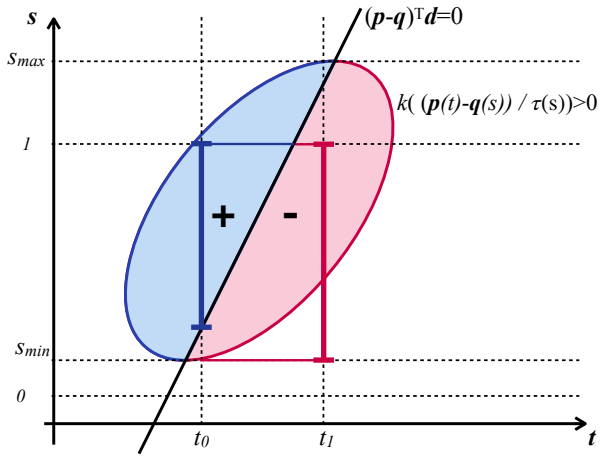


Figure 2.3: Clipping-space for ray/primitive intersection. The primitive is parametrized with $s \in [0, 1]$. Input range for Lipschitz bound query is $t \in [t_0, t_1]$. The support of a primitive can be divided into two halves based on the sign of $(\mathbf{p}-\mathbf{q})^T\mathbf{u}$. The blue half (resp. red) has a positive (resp. negative) contribution to the derivative.

Integral evaluation

We present here additional details on the derivation of the directional Lipschitz bound, more specifically on bound computation for the integral in Equation 2.2, where the line segment primitive S is parametrized by $\mathbf{q} = \mathbf{q}_0 + s\,\mathbf{v}$ and the ray is parametrized by $\mathbf{p} = \delta(t) = \mathbf{o} + t\,\mathbf{u}$.

Let's define $\mathbf{m} = \mathbf{o} - \mathbf{q}_0$, we then have:

$$\mathbf{u}^T \frac{\mathbf{p}-\mathbf{q}}{\|\mathbf{p}-\mathbf{q}\|} = \frac{\mathbf{m}^T\mathbf{u} + t - s * \mathbf{v}^T\mathbf{u}}{\sqrt{(\mathbf{m} + t\mathbf{u} - s * \mathbf{v})^T(\mathbf{m} + t\mathbf{u} - s * \mathbf{v})}} \tag{2.3}$$

which is a quotient between a linear function and the square root of a quadratic function. The denominator theoretically remains positive or null and can only cancel when the numerator cancels. The integral admits a closed-form expression, however the formula is numerically unstable when the minimal distance between the ray line and segment line tends toward zero (floating point value of the denominator can become negative resulting in an infinite value for the integral). We instead bound the integrand on the domain of interest which also has the advantage of being less costly to evaluate.

Local extrema in s of Equation (2.3) are reached in $\pm\infty$ and in

$$s_{argmax} = -\frac{\mathbf{v}^T(\mathbf{m} - \mathbf{m}^T\mathbf{u}\mathbf{u})}{\mathbf{v}^T(\mathbf{v} - \mathbf{v}^T\mathbf{u}\mathbf{u})}$$

we can therefore easily deduce an upper bound of the integrand in the range of interest (either the value reached at the argminimum if the latter is in range or the maximal absolute value of the integrand on the boundary of the range).

2.2 Slicing and rendering : ray processing

As an application, we present ray-processing for both transparent rendering and slicing of integral surfaces blended with summation blending, i.e. $f(\mathbf{p}) = \sum f_i(\mathbf{p})$. For slicing, we use the methodology described in [Lef13].

First, in order to process rays efficiently, we use both the same per-pixel linked-list of primitives as well as the ray segmentation procedure described in [AZ21]. The per-pixel linked list provides empty space skipping based on kernel supports without any field evaluation. A given ray is pre-segmented using the point where the minimal distance between a ray and line-segment is reached (e.g., where the kernel function is maximized). This segmentation allow having intervals close to the ideal behavior where they would be either monotonously increasing or decreasing.

During ray-processing, per-primitive interval ranges are combined with regular interval arithmetic to obtain the interval range of the global field. In all our examples, we use summation blending only. This can be extended to other blending operations as long as an interval inclusion can be found for them.

After ray segmentation, the core of the processing loop is similar to other interval methods such as [FPC10]. We check whether an intersection is possible, then either discard the interval or bisect until the required precision is reached (see Figure 2.4). Resolution used for slicing is the half the size of a voxel.

For transparent rendering implementation, we also use polynomial interpolation as soon as we have small enough intervals, this reduces the number of field evaluations in comparison to bisection.

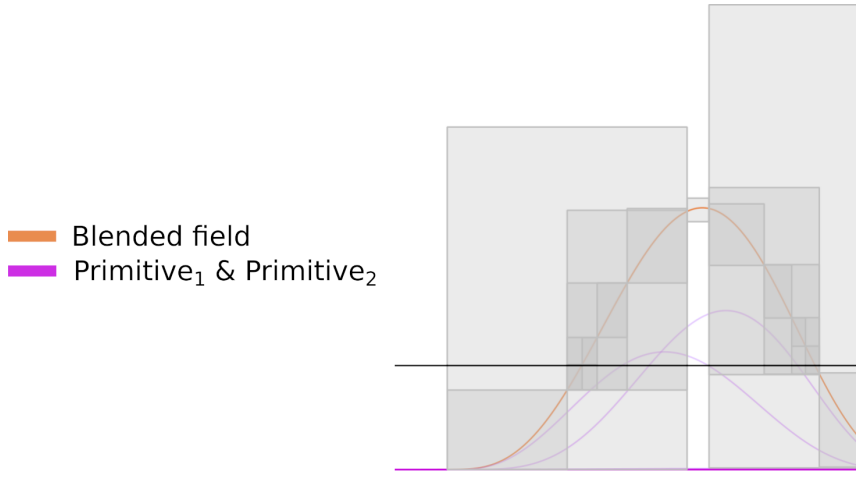


Figure 2.4: Progressive range computation for two primitives (purple) along a ray. The blended field (summation) is displayed in orange. The ray is initially segmented in three (one cut per-primitive).

2.3 Results

We compare both our bound computation to regular interval arithmetic methods (e.g. using directional derivative computed from closed form gradient for the Lipschitz bound). We observe large improvements in bound quality. In order to demonstrate the usefulness of our method, we present applications to both slicing and transparent rendering, both relying on the same ray-processing methodology.

Ray-tracing Our ray-tracing implementation was done in OpenGL. Our runtimes are measured on an NVIDIA GeForce GTX 1650 graphics card with a resolution of 700x700. The jellyfish model (Figure 2.1(right)), consisting of 154 line segment primitives, was rendered using an average of 431.4 ms per frame. The microstructure model (Figure 2.1(left)), consisting of 2895 primitives was rendered with an average of 1037.1 ms per frame.

Note that the memory requirement for the stack used in recursive processing is limited thanks to the initial ray segmentation. Depending on the application, computational time could be reduced by stopping the ray processing loop based on the level of opacity reached instead of capturing all transparent layers.

Slicing When using the method for slicing, we use the same acceleration structure as in [LLZ*21] with the notable exception that we create one linked-list of primitives per bucket of pixels instead of a single linked-list for the full space. This can dramatically impact runtime for complex skeletons sliced at high resolution. We compare our slicing runtime with the method of [LLZ*21] in Table 2.1. Comparisons were run on an Intel Corei7-8850H (2.60GHz, single-core used). Due to the bucketing strategy's major positive impact, we applied it to both methods. We observe larger runtime improvements when the resolution required along the ray direction increases. Indeed adding more slices does not require additional field evaluations in our case (only the generation of the voxel image from the computed roots is increased). Note that the ray direction is always the printing direction in our implementation, but this could be modified based on input models.

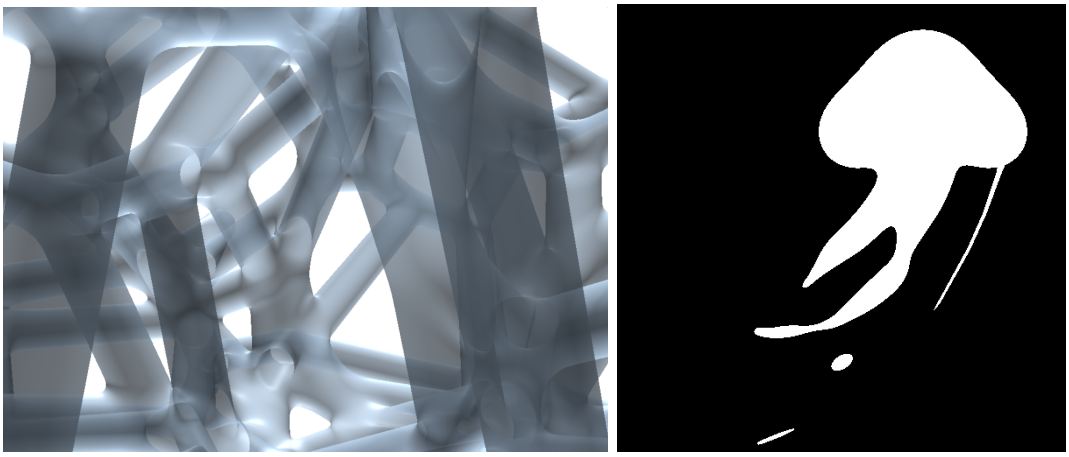


Figure 2.5: Detail on microstructure with smooth blending (left) and a slice produced for additive manufacturing (right).

Object	[LLZ*21]	[LLZ*21] Bucket 64^2	Our Bucket 64^2
Jellyfish	$27.2s$ / $55.4s$	$11.1s$ / $21.9s$	$2.9s$ / $3.5s$
Microstructure	$201s$ / $414s$	$5.1s$ / $10.3s$	$6.9s$ / $8.0s$

Table 2.1: Slicing models into respectively 512 and 1024 slices with an XY resolution of 512×512 voxels.

Our directional Lipschitz bound also allows the use of integral surfaces within the segment-tracing algorithm. From our experiments, bisection-based approach tends to be more efficient when computing all the roots (see Section 2.3.2).

2.3.1 Comparison to interval arithmetic on closed form gradient and field values

We have compared the sizes of the intervals calculated with our method to the ones calculated with interval arithmetic for both field values (figure 2.6) and directional derivatives (figure 2.7). We calculated the ratio by dividing the size of the interval range calculated by our method with the range calculated by interval arithmetic. We have done this calculation for several ray configurations with different interval widths (66184 configurations).

We produce improved bounds on most cases. For example for the field values we observe that 95% of the samples have 4 times improvement (65% for directional derivatives). The red lines denote where the ratio is equal to one (i.e. when they are equal). We can observe that our samples cluster on the left side of the line, approaching to zero. This is due to the fact

Object	[LLZ*21]	[GGPP20]	Our	Our with Segment Tracing bisection [GGPP20])
Jellyfish	$11.1s$ / $21.9s$	$4.8s$ / $5.3s$	$2.9s$ / $3.5s$	$3.1s$ / $3.7s$
Microstructure	$5.1s$ / $10.3s$	$12.9s$ / $16.2s$	$6.9s$ / $8.0s$	$6.9s$ / $7.9s$

Table 2.2: Slicing models into respectively 512 and 1024 slices with an XY resolution of 512×512 voxels. All methods use 64^2 buckets.

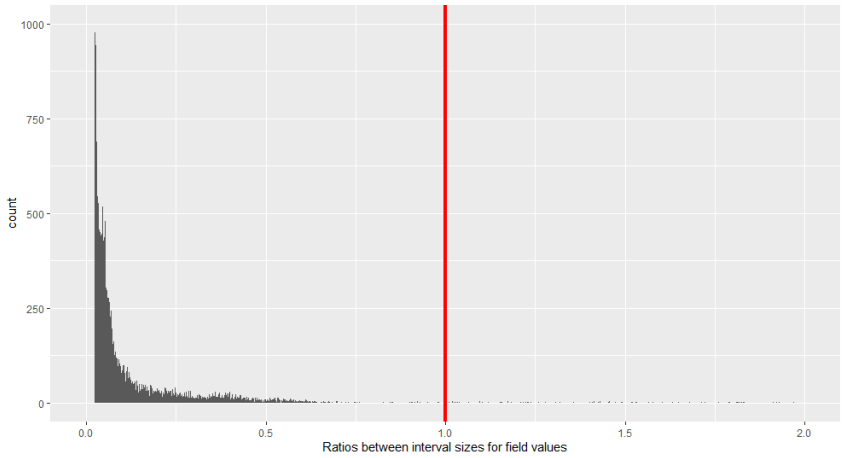


Figure 2.6: Histogram of the ratio of the sizes of intervals calculated with our method to the ones calculated with interval arithmetic for compact kernel for field values.

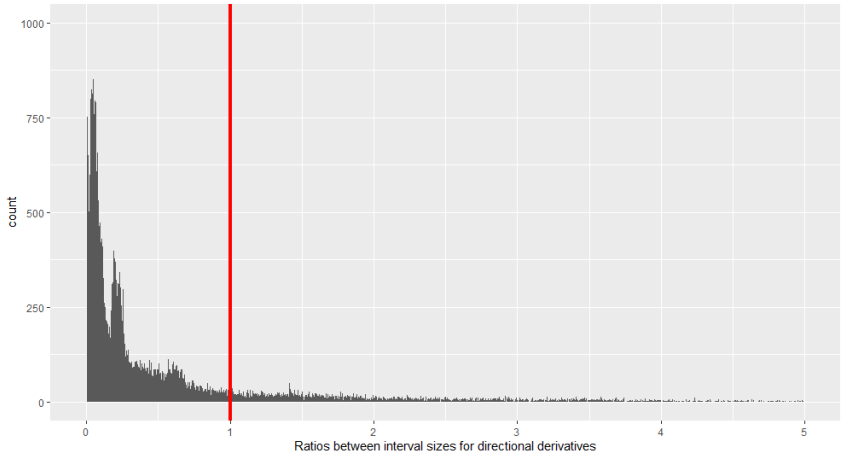


Figure 2.7: Histogram of the ratio of the sizes of intervals calculated with our method to the ones calculated with interval arithmetic for compact kernel for directional derivatives.

that interval arithmetic can create arbitrarily large or infinite bounds, therefore our division approaches zero.

2.3.2 Comparisons with local Lipschitz bounds

We provide two additional slicing comparisons, namely application to iterative segment tracing [GGPP20] and a combination of our method with segment tracing bisection (presented in [GGPP20]).

For all methods, we use a minimal interval/step size equal to half the size of a voxel. For the two variants of our method, we also filter the initial ray cuts (e.g. where the kernel function of individual primitives is maximized) in order to obtain a minimal size of 8 voxels for each initial interval where bisection will be computed.

Segment Tracing: In order to provide a fairer comparison, we implemented Segment Tracing with empty space skipping based on primitive supports (i.e. by relying on the same approach as our method). When applying Segment Tracing to transparency, the minimal step-size is important. The iso-surface itself being a fixed point in the iteration, we need the minimal step size to allow the iteration to go past the iso-surface itself. Without a minimal step size, it is also possible to run into an infinite loop (i.e. when using floating point, small enough steps can left the current ray parameter unchanged, hence never reaching the iso-surface).

Our method with Segment Tracing bisection: We can combine our method with the Segment Tracing bisection presented in [GGPP20], i.e. in addition to our strategy, we can also reduce the interval size using the directional Lipschitz bounds before applying a bisection step.

Runtimes are presented in Table 2.2. We can note that all interval arithmetic and Segment Tracing methods are less sensitive than [LLZ*21] to an increase in the number of slices. Number of steps per ray are displayed in Figure 2.8.

From our experiment, iterative Segment Tracing with integral surfaces is less efficient than our bisection-based approaches and it is also more sensitive to the minimal step size. However, it is important to note that derivation of tighter directional Lipschitz bound - provided the bounds are not too expensive to compute - is likely to make segment tracing more efficient than our method (the variant without the Segment Tracing bisection).

Finally, it is also important to note that we do not use combined evaluation of field and directional Lipschitz bound in our implementation. Doing so would further reduce the run-time of the methods using Segment Tracing (including ours with Segment Tracing bisection). Similarly, the number of field evaluations required by our method can be reduced : in our implementation, at each bisection step, we recompute field value at intervals end-points - resulting in three field evaluations per step - while this is only required for primitives with ambiguous field variation. Storing the field values independently at interval end-points for primitives with increasing and decreasing fields would largely reduce the required number of field evaluations while having a limited impact on the memory usage.

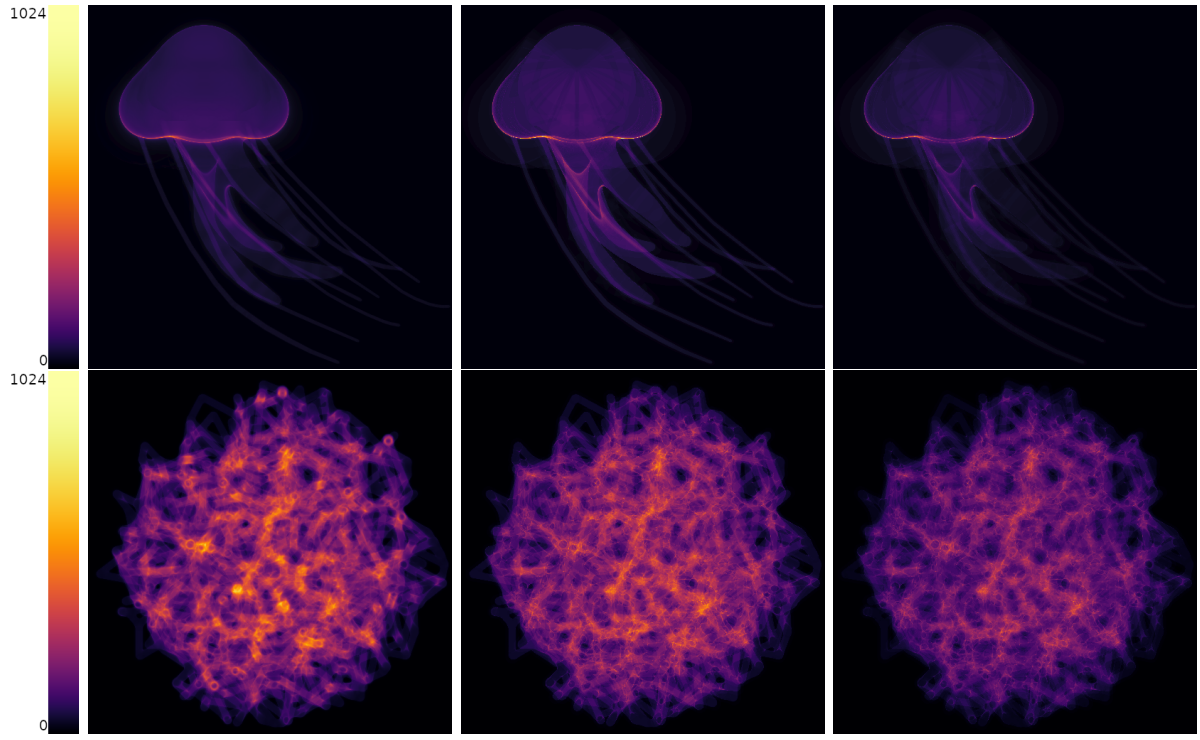


Figure 2.8: Number of field evaluation for 1024 slices with Segment Tracing (left), our method (middle), our method with Segment Tracing bisection (right). Note that both our methods require less steps (darker images).

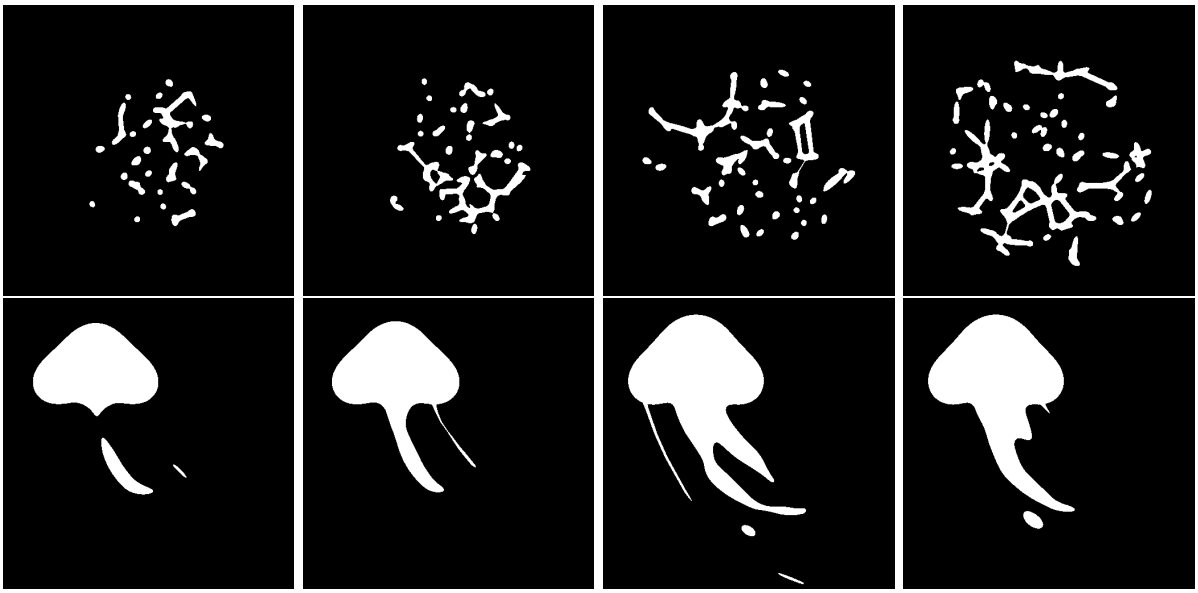


Figure 2.9: Slices produced for additive manufacturing.

2.3.3 Conclusion

We have presented a simple way to use scale-invariant integral surfaces with both the interval arithmetic framework and the segment-tracing algorithm. For interval arithmetic, we only rely on simple observation of the field behavior. A similar approach could also apply to a broader range of primitives. For the derivation of directional Lipschitz bound, we rely on careful analysis of the field integral to provide efficient bound computation. We provide applications of our methodology for both slicing and transparent rendering. Contrary to previously dedicated algorithms, the proposed approach would allow incorporation of integral surfaces in more general frameworks and allow their use with other representations or more complex blending.

While applied to scale-invariant integral surfaces, our methodology can also be applied to other integral surfaces with linearly varying weights. Several improvements are worth investigating, such as the adaptation to interval arithmetic queries on volumes and tighter bound computation. The latter requires a careful trade-off between the tightness of the computed bound and the required amount of computation to obtain them.

In the next chapter, we present a more general robust marching method that can be applied to arbitrary implicit surfaces.

3
Forward Inclusion Functions

The number of field function evaluations performed along the rays are the driving complexity factor with typically challenging areas such as grazing rays where the ray is close to tangent to the surface, producing multiple roots for the root finding problem. In addition, transparent rendering amplifies numerical robustness and complexity issues.

Ray-tracing methods can be divided into two main families of approaches: the self-validated numerical methods, such as Interval Arithmetic [Mit90] and Affine Arithmetic [dFS04] and Lipschitz methods, such as Sphere Tracing [Har96] and Segment Tracing [GGPP20].

Self-validated numerical methods construct bottom-to-top inclusion functions by bounding each elementary arithmetic operation of the field function and are used with recursive ray bisection to isolate the roots. Inclusion functions are designed to limit the maximal error on a ray sub-interval, a property well adapted to the ray subdivision approach. However, the bounds suffer from overestimation for complex expressions due to error accumulations.

For Lipschitz methods, the evaluation position is marched iteratively on the ray with a guaranteed intersection-free step size computed from a bound on the absolute value of the field derivative (either globally [Har96] or on a ray-subinterval to improve the processing of grazing rays [GGPP20]). However, due to the symmetrical nature of these bounds, they do not capture the field functions' local increasing/decreasing behavior. This results in a large number of steps – and field evaluations – in blending areas as the variations of primitive fields cannot compensate for each other, and for transparent rendering as the rays leave the surface slowly.

We propose to bridge the gap between these two families of approaches by considering the Lipschitz property as a *forward* inclusion function: an inclusion function that is exact at the beginning of the interval and, therefore, well suited for iterative ray processing. Based on this observation, we first propose using *asymmetric* bounds; better suited for processing blending areas and transparent objects. Then, we extend our forward asymmetric inclusion functions to *quadratic* forward bounds that can be built using either *derivative analysis* or a *bottom-up constructive* approach. Finally, we show that both approaches can be combined in the same framework. We emphasize the efficiency of the proposed approach – including a significant reduction in the number of processing steps and lesser sensitivity to ray-tracing hidden parameters (minimal step size and the maximum number of steps) – on various examples consisting of blobby surfaces and Hermite radial basis functions, including both opaque and transparent

rendering (Figure 3.1).

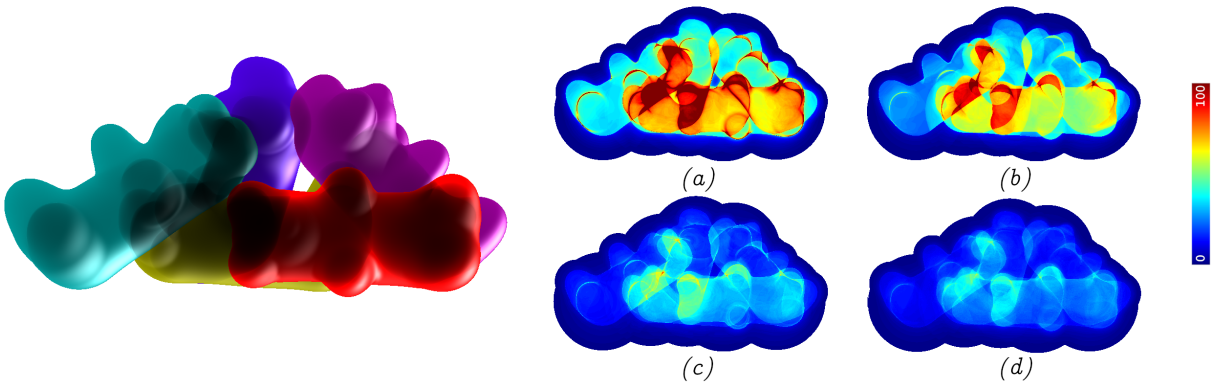


Figure 3.1: Top: Transparent rendering of blobby objects combined with set-theoretic union operation defined as max() function. Comparison of number of iteration steps for (a) Segment Tracing [GGPP20], (b) Linear mixed inclusion function, (c) Quadratic bottom-up inclusion function, (d) Quadratic mixed inclusion function. In all figures, color bar shows the total number of steps to converge to the root(s).

3.1 Overview

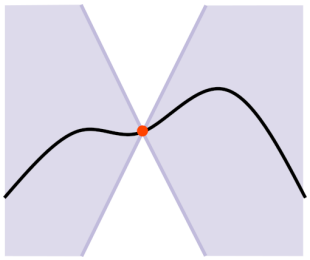


Figure 3.2: Geometric interpretation of the Lipschitz bound

Building on the previous methods, we start by emphasizing the inclusion property of Sphere Tracing [Har96]. The Lipschitz property provides us with a robust (up to numerical stability) inclusion function that can be visualized as a linear bound around the query point (Figure 3.2). Similarly, Segment Tracing [GGPP20] defines this inclusion function locally.

We first extend this idea to local *asymmetric* linear bounds, and then, to quadratic bounds. We then show two different ways of calculating these bounds: bottom-to-top inclusion functions and, more directly, by bounding the derivatives.

These two methods can be used together, bounding the directional derivatives when it gives tighter bounds, and using bottom-up inclusion functions where it is not possible to work with derivatives (e.g., when the functions are defined by part).

Keeping the values exact at the interval starting point (Figure 3.3), it is possible to use these bounds in an iterative ray-marching algorithm for finding ray-surface intersections.

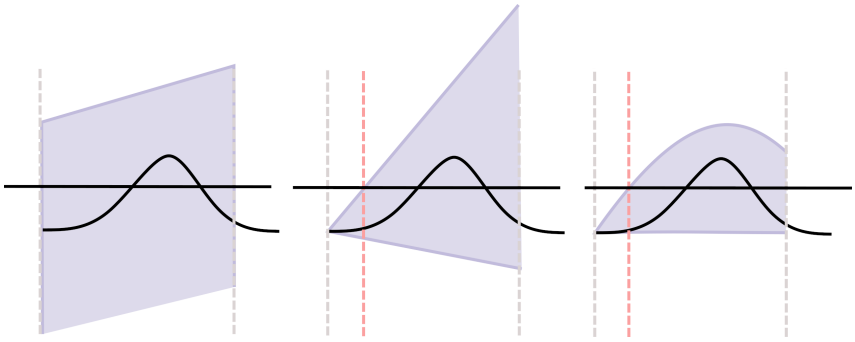


Figure 3.3: Behaviour of Revised Affine Arithmetic [FPC10] (left), asymmetric linear (middle) and quadratic (right) inclusion function. The latter two always allow to discard a subpart of the ray and quadratic bounds are more likely to discard the full interval.

We define our *forward* inclusion functions to be exact at the query point t_0, the starting point of the interval $[t_0, t_1]$:

$$f(t_0) = f_-(t_0) = f_+(t_0) \tag{3.1}$$

Since our contribution focuses on 1-dimensional ray processing, for the sake of brevity, we denote $f \circ \delta(t)$ as $f(t)$ and $\nabla_{\mathbf{d}} f \circ \delta(t)$ as $f'(t)$ in the remainder of this chapter.

3.1.1 Validity Intervals

When combining inclusion functions, some inputs or intermediate results can cause invalid results, such as negative values in logarithms or square roots and division by zero. In the previous self-validated numerical methods, these were declared *infinite* bounds and worked out by bisection; as the intervals get smaller, the degenerate cases would resolve. Since we march on the inclusion functions instead of bisecting them, we introduce validity intervals. Whenever an invalid result is encountered, the interval is shortened automatically during the evaluation of the inclusion function up to a point where the results are valid. In the subsequent operations of the inclusion function evaluation, this new shortened interval is used.

3.1.2 Ray Surface Intersection

Similar to the previous safe marching methods, asymmetric linear and quadratic inclusion functions can be marched safely provided that the inclusion function has the same value as the field function at the starting point of the interval (i.e., it is a *forward* inclusion function). This is illustrated in Algorithm 3, which follows the same overall strategy as Segment Tracing [GGPP20]. The step size to advance along the ray is defined by the x-intercept of the upper or lower bound of the inclusion function, depending on whether the query point is inside or outside the surface. As inclusion functions must be constructed on some input interval, an heuristic is used to compute the length of the next interval to process. In practice, we choose the next interval length to be twice the last safe step size (i.e. $k = 2$). In our case, the inclusion functions can also shorten their range of validity whenever an invalid operation is encountered during the calculations (Section 3.1.1). This algorithm, combined with asymmetric local bounds, provides a straight-forward and efficient marching for grazing rays and transparent rendering. The behaviour of different methods as forward inclusion functions can be seen in Figure 3.5.

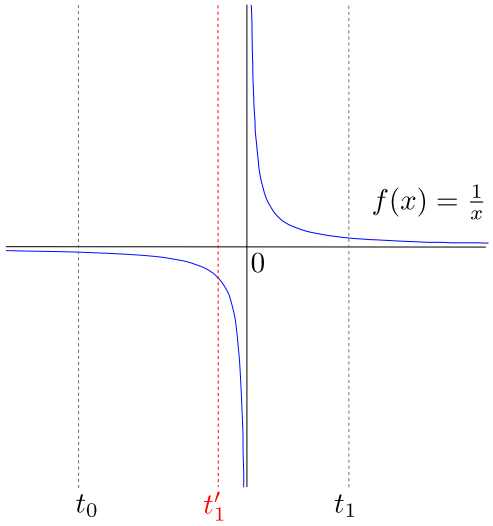


Figure 3.4: Shortening the interval in case of division by zero. The subsequent operations in the field expression are going to be evaluated on the new interval $[t_0, t_1']$.

Algorithm 3 Ray Marching Algortihm

```
procedure INTERSECT(Ray, Interval [t_0, t_1])
    t = t_0
    t_end = t_1
    while t < t_1 do
        eval = f(o + td)
        if eval < eps then
            root ← t                                   ▷ A root is registered
        end if
        t_end, Inclusion = CalculateInclusion(f, Ray, [t, t_end])
                                   ▷ Inclusion calculation can shorten the interval
        r = Inclusion.CalcNextRoot()
        if r ∈ [t, t_end] then
            stepsize = r − t
        else
            stepsize = t_end − t                       ▷ No roots in the interval, skip
        end if
        t += stepsize;
        t_end = min(t + k ∗ stepsize, t_1);
                                   ▷ Interval size based on the previous step
    end while
end procedure
```

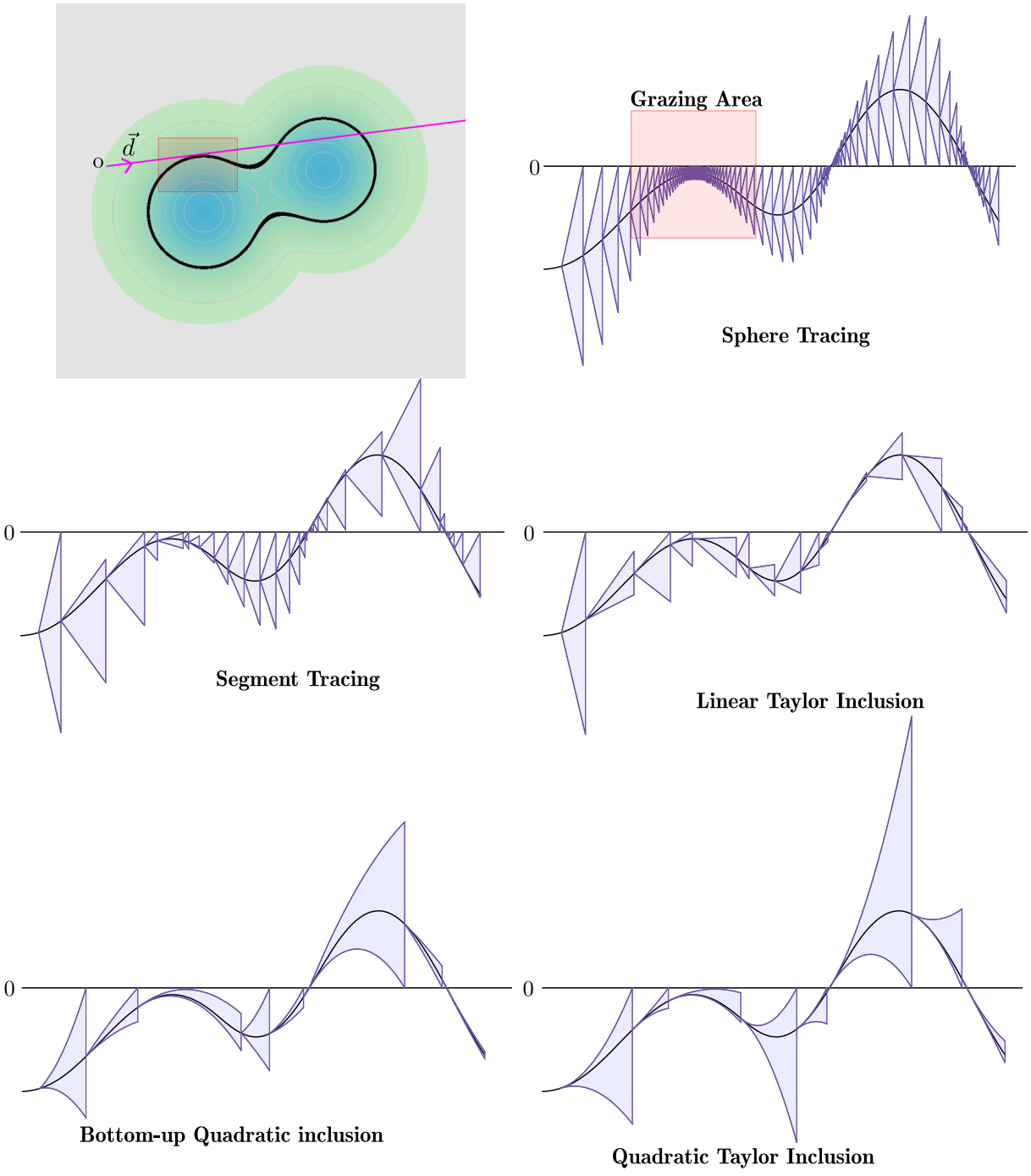


Figure 3.5: Behaviour of Sphere Tracing [Har96], Segment Tracing [GGPP20], our asymmetrical linear tracing, and quadratic tracing with bottom-up quadratic inclusion function and quadratic Taylor inclusion function along a near grazing ray with transparency.

3.2 Linear Tracing

We define *forward linear inclusion functions* as a pair of bounding lines that have the same value as the field function at the starting point of the queried interval.

Considered as a linear inclusion function (Figure 3.2), Sphere Tracing [Har96] bounds can be written as:

$$\begin{aligned}[f](t) &= f(t_0) \pm m(t - t_0) \ , \quad \text{where} \\ m &= m_{global} = \max_{\mathbf{p} \in \mathbb{R}^3} (\|\nabla f(\mathbf{p})\|) = L\end{aligned} \tag{3.2}$$

When the first point of a query interval is outside (or inside) of the implicit volume, computing the root of the upper (or lower) bound allows performing the same marching step as with the previous formulation (see Equation 2.5).

Similarly, we can formulate the local directional Lipschitz bounds of Segment Tracing [GGPP20] as a forward linear inclusion function with the following formula:

$$m = m_{local} = \max_{t \in [t_0, t_1]} (|f'(t)|) = \lambda, \quad t \in [t_0, t_1]. \tag{3.3}$$

In this form, we can see that such bounds can be easily extended to asymmetric bounds as follows:

$$\begin{aligned}f_-(t) &= f(t_0) + m_-(t - t_0) \\ f_+(t) &= f(t_0) + m_+(t - t_0), \qquad t \in [t_0, t_1],\end{aligned} \tag{3.4}$$

We will study two alternative ways to compute $m_\pm$: first, by bounding the derivatives, and second, with bottom-up inclusions on the arithmetic operations.

3.2.1 Linear Taylor Inclusion Function

The local Lipschitz constant of a differentiable function can be obtained by studying its first derivative:

$$\begin{aligned}m = \lambda &= f'(\xi), \\ \text{where } \xi &= \arg\max_{t \in [t_0, t_1]} (|f'(t)|)\end{aligned} \tag{3.5}$$

on an interval. Similarly, we can define our asymmetric local maximum and minimum bounds as:

$$\begin{aligned}m_\pm &= f'(\xi_\pm) \\ \text{where } \xi_- &= \operatorname*{argmin}_{t \in [t_0, t_1]} \left(f'(t)\right) \text{ for } f_- \\ \text{and } \xi_+ &= \arg\max_{t \in [t_0, t_1]} \left(f'(t)\right) \text{ for } f_+\end{aligned} \tag{3.6}$$

or equivalently:

$$\begin{aligned}m_- &= \min_{t \in [t_0, t_1]} \left(f'(t)\right) \\ m_+ &= \max_{t \in [t_0, t_1]} \left(f'(t)\right)\end{aligned} \tag{3.7}$$

Due to the close relation of this formulation to local Taylor expansion at $t = t_0$, we call this approach *Linear Taylor Inclusion* in the remainder of this chapter. Depending on the field function, the exact values of $m_\pm$ can be derived analytically. In practice, we rely on the approach proposed in [GGPP20] except considering the sign of the directional derivative of the distance function in the bound derivation. Details are provided in 3.4.1.

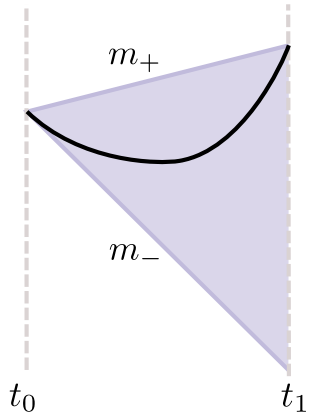


Figure 3.6: Linear inclusion functions for a convex function.

3.2.2 Bottom-up Linear Inclusion Function

In cases where the bounds on the derivative are challenging to calculate – or not defined in case of functions defined by parts (e.g., max operation) – we build the linear inclusion functions using the same strategy as previous self-validated numerical methods; bounds are computed bottom-up by using properties of the arithmetic operations constituting the field function defined along the ray.

Rules for summation, subtraction, and scalar multiplication of forward linear inclusion functions are trivially defined. For instance, for the summation, we have:

$$[f+g](t) = (f(t_0) + g(t_0)) + (m_{f,\pm} + m_{g,\pm})(t - t_0) \tag{3.8}$$

For non-linear operations, we introduce linear inclusion functions exploring the properties of each operation. A first common case is the squared distance to primitives (see Equation 1.6-3.12), which is a quadratic function for point primitives.

Convex/Concave functions: All convex/concave functions of the ray parameter – including quadratic functions – can be handled in the same way. This special case can then be used as a base building block for operations such as multiplications.

For a convex function, we define the forward linear inclusion functions using the derivative at the interval starting point for the lower slope m_- and the line interpolating the values at interval endpoints for the upper slope m_+ (Figure 3.6). This leads to the following formulas

$$\begin{aligned} m_-(t) &= f'(t_0) \\ m_+(t) &= \frac{f(t_1) - f(t_0)}{t_1 - t_0} \end{aligned} \tag{3.9}$$

For the forward inclusion functions, i.e., when the value at the interval starting point is equal to the function value, this provides the tightest possible linear inclusion function for the interval. As the interval size gets smaller, the bounds get closer to the function, reducing the error. Inclusion functions for concave functions are obtained by simply swapping the definition of the upper and lower bound in Equation 3.9.

Multiplication Given two forward linear inclusion functions, all pairwise multiplications of linear functions defining the inclusion produce four quadratics: f_+g_+, f_+g_-, f_-g_+ and, f_-g_-. We can then find linear bounds on each of them separately. By construction, all quadratics share the same value at the beginning of the range. Then, the values of m_- and m_+ are simply defined as the highest and lowest slopes of all four bounds.

Maximum Building the maximum operation on two forward linear inclusion functions is similar. For the upper bound, the value at the beginning of the range t_0 is simply the maximum of the two input inclusion functions in t_0, and for the interval end-point, it is possible to use the maximum values of the two upper bounds. For the lower bound, the lower linear bound with the maximum initial value is chosen.

Once we define the inclusion functions for the basic operations, we can combine them to build the field functions. As in the previous inclusion methods, after combining the operations, the bounds may lose their tightest possible bound property, still providing the guarantee to contain the function reliably. To reduce the overall error in the inclusion functions, we introduce quadratic inclusion functions.

3.3 Quadratic Tracing

Quadratic inclusion functions can provide better approximation of the field functions as they have more degrees of freedom than linear ones. Therefore they can get closer to the field function along the ray and reduce the error; and converge faster to the root during ray marching. Moreover, due to the shape of the bounds, empty areas are skipped more quickly, which is especially advantageous for grazing rays.

Similar to local asymmetric linear inclusion function, we can define quadratic inclusion function that are exact at the interval starting point as two quadratics bounds given in Horner notation:

$$[f](t) = f(t_0) + (t - t_0)\left(a_\pm + b_\pm(t - t_0)\right), \; t \in [t_0, t_1], \; a_\pm, b_\pm \in \mathbb{R}. \tag{3.10}$$

As for our linear inclusion functions, we propose two strategies to derive parameters $a_\pm$ and $b_\pm$: directly, by bounding the derivatives, and bottom-up construction.

3.3.1 Quadratic Taylor Inclusion Function

Similar to linear inclusion functions, we can bound the higher degree derivatives to get an inclusion function.

If the function is twice differentiable, and the bounds on the second derivative are easily found, using the Taylor expansion at $t = t_0$, we can define the quadratic inclusion function as:

$$\begin{aligned}[f](t) = f(t_0) + (t - t_0)\left(f'(t_0) + \frac{1}{2}f''(\xi_\pm)(t - t_0)\right), \\ \text{where } \xi_- = \operatorname*{argmin}_{t\in[t_0,t_1]}\left(f''(t)\right) \text{ for } f_- \\ \text{and } \xi_+ = \operatorname*{arg\,max}_{t\in[t_0,t_1]}\left(f''(t)\right) \text{ for } f_+\end{aligned} \tag{3.11}$$

keeping both the function and derivative values fixed at t_0. Derivation for the point primitives are provided in 3.4.1.

3.3.2 Bottom-up Quadratic Inclusion Function

Similarly to the bottom-up linear inclusion functions, we define the quadratic inclusion functions for each arithmetic operation we need for the field function calculations along a given ray. Basic arithmetic operations are again trivial for summation, subtraction and scalar multiplication.

For non-linear operations, we rely heavily on local convexity/concavity. We present here the case of the multiplication. The general case for convex/concave functions is discussed in 3.4.2 and the maximum operation for two quadratic inclusion functions is discussed in 3.4.3.

Quadratic Multiplication A quadratic forward inclusion function can be formulated as the integral of linear bounds of a function derivative around the query point:

$$[f](t) = f(t_0) + \int_{t_0}^{t} L_{\pm}(t).$$

We use this approach to bound the multiplication between two quadratic inclusion functions $[f]$ and $[g]$.

Multiplication between bounds $f_{\pm}$ and $g_{\pm}$ define four quartic polynomials f_+g_+, f_+g_-, f_-g_+, and f_-g_-; and their derivatives are cubic polynomials, which are simpler to analyze since they can be split into convex and concave parts. For a given cubic $(f_{\pm}g_{\pm})'$, the inflection point defining the two parts can be calculated directly by solving a linear equation. Once we identify the convex and concave parts, we can linearly bound them as described in Section 3.2.2 and Figure 3.7 (left).

We can then generate a safe linear inclusion function using the highest and lowest possible points of each linear bound at the interval end-points – see Figure 3.7 (right) – which can then be integrated to obtain the quadratic inclusion function of the multiplication operator.

Per-Primitive Bounds

When using compactly supported kernels (Equation 1.7), the smoothing function is defined by parts. When the query interval includes the clipped part, we calculate the inclusion functions only on the unclipped part to get the appropriate bounds. Then, we can either extend this over the entire interval or, as illustrated in Figure 3.8 (top), modify this quadratic bound without recalculating any additional field function values, and achieve a larger iteration step size.

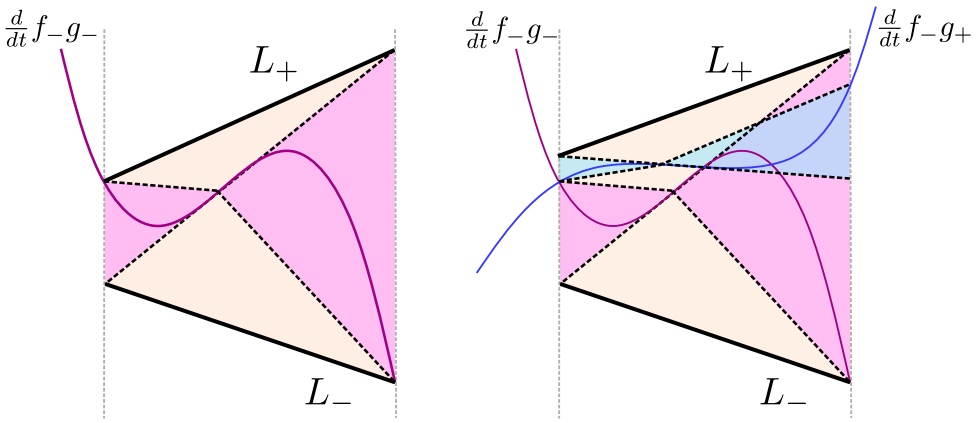


Figure 3.7: Multiplication of two quadratics produces a quartic. We can bound its derivative using convex/concave regions (left). When multiplying two quadratic inclusion functions, four quartics are produced. They can be linearly bounded the same way and these bounds can be combined to get the overall bound for the multiplication operation. The illustration for two curves is shown on the right.

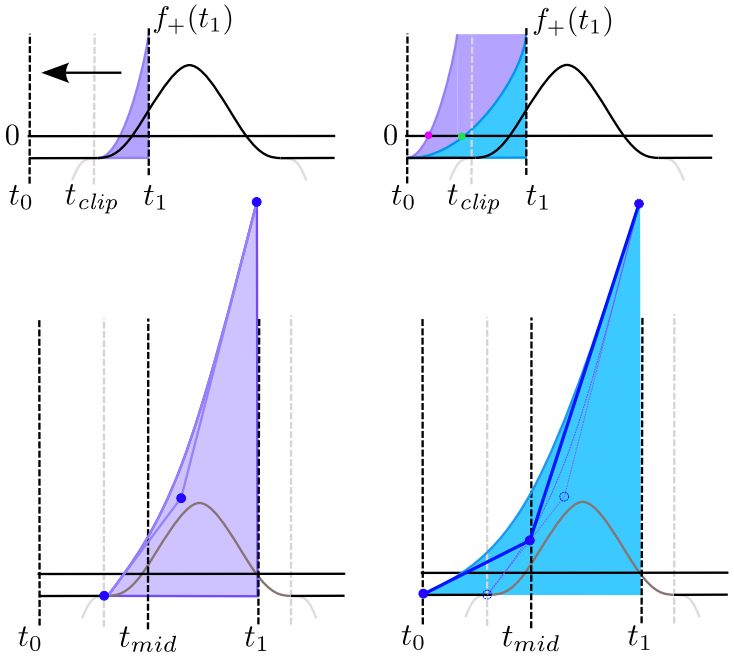


Figure 3.8: Per-primitive bound extension for compact kernels: computing the bound on the intersection of the kernel support and ray interval, then extending it to the whole interval improves the step-size (top). For this, we manipulate the control points defining the quadratic and move the initial query point to the interval start point. Carrying the middle control point as well guarantees that the resulting bounds are still valid after the extension.

3.4 Derivation of bounds

3.4.1 Linear and Quadratic Taylor Bounds

To calculate the local Taylor inclusion functions for point primitives, we analyze the first and second derivatives for Equation 3.12. For linear Taylor inclusion functions, as described for Segment Tracing [GGPP20], the linear bounds are calculated by bounding the gradient of the distance and derivative of the kernel along the ray separately.

$$f(t) = k \circ \ d \circ \delta(t) \tag{3.12}$$

$$f'(t) = \left(k' \circ d \circ \delta(t)\right) \left(\nabla_{\mathbf{u}} d \circ \delta(t)\right) \tag{3.13}$$

Since the gradient of the distance function along the ray is monotonous (Figure 3.9 left), the maximum and minimum values occur at the interval end points. For the derivative of kernel as a function of the distance d (Figure 3.9 right), the maximum and the minimum are reached either at the interval end points or at the extrema. For calculating the bounds on $f'(t)$, we analyze the product of these extremal points. This way, these bounds can be extended into more general primitives only by changing the distance function.

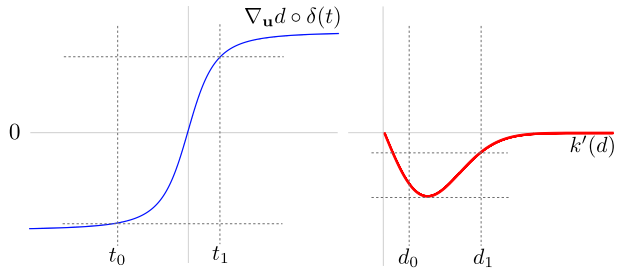


Figure 3.9: Derivative analysis for the point primitive (3.13). By analyzing the maximum and minimum values of the multiplication of the two values in a given interval, the minimum and maximum bounds on the derivative of the field equation can be calculated.

The bounds for second derivatives are calculated directly by analyzing the roots of the third derivatives to localize the extrema in a given interval. By examining the existence of the extrema in a given interval, it is possible to locate the minimum and maximum values of the second derivatives of the field function at either interval end-points or at the extrema. (Figure 3.10). For the Gaussian case, for a general quadratic given in the form:

$$ax^2 + bx + c \quad a, b, c \in \mathbb{R}, \tag{3.14}$$

its third derivative can be found as:

$$\frac{d^3}{dx^3}\left(e^{ax^2+bx+c}\right) = (2ax + b)(4a^2x^2 + 4abx + b^2 + 6a)e^{ax^2+bx+c} \tag{3.15}$$

and the three roots that correspond to the maximum and minimum values of the second derivative can be calculated directly.

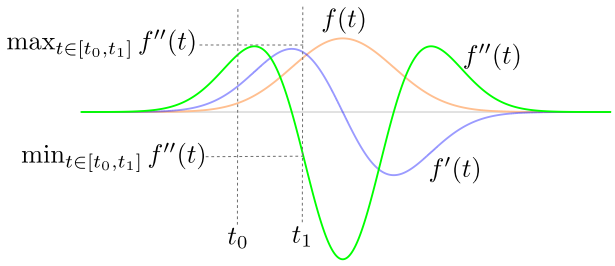


Figure 3.10: Equation 1.7 with Gaussian kernel (orange), first derivative (blue), second derivative (green). The behaviour of extrema is the same for the compact kernel (1.7) within the support.

3.4.2 Quadratic inclusion function for monotonous convex/concave functions

We present here the computation of quadratic inclusion function for a monotonously decreasing function f and a convex input quadratic Q as illustrated in Figure 3.11 with an exponential function ($f(x) = e^{-x}$). We rely on the fact that the composition of a quadratic and linear function is a quadratic function. We therefore build a linear inclusion function for the function f on a range of interest defined by the input quadratic Q (see Figure 3.11). Let's first assume that the value of the quadratic Q at the interval start point is an extrema on the interval $[t_0, t_1]$. Given the extrema I_0 and I_1 of the quadratic, since the exponential f is convex, linear bounds can be constructed for f on the interval $[I_0, I_1]$ as described in Section 3.2.2. These linear bounds L_- and L_+ decrease monotonously as the exponential e^{-x} they are bounding. The composition of those bounds with the quardatic Q gives an upper and lower bounds on $[t_0, t_1]$.

This strategy cannot be applied when the start point is not an extrema of Q as the linear bound of f would not be exact at the interval starting point. In this case, the interval is shortened up to t_1' the symmetric point where $Q(t_0) = Q(t_1')$ as shown in Figure 3.11 (top right). This strategy can cause smaller steps as seen in molecule rendering (see Section 3.5.1).

We explored alternative strategies as shown in Figure 3.12: we either relax the value at the global extrema of Q or at the interval starting point. In our experiments, those strategies provide similar runtimes (Table 3.2).

When we start with a quadratic range, instead of a single quadratic, we simply perform the above mentioned steps separately for each quadratic and use their upper and lower bounds respectively.

3.4.3 Quadratic inclusion function for maximum

The maximum of two quadratic inclusion functions is calculated by finding the maximum of two upper and lower quadratic bounds separately. A sketch is given for a maximum bound in Figure 3.13. In this case, we build the quadratic with a control polygon consisting of three points: an initial fixed point matching the maximal value of the two input bounds at the beginning of the interval, a middle point computed from the highest extent of the two derivatives calculated at the start point, and the maximum value at the interval end point. For the lower bounds, we simply pick the lower bound with the highest value at the interval starting point.

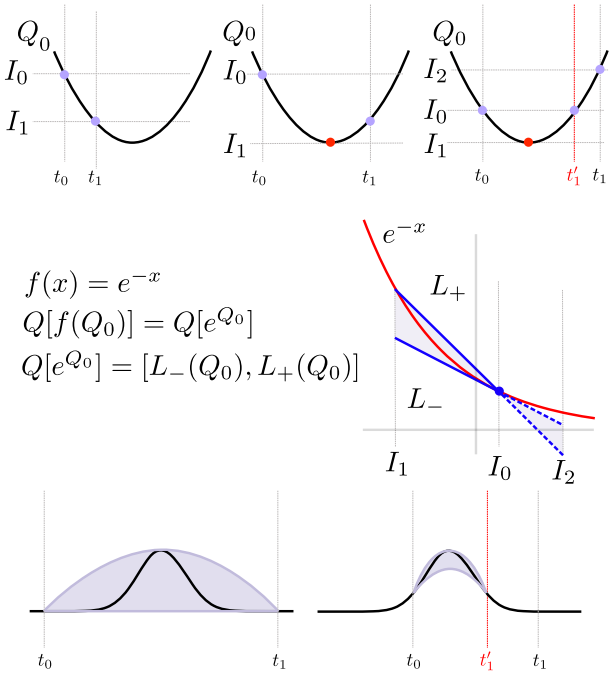


Figure 3.11: Quadratic inclusion function for a monotonous convex function with a validity interval. The terms I_0 and I_1 denote the range of values the quadratic can reach in the given interval. If this prevents keeping the value at the interval start fixed, the interval is shortened.

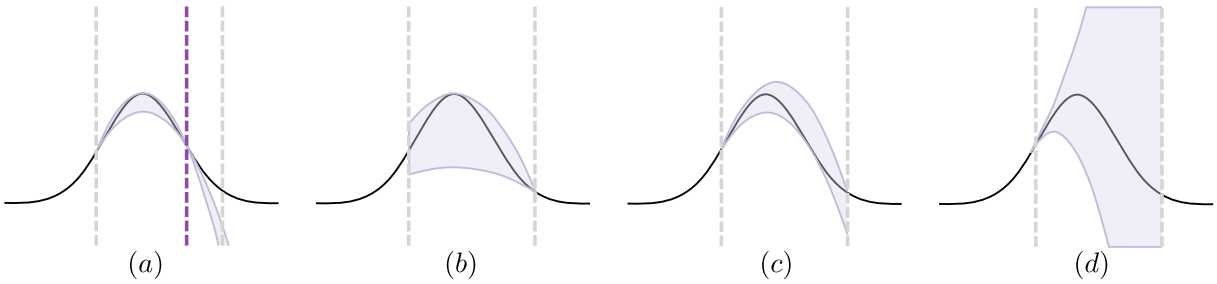


Figure 3.12: Different approaches we have used for calculating the bound on the Gaussian kernel. (a) Validity intervals, (b) relaxing at the query point, (c) relaxing the maxima, (d) quadratic Taylor inclusion function.

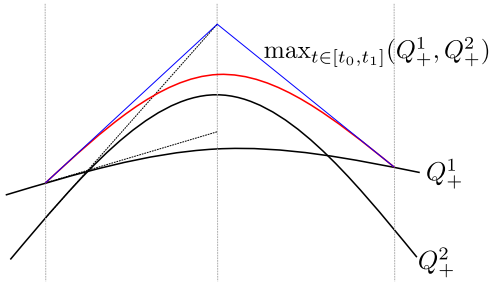


Figure 3.13: Maximum of two upper quadratic bounds

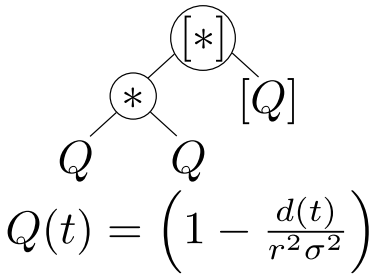


Figure 3.14: Building the inclusion for one polynomial blob.

3.5 Results

We compare our methods with the methods described in Segment Tracing [GGPP20], Revised Affine Arithmetic [FPC10] and molecular rendering [Bru19] both in terms of the number of steps and processing time. We provide run-times for both CPU and GPU computations. Our CPU run-times are obtained with a C++ implementation; this configuration is used for all figures and tables if not specified otherwise. GPU run-times for molecular rendering [Bru19] are obtained using a modified version of the author's implementation available at `https://github.com/sbruckner/dynamol`. GPU run-times for transparency (see Figure 3.20) on an animated scene are obtained using a GLSL fragment shader. All CPU run-times are obtained on a Intel® Core i7-10750H, and GPU run-times are obtained on an NVIDIA GeForce® GTX 1650. We discuss opaque rendering in section 3.5.1 (including molecular rendering) and transparent rendering in section 3.5.2.

We show our results on point skeletons with the two fall-off kernels described in Section 1, the Gaussian and the compact kernel.

For compact kernels (see Equation 1.7) with $n = 6$, linear and quadratic inclusion functions for the field function along the ray for a single blob are calculated using two subsequent multiplications (see Figure 3.14). The resulting bounds are then summed for blending. For the Gaussian kernels, we use the convexity property of the exponential function to construct the inclusion functions. The details are provided in 3.4.2.
We have defined the surfaces in Figures 3.1 and 3.21 with respectively a sharp union (max operation) and a summation blend to combine the base shapes.

$$f(t) = \max_n \left(\sum_m Blob_{mn}(t) \right) \tag{3.16}$$

For Hermite Radial Basis Implicits (HRBFs) defined as [MGV11]:

$$f(\mathbf{p}) = \sum_i \alpha_i \phi(\|\mathbf{p} - \mathbf{p}_i\|) + \beta_i^T \nabla \phi(\|\mathbf{p} - \mathbf{p}_i\|), \tag{3.17}$$

where ϕ is a smooth radial basis function and α_i and β_i are weights computed to approximate a set of points p_i with prescribed normals. We use $\phi(x) = x^3$, which results in a field that is neither polynomial nor a distance field. Comparison with Revised Affine Arithmetic [FPC10] is given in Figure 3.17 and show a significant reduction in number of steps.

3.5.1 Opaque rendering

In Figure 3.15, we show how our proposed methods compare with the state-of-art methods and with each other. In (b) and (c), we show that using the asymmetrical version of the same

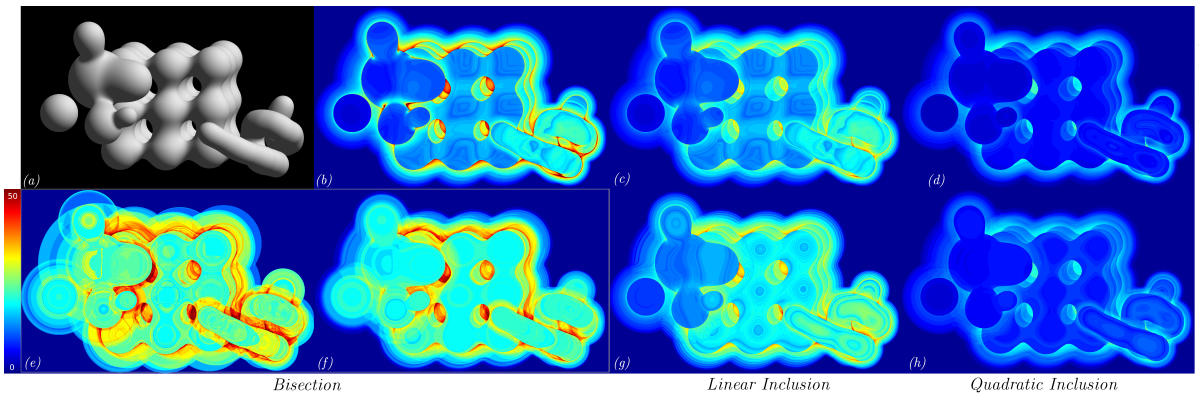


Figure 3.15: An example scene (a) and comparison between the number of steps for (b) Segment Tracing [GGPP20], (c) linear Taylor inclusion function, (d) quadratic Taylor inclusion function, (e) Revised Affine Arithmetic [FPC10], (f) bottom-up linear inclusion function with bisection, (g) iterative bottom-up linear inclusion function and (h) bottom-up quadratic inclusion function. Improvement is seen in grazing rays and blending areas.

Table 3.1: Comparison of average calculation time and the number of steps per ray between different methods for the scene in Figure 3.15 for 11498388 rays around the bounding box.

$\sim$ 11M Bounding Box Rays	Avg. time per ray (μs)	Avg. # of steps	Gain (time)
Segment Tracing [GGPP20]	19.86	8.90	-
Revised Affine Arithmetic [FPC10]	41.56	15.40	-109.3%
Bottom-up Linear Bisection	27.18	12.55	-36.9%
Linear Taylor	19.98	8.11	-0.6%
Bottom-up Linear	20.91	9.62	-5.3%
Bottom-up Quadratic	14.22	5.56	28.4%
Quadratic Taylor	11.48	4.79	42.2%

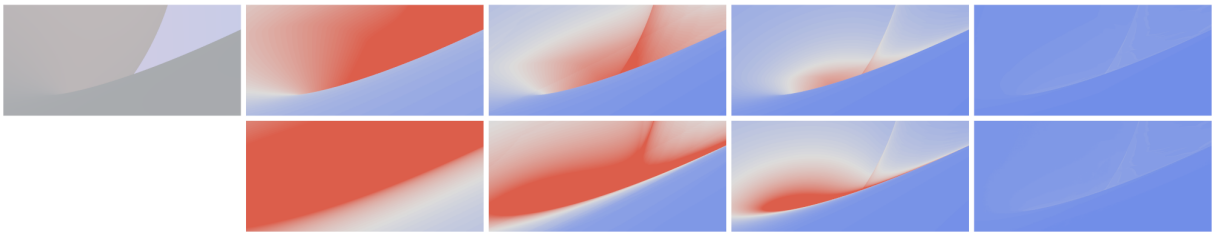


Figure 3.16: Close-up in a challenging area. Top row use a minimal step size of 0.1 (used in all examples) and bottom use 0.01. From left to right: Sphere, Segment, Linear and Quadratic Taylor tracing - number of step is clamped to 75 for display. Quadratic tracing is less sensible to minimal step size.

kind of linear bounds (Section 3.2.1) provides noticeable improvement for blended areas and grazing rays. Our linear bottom-up inclusion provides improvement compared to Revised Affine Arithmetic [FPC10] when roots are found using bisection with interval pruning (in (e) and (f)). Further improvement is shown when rays are processed iteratively using the same bottom-up linear inclusion functions (in (g)). Quadratic Taylor and bottom-up inclusion functions show the most improvement (in (d) and (h)).

Overall we demonstrate that Taylor/Lipschitz methods require fewer steps to converge to the surface, than bottom-up inclusion function calculations. The average number of field function evaluations and average time per ray is given in Table 3.1. Despite the increased number of computations per step, quadratic bottom-up inclusions for ray processing improve run-times over linear inclusion functions. The greatest run-time improvements are observed for the Quadratic Taylor inclusion functions, which is expected as they require less computation per steps compared to bottom-up inclusion functions.

Finally, our quadratic bounds provides the best benefits in challenging areas as shown in Figure 3.16.

Molecules We have compared our method with the recent molecule rendering method [Bru19], where molecules are represented with density fields using Gaussian kernels (see Figure 3.19). Their method uses a highly specialized field inversion to transform the density field into a weak SDF. This allows efficient use of the sphere tracing algorithm to visualize scenes with a large number of molecules. The field inversion removes the inflection points that are due to the kernel shape (see Figure 2.5). This is especially useful for rays that are directed directly toward primitives center as visible in the close-up of Figure 3.18. In those areas, field inversion [Bru19] provides the smallest number of steps, however grazing rays are problematic. As our methods and Segment Tracing work directly with the density field, they provide a different trade-off: improving grazing rays at the cost of direct rays. Segment-tracing results in increased run-time (see comparisons in Table 3.2) but is less sensitive to the limit on number of steps used in the algorithm. Our experiments show that our quadratic inclusion functions are on par with the field inversion technique [Bru19] – or faster depending the variant used – and do not suffer from increased limit on number of steps.

Our base strategy for quadratic inclusion function on exponential function uses validity intervals as explained in 3.4.2. This cause many seemingly arbitrary jumps in the number of steps in neighboring regions as visible in Figure 3.18 (bottom right).

This would require further inspection regarding the effects of nearby molecules, which may have a very small influence on the final field value but impose unnecessary interval subdivisions.

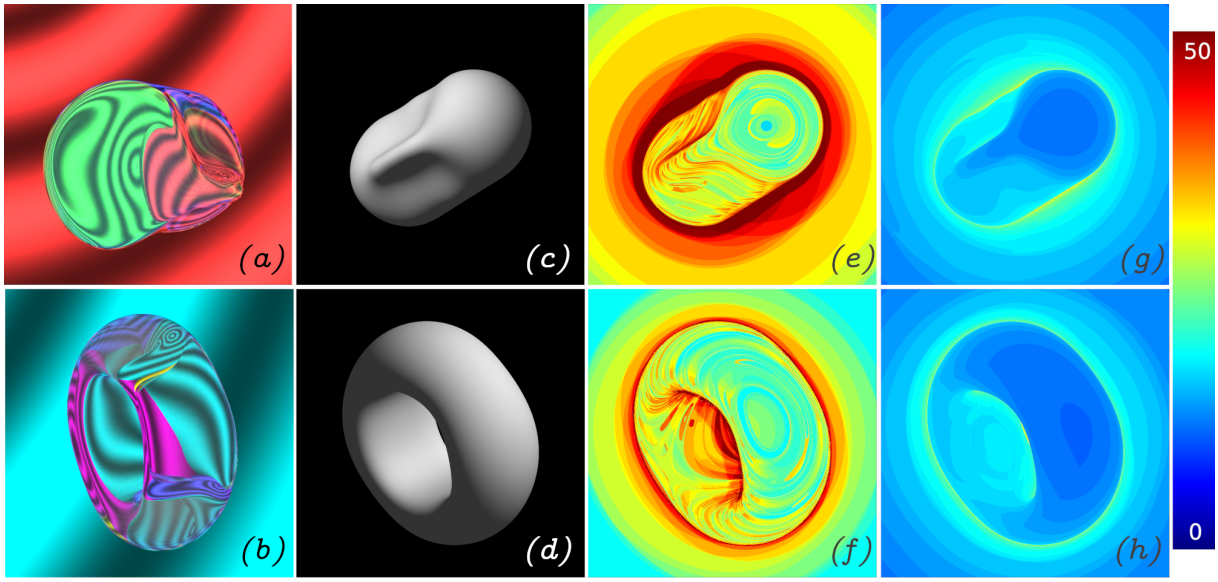


Figure 3.17: Transparent (a, b) and opaque rendering (c, d) of HRBF for 17 (top) and 8 control points (down). Opaque rendering: comparison between number of steps between Revised Affine Arithmetic (e, f) and bottom-up quadratic tracing (g, h) shows a large reduction.

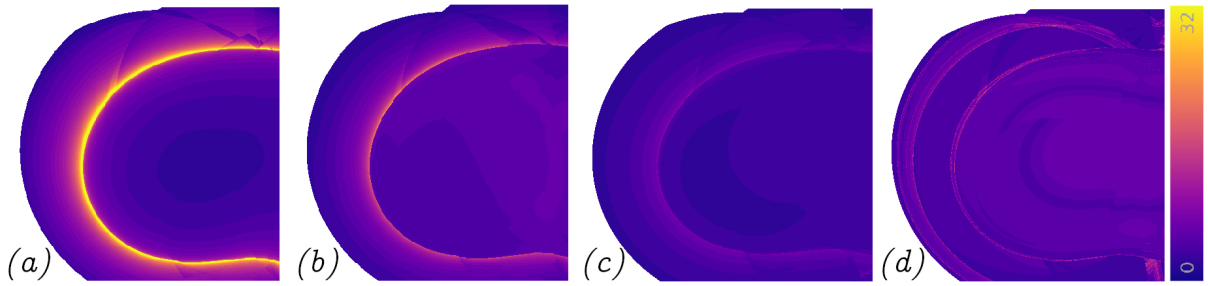


Figure 3.18: Comparison between the number of steps for molecule rendering [Bru19] (a), Segment Tracing [GGPP20] (b), quadratic tracing with Taylor quadratic inclusion function (c) and bottom-up quadratic inclusion function with validity intervals (d). Visible discontinuities are due to Gaussian kernel clipping.

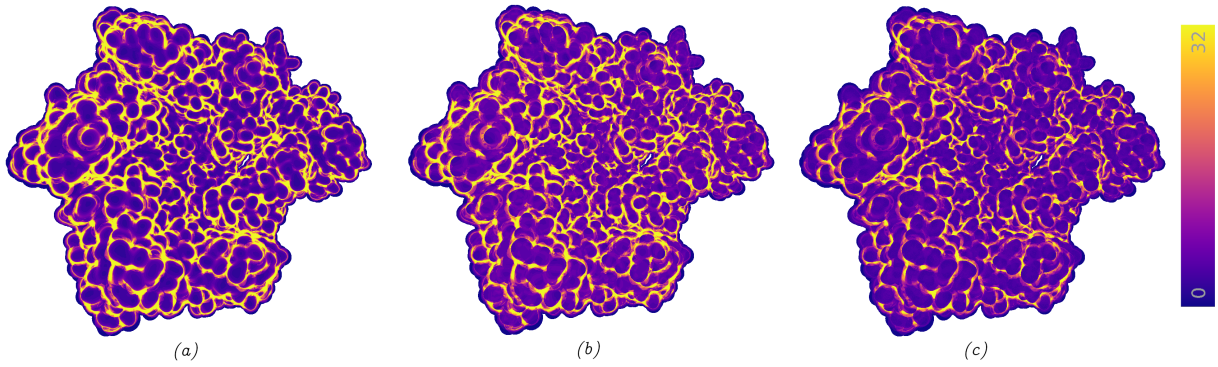


Figure 3.19: Comparison between the number of steps for (a) molecule rendering [Bru19], (b) Segment Tracing [GGPP20] and (c) quadratic tracing with local Taylor quadratic inclusion function.

Table 3.2: Run-time comparison (frame per second) for molecule example. Increasing the maximum number of marching steps does not change the performance of our methods.

Resolution 750x750	FPS		Gain
16791 atoms	Max # of steps = 32	Max # of steps = 1000	(time)
Molecule Rendering [Bru19]	22	19.7	-
Segment Tracing [GGPP20]	15	15	-46.7%
Quadratic bottom-up relaxed at t_0	25	25	12%
Quadratic bottom-up relaxed at maxima	23	23	4.4%
Quadratic bottom-up with validity	25	25	12%
Quadratic Taylor	28	28	21.4%

Table 3.3: Comparison of average calculation time and the number of steps per ray between different methods for the transparent scene in Figure 3.1 for 3305668 random rays around the bounding box. For this example, linear Taylor and quadratic Taylor inclusion functions are used as building blocks for the bottom-up inclusion function with the max operation.

$\sim$ 3M Bounding Box Rays	Avg. time per ray(μs)	Avg. # of steps	Gain (time)
Segment Tracing [GGPP20]	124.26	39.36	-
Linear Mixed	123.07	29.03	0.1%
Bottom-up Quadratic	62.491	15.90	49.7%
Quadratic Mixed	51.22	13.28	58.8%

To overcome this, we also experimented with calculating bounds by relaxing some constraints (see Figure 3.12) but did not observe major changes in terms of run-times.

3.5.2 Transparency

Transparent rendering results are shown for the scenes in Figures 3.1 and 3.20 for blobby surfaces blended with summation, and max operation for Figure 3.1 (top). Run-times are provided in Tables 3.3, 3.4, 3.5.

Transparent rendering is challenging for marching algorithms like Sphere Tracing [Har96]. After registering the first entry root, the distance bounds keep the iteration close to the surface and have convergent behaviour that needs to be managed to leave the registered root. Asymmetric bounds greatly improve transparent rendering. Small step sizes can thus be avoided while inside the object and still close to the recently recorded entry point. We saw a substantial reduction in the number of steps required to reach the surface as well as the time performance in our examples.

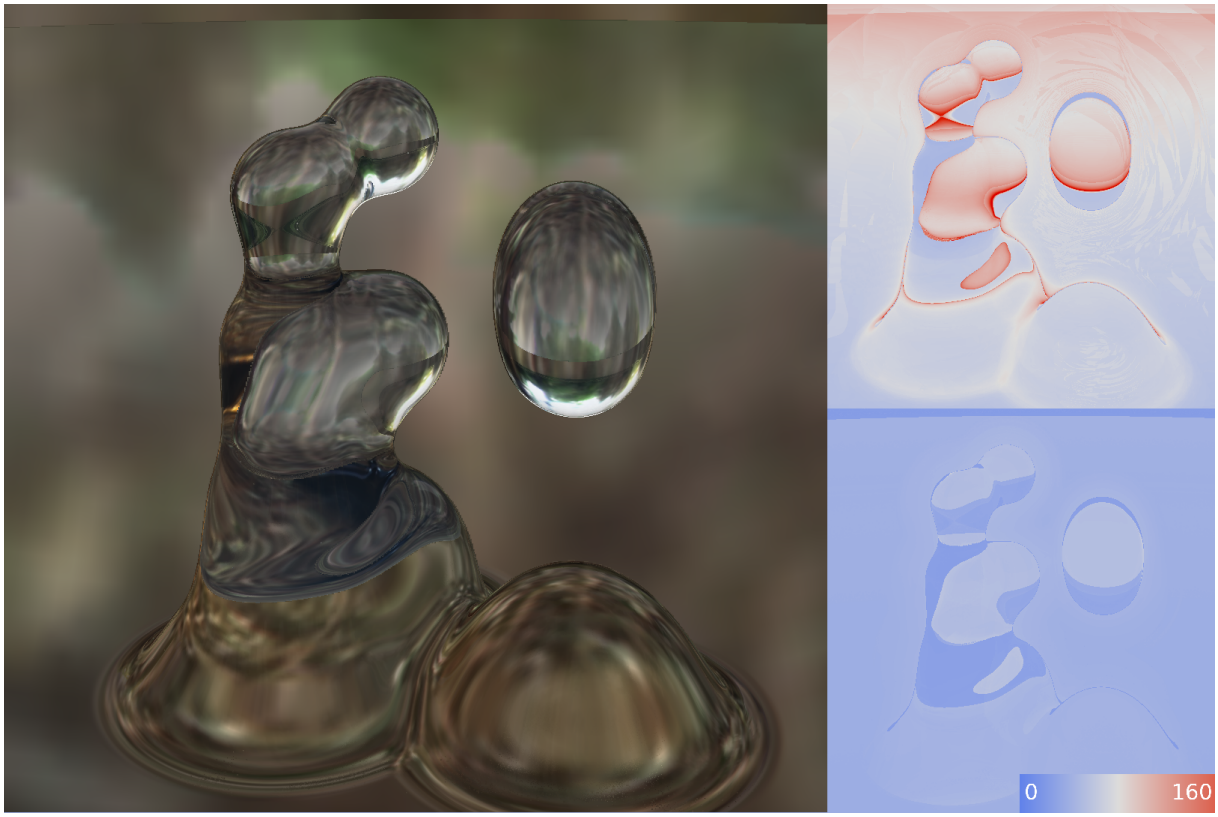


Figure 3.20: Comparison between the number of steps for Segment Tracing [GGPP20] (top) and our quadratic bottom-up inclusion function for transparent rendering with reflection and refraction.

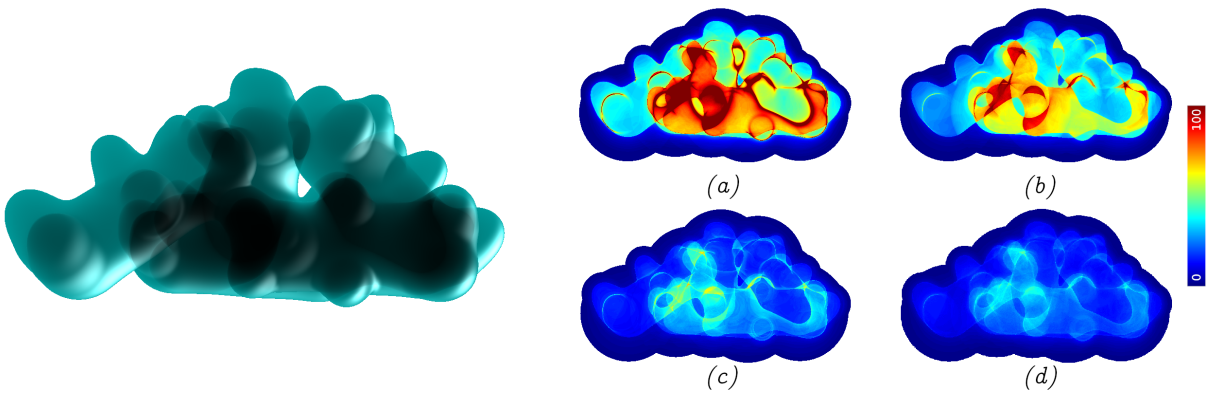


Figure 3.21: Comparison between the number of steps for the transparent scene (left) combined with summation operation. (a) Segment Tracing [GGPP20], (b) Linear Taylor inclusion function, (c) Quadratic bottom-up inclusion function, (d) Quadratic Taylor inclusion function.

Table 3.4: Comparison of average calculation time and the number of steps per ray between different methods for the transparent scene in Figure 3.21 for 3305668 random rays around the bounding box.

∼ 3M Bounding Box Rays	Avg. time per ray(μs)	Avg. # of steps	Gain (time)
Segment Tracing [GGPP20]	136.06	40.97	-
Bottom-up Linear	125.33	28.95	%7.89
Linear Taylor	107.03	28.95	%21.34
Bottom-up Quadratic	59.39	15.01	%56.35
Quadratic Taylor	44.26	12.64	%67.47

Table 3.5: GPU run-time comparison for the animated scene with refraction in Figure 3.20 average frame per second for 100 frames with bottom-up quadratic inclusion function.

Resolution 1312 x 1017	FPS	ms	Gain (time)
Segment Tracing [GGPP20]	11	89	-
Bottom-up Quadratic	13.8	72.7	19%
Quadratic Taylor	14.8	67.5	24 %

3.6 Conclusion

We have introduced an extension to the two families of ray-tracing methods, namely Lipschitz and interval methods, by creating asymmetrical marching bounds. We show that using asymmetrical linear and quadratic inclusion bounds that are exact at the starting point of the query interval provides significant improvement for ray tracing, especially when rendering transparent objects. Our bottom-up bound calculations show that we can achieve these improvements without directly calculating the bounds on the higher-order derivatives. While bottom-up bounds simplify the derivation of the new bounds by defining re-usable base building blocks, Taylor-based direct bounds are more efficient when available. We also show that the same framework can combine bottom-up and derivative-based bound computations.

Our study was focused on point-based primitives and HRBF, with summation and the set theoretic union operation defined as a maximum function. Extending the study to a more extensive set of primitives and operators is a natural follow-up. Similarly, providing tighter bounds or bounds that are less expensive to evaluate for the bottom-up approach would also be an interesting research direction.

4
Conclusion and Future Work

In conclusion, in this thesis, we have studied ray-tracing implicit surfaces in real-time. We first studied quadratic interpolation for finding roots along the ray. Then, to guarantee ray-surface intersections in a robust way we have utilized self validated numerical methods. And finally, combining the guaranteed marching methods with inclusion functions, we have produced flexible yet robust linear and quadratic marching bounds.

We have shown that it's possible to calculate distance bounds for integral surfaces in a similar way it was proposed for density fields with point primitives. We have also shown that using quadratic interpolation speeds up the root-finding for functions that do not exhibit linear behaviour, and screen space acceleration with GPU implementation provides real-time processing.

We have remarked that for transparent rendering, self validated numerical methods are advantageous because of the non-iterative procedure that does not slow down after registering the first root. We have also shown that when using self validated numerical bounds, we can use the high level information we have on the expressions where the complex calculations would create prohibitively large bounds. This way, we achieved much tighter bounds without compromising on the robustness.

Focusing on the inclusion property of the Lipschitz based marching methods, we have shown that they can be easily extended beyond the black-box marching bounds and specific improvements can be made for each problematic use-case. These use-cases include density fields where the field behaviour along rays is not linear, behavior on near grazing rays where the rays are close to tangent to the surface, and in the configurations where the Lipschitz bounds are shown to be overly conservative.

Based on the connection between Lipschitz-bound based iterative methods and Newton-Raphson iteration for root finding, we show that quadratic bounds can be marched in a similar way to linear bounds. This way, using the same processing loop, we were able to extend the linear bounds to quadratic bounds and improve the run-times and the number of field function evaluations that cause the bulk of the computational complexity, therefore, improve the speed of rendering.

The first future direction would be applying these bounding methods to a larger family of operations and optimizing the bounds for the provided operations further. This can be promising for neural implicits. As it has been suggested recently [SJ22], self-validated numerical methods

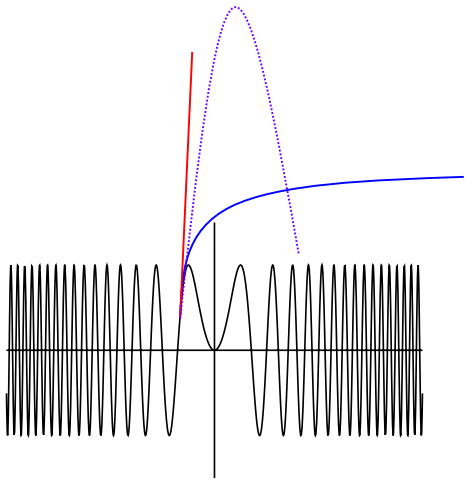


Figure 4.1: Approximation of a periodical function $sin(x^2)$ for building the upper inclusion bound. Keeping the value of the inclusion function exact at the interval starting point becomes inefficient with increasing oscillations for linear (red) and quadratic (purple) bounds. A rational quadratic bound (blue) can improve the bound quality without compromising on the fast root finding for the iteration.

such as Interval Arithmetic and Affine Arithmetic provide an efficient visualization method for generic implicit surface definitions which include neural implicits. These methods allow us to provide inclusion bounds for the field function values by evaluating the arithmetic operations, without having access to higher order information about them. For this particular purpose, the family of operations that we calculate the asymmetrical linear and quadratic bounds needs to be extended. Mainly, periodical functions such as trigonometric functions and case-specific piece-wise defined activation functions need to be studied further. For trigonometric functions, there is a trade-off between the good local approximation and the error on the overall interval. A good bounding strategy may depend on the number of oscillations. If there are a large number of small oscillations, a quadratic bound matching the tangent at the starting point would lead to very large error further along the interval (Figure 4.1).

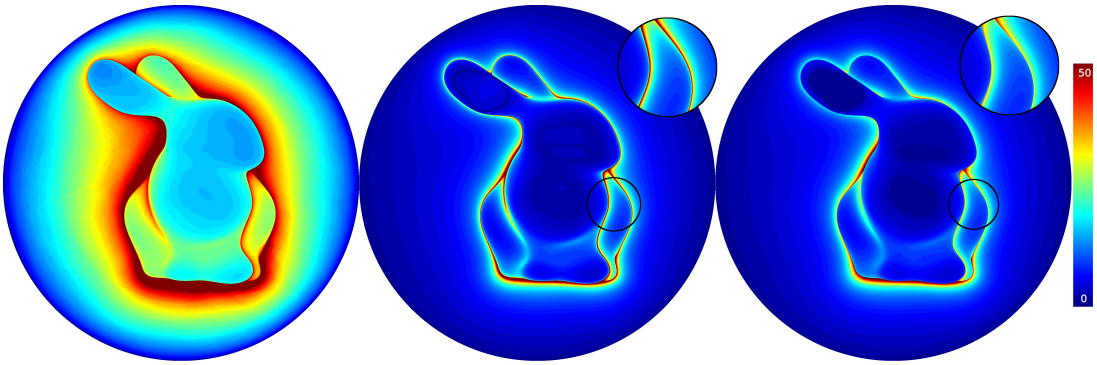


Figure 4.2: *From left to right:* Number of steps for opaque rendering with Linear Inclusion, Sphere Tracing [Har96], Sphere Tracing with fallback to linear inclusion whenever it provides a better bound. The model is a neural SDF representation with a periodical activation function [SMB*20].

In Figure 4.2, a neural SDF representation [SMB*20] is visualized using Sphere Tracing [Har96] and bottom-up linear inclusion. In the right-most figure, whenever the linear inclusion provides a better bound, it is used instead of Sphere Tracing where the bound is $L = 1$.

The surface definition using periodical activation functions limits the improvement in this case. As well as the SDF definition with the constant Lipschitz bound, allows direct Sphere Tracing, which is not the case for an arbitrary field definition. Even for this case we see a decrease in the number of steps around the grazing rays (zoomed in area in the Figure 4.2) For more general neural implicits and quasi-linear or non-periodical activation functions, we would expect more improvement with our asymmetrical bottom-up inclusion functions.

Bibliography

[AZ21] Aydinlilar M., Zanni C.: Fast ray tracing of scale-invariant integral surfaces. *Computer Graphics Forum 40*, 6 (2021), 117–134. doi:10.1111/cgf.14208.

[AZ22] Aydinlilar M., Zanni C.: Transparent rendering and slicing of integral surfaces using per-primitive Interval Arithmetic. In *Eurographics 2022 - Short Papers* (2022), The Eurographics Association. doi:10.2312/egs.20221027.

[AZ23] Aydinlilar M., Zanni C.: Forward inclusion functions for ray-tracing implicit surfaces. *Computers & Graphics (Special Section on SMI 2023) 114* (2023), 190–200. doi:https://doi.org/10.1016/j.cag.2023.05.026.

[BJ07] Bartoň M., Jüttler B.: Computing roots of polynomials by quadratic clipping. *Computer Aided Geometric Design 24*, 3 (2007), 125–141. doi:10.1016/j.cagd.2007.01.003.

[Bli82] Blinn J. F.: A generalization of algebraic surface drawing. *ACM Trans. Graph. 1*, 3 (July 1982), 235–256.

[Blo97] Bloomenthal J. (Ed.): *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., 1997.

[Bru19] Bruckner S.: Dynamic visibility-driven molecular surfaces. In *Computer Graphics Forum* (2019), vol. 38(2), Wiley Online Library, pp. 317–329.

[BS91] Bloomenthal J., Shoemake K.: Convolution surfaces. In *Proceedings SIGGRAPH '91* (1991), ACM, pp. 251–256.

[BV18] Bálint C., Valasek G.: Accelerating Sphere Tracing. In *Proceedings of the 39th Annual European Association for Computer Graphics Conference: Short Papers* (Goslar Germany, Germany, 2018), EG, Eurographics Association, pp. 29–32. URL: http://dl.acm.org/citation.cfm?id=3308470.3308480.

[Can93] Cani M.-P.: An implicit formulation for precise contact modeling between flexible solids. In *20th annual conference on Computer graphics and interactive techniques (SIGGRAPH '93))* (Anaheim, United States, 1993), ACM, ACM SIGGRAPH, pp. 313–320. doi:10.1145/166117.166157.

[CHMS00] Caprani O., Hvidegaard L., Mortensen M., Schneider T.: Robust and efficient ray intersection of implicit surfaces. *Reliable Computing 6*, 1 (2 2000), 9–21. URL: https://doi.org/10.1023/A:1009921806032, doi:10.1023/A:1009921806032.

[CNLE09] Crassin C., Neyret F., Lefebvre S., Eisemann E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, Association for Computing Machinery, p. 15–22. URL: https://doi.org/10.1145/1507149.1507152, doi:10.1145/1507149.1507152.

[Cog21] Cognitive Design Systems: Cognitive Design Systems , 2021. URL: https://www.cognitive-design-systems.com/.

[CTFZ22] Chen Z., Tagliasacchi A., Funkhouser T., Zhang H.: Neural dual contouring. *ACM Trans. Graph. 41*, 4 (jul 2022). URL: https://doi.org/10.1145/3528223.3530108, doi:10.1145/3528223.3530108.

[CZ19] Chen Z., Zhang H.: Learning implicit fields for generative shape modeling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2019).

[CZ21] Chen Z., Zhang H.: Neural marching cubes. *ACM Trans. Graph. 40*, 6 (dec 2021). URL: https://doi.org/10.1145/3478513.3480518, doi:10.1145/3478513.3480518.

[dALJ*15] de Araújo B. R., Lopes D. S., Jepp P., Jorge J. A., Wyvill B.: A survey on implicit surface polygonization. *ACM Comput. Surv. 47*, 4 (may 2015). doi:10.1145/2732197.

[dFS04] de Figueiredo L. H., Stolfi J.: Affine Arithmetic: Concepts and applications. *Numerical Algorithms 37*, 1-4 (dec 2004), 147–158. doi:10.1023/b:numa.0000049462.70970.b6.

[DOG04] Dekkers D., Overveld, van C., Golsteijn R.: Combining csg modeling with soft blending using lipschitz-based implicit surfaces. *The Visual Computer 20*, 4 (2004), 380–391. doi:10.1007/s00371-002-0198-3.

[FGW01] Fox M., Galbraith C., Wyvill B.: Efficient use of the Blobtree for rendering purposes. In *Proceedings of the International Conference on Shape Modeling & Applications* (Washington, DC, USA, 2001), SMI '01, IEEE Computer Society, pp. 306–.

[FPC10] Fryazinov O., Pasko A. A., Comninos P.: Fast reliable interrogation of procedurally defined implicit surfaces using extended revised affine arithmetic. *Computers & Graphics 34* (2010), 708–718.

[FSHZ19] Fuentes Suárez A. J., Hubert E., Zanni C.: Anisotropic convolution surfaces. *Computers and Graphics 82* (2019), 106–116. URL: https://hal.inria.fr/hal-02137325, doi:10.1016/j.cag.2019.05.018.

[GBC*13] Gourmel O., Barthe L., Cani M.-P., Wyvill B., Bernhardt A., Paulin M., Grasberger H.: A gradient-based implicit blend. *ACM Trans. Graph. 32*, 2 (Apr. 2013), 12:1–12:12. doi:http://dx.doi.org/10.1145/2451236.2451238.

[GDW*16] Grasberger H., Duprat J.-L., Wyvill B., Lalonde P., Rossignac J.: Efficient data-parallel tree-traversal for blobtrees. *Computer-Aided Design 70* (2016), 171–181. SPM 2015. doi:https://doi.org/10.1016/j.cad.2015.06.013.

[GFN11] Ganacim F., Figueiredo L. H., Nehab D.: Beam casting implicit surfaces on the GPU with Interval Arithmetic. In *2011 24th SIBGRAPI Conference on Graphics, Patterns and Images* (2011), pp. 72–77. doi:10.1109/SIBGRAPI.2011.5.

[GGP*15] Génevaux J.-D., Galin E., Peytavie A., Guérin E., Briquet C., Grosbellet F., Benes B.: Terrain modelling from feature primitives. *Computer Graphics Forum 34*, 6 (May 2015), 198–210. doi:10.1111/cgf.12530.

[GGPP20] Galin E., Guérin E., Paris A., Peytavie A.: Segment tracing using local Lipschitz bounds. *Computer Graphics Forum* (2020). URL: https://hal.archives-ouvertes.fr/hal-02507361.

[GPP*10] Gourmel O., Pajot A., Paulin M., Barthe L., Poulin P.: Fitted BVH for fast raytracing of metaballs. *Computer Graphics Forum 29*, 2 (May 2010), xxx–xxx.

[HAC03] Hornus S., Angelidis A., Cani M.-P.: Implicit modelling using subdivision curves. *Visual Comput. 19*, 2-3 (May 2003), 94–104.

[Har96] Hart J. C.: Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 10 (12 1996), 527–545. doi:10.1007/s003710050084.

[HC12] Hubert E., Cani M.-P.: Convolution surfaces based on polygonal curve skeletons. *Journal of Symbolic Computation 47*, 6 (2012), 680 – 699. doi:10.1016/j.jsc.2011.12.026.

[Hub12] Hubert E.: Convolution surfaces based on polygons for infinite and compact support kernels. *Graphical Models 74*, 1 (2012), 1 – 13. doi:10.1016/j.gmod.2011.07.001.

[JK23] Jazar K., Kry P.: Temporal set inversion for animated implicits. *ACM Trans. Graph. 42*, 4 (8 2023). doi:10.1145/3592448.

[JKDW01] Jaulin L., Kieffer M., Didrit O., Walter E.: *Applied Interval Analysis with Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer London, 2001. doi:https://doi.org/10.1007/978-1-4471-0249-6.

[JLSW02] Ju T., Losasso F., Schaefer S., Warren J.: Dual contouring of Hermite data. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2002), SIGGRAPH '02, Association for Computing Machinery, p. 339–346. doi:10.1145/566570.566586.

[JT02a] Jin X., Tai C.-L.: Analytical methods for polynomial weighted convolution surfaces with various kernels. *Computer Graphics 26*, 3 (2002), 437–447.

[JT02b] Jin X., Tai C.-L.: Convolution surfaces for arcs and quadratic curves with a varying kernel. *The Visual Computer 18*, 8 (2002), 530–546.

[JTFP01] Jin X., Tai C.-L., Feng J., Peng Q.: Convolution surfaces for line skeletons with polynomial weight distributions. *J. Graph. Tools 6*, 3 (2001), 17–28. doi:10.1080/10867651.2001.10487542.

[JTZ09] Jin X., Tai C.-L., Zhang H.: Implicit modeling from polygon soup using convolution. *Vis. Comput. 25*, 3 (Feb. 2009), 279–288. doi:10.1007/s00371-008-0267-3.

[KB89] Kalra D., Barr A. H.: Guaranteed ray intersections with implicit surfaces. *SIGGRAPH Comput. Graph. 23*, 3 (July 1989), 297–306. doi:10.1145/74334.74364.

[Kee20] Keeter M. J.: Massively parallel rendering of complex closed-form implicit surfaces. In *Proceedings of SIGGRAPH* (2020).

[KHK*09] Knoll A., Hijazi Y., Kensler A., Schott M., Hansen C., Hagen H.: Fast ray tracing of arbitrary implicit surfaces with Interval and Affine arithmetic. *Computer Graphics Forum* (2009). doi:10.1111/j.1467-8659.2008.01189.x.

[KSK*14] Keinert B., Schäfer H., Korndörfer J., Ganse U., Stamminger M.: Enhanced Sphere Tracing. In *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference* (2014), Giachetti A., (Ed.), The Eurographics Association. doi:10.2312/stag.20141233.

[LC87] Lorensen W. E., Cline H. E.: Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1987), SIGGRAPH '87, Association for Computing Machinery, p. 163–169. doi:10.1145/37401.37422.

[Lef13] Lefebvre S.: Icesl: A GPU accelerated CSG modeler and slicer. In *AEFA'13, 18th European Forum on Additive Manufacturing* (Paris, France, June 2013). URL: https://inria.hal.science/hal-00926861.

[LHL14] Lefebvre S., Hornus S., Lasram A.: Per-pixel lists for single pass A-Buffer. In *GPU Pro 5: Advanced Rendering Techniques*, Engel W., (Ed.). A K Peter / CRC Press, Mar. 2014. URL: https://hal.inria.fr/hal-01093158.

[Lin91] Lindeberg T.: *Discrete Scale-Space Theory and the Scale-Space Primal Sketch.* PhD thesis, 1991.

[LLZ*21] Liu S., Liu T., Zou Q., Wang W., Doubrovski E. L., Wang C. C.: Memory-efficient modeling and slicing of large-scale adaptive lattice structures. *Journal of Computing and Information Science in Engineering 21*, 6 (2021).

[LZLW09] Liu L., Zhang L., Lin B., Wang G.: Fast approach for computing roots of polynomials using cubic clipping. *Computer Aided Geometric Design 26*, 5 (2009), 547–559. doi:10.1016/j.cagd.2009.02.003.

[Mag21] MagicaCSG: MagicaCSG, 2021. URL: http://ephtracy.github.io/index.html?page=magicacsg.

[MCTB11] Maule M., Comba J. L. D., Torchelsen R. P., Bastos R.: A survey of raster-based transparency techniques. *Computers & Graphics 35*, 6 (2011), 1023–1034.

[MCTB12] Maule M., Comba J., Torchelsen R., Bastos R.: Memory-efficient order-independent transparency with dynamic fragment buffer. In *Proc. 25th Conference on Graphics, Patterns and Images (SIBGRAPI)* (2012), IEEE, pp. 134–141.

[MDL16] Martínez J., Dumas J., Lefebvre S.: Procedural voronoi foams for additive manufacturing. *ACM Trans. Graph. 35*, 4 (jul 2016). doi:10.1145/2897824.2925922.

[Med20] Media Molecule: Dreams, 2020. URL: http://dreams.mediamolecule.com/.

[MGV11] Macêdo I., Gois J. P., Velho L.: Hermite radial basis functions implicits. *Computer Graphics Forum 30*, 1 (2011), 27–42. doi:https://doi.org/10.1111/j.1467-8659.2010.01785.x.

[Mit90] Mitchell D. P.: Robust ray intersection with Interval Arithmetic. In *Proceedings on Graphics Interface '90* (CAN, 1990), Canadian Information Processing Society, p. 68–74.

[MON*19] Mescheder L., Oechsle M., Niemeyer M., Nowozin S., Geiger A.: Occupancy networks: Learning 3D reconstruction in function space. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)* (2019).

[Moo66] Moore R.: *Interval Analysis.* Prentice-Hall series in automatic computation. Prentice-Hall, 1966.

[MS98] McCormack J., Sherstyuk A.: Creating and rendering convolution surfaces. *Computer Graphics Forum 17*, 2 (1998), 113–121.

[MS16] Marschner S., Shirley P.: *Fundamentals of Computer Graphics, Fourth Edition*, 4th ed. A. K. Peters, Ltd., USA, 2016.

[MT06] Messine F., Touhami A.: A general reliable quadratic form: An extension of Affine Arithmetic. *Reliable Computing 12* (06 2006), 171–192. doi:10.1007/s11155-006-7217-4.

[Neu02] Neumaier A.: Taylor forms - use and limits. *Reliable Computing 2003* (2002), 9–43.

[NHK*85] Nishimura H., Hirai M., Kawai T., Kawata T., Shirakawa I., Omura K.: Object modeling by distribution function and a method of image generation. *Transactions IECE Japan 4* (1985), 718–725.

[NN94] Nishita T., Nakamae E.: A method for displaying metaballs by using Bézier clipping. *Comput. Graph. Forum 13* (08 1994), 271–280. doi:10.1111/1467-8659.1330271.

[nTo15] nTopology Inc.: nTopology, 2015. URL: https://www.ntop.com/.

[PASS95] Pasko A., Adzhiev V., Sourin A., Savchenko V.: Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer 11*, 8 (1995), 429–446.

[PFS*19] Park J. J., Florence P., Straub J., Newcombe R., Lovegrove S.: DeepSDF: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2019).

[PFV*11] Pasko A., Fryazinov O., Vilbrandt T., Fayolle P.-A., Adzhiev V.: Procedural function-based modelling of volumetric microstructures. *Graphical Models 73*, 5 (2011), 165–181. doi:10.1016/j.gmod.2011.03.001.

[PGGM09] Peytavie A., Galin E., Grosjean J., Merillou S.: Arches: a framework for modeling complex terrains. *Computer Graphics Forum 28*, 2 (2009), 457–467. doi:https://doi.org/10.1111/j.1467-8659.2009.01385.x.

[PGMG09] Peytavie A., Galin E., Merillou S., Grosjean J.: Arches: A framework for modeling complex terrains. *Computer Graphics Forum (Proceedings of Eurographics) 28*, 2 (2009), 457–467.

[PGP*19] Paris A., Galin E., Peytavie A., Guérin E., Gain J.: Terrain amplification with implicit 3D features. *ACM Trans. Graph. 38*, 5 (sep 2019). doi:10.1145/3342765.

[PH89] Perlin K., Hoffert E. M.: Hypertexture. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1989), SIGGRAPH '89, Association for Computing Machinery, p. 253–262. doi:10.1145/74333.74359.

[QJ13] Quilez I., Jeremias P.: Shadertoy, 2013. URL: https://www.shadertoy.com/.

[Rat97] Ratz D.: An optimized interval slope arithmetic and its application.

[Ric73] Ricci A.: Constructive geometry for computer graphics. *Computer Journal 16*, 2 (1973), 157–60.

[RLD*12] Reiner T., Lefebvre S., Diener L., García I., Jobard B., Dachsbacher C.: A runtime cache for interactive procedural modeling. *Computers & Graphics 36*, 5 (2012), 366–375. Shape Modeling International (SMI) Conference 2012. URL: https://www.sciencedirect.com/science/article/pii/S0097849312000702, doi:https://doi.org/10.1016/j.cag.2012.03.031.

[SAJ21] Sellán S., Aigerman N., Jacobson A.: Swept volumes via spacetime numerical continuation. *ACM Transactions on Graphics* (2021).

[Sec18] Second Order: Claybook, 2018. URL: https://www.claybookgame.com/.

[SF14] Schollmeyer A., Froehlich B.: Direct isosurface ray casting of nurbs-based isogeometric analysis. *IEEE Transactions on Visualization and Computer Graphics 20*, 9 (2014), 1227–1240.

[She99a] Sherstyuk A.: Fast ray tracing of implicit surfaces. *Comput. Graph. Forum 18* (1999), 139–147.

[She99b] Sherstyuk A.: Kernel functions in convolution surfaces: A comparative analysis. *The Visual Computer 15*, 4 (1999), 171–182.

[SJ22] Sharp N., Jacobson A.: Spelunking the deep: Guaranteed queries on general neural implicit surfaces via range analysis. *ACM Trans. Graph. 41*, 4 (jul 2022). doi:10.1145/3528223.3530155.

[SJNJ19] Seyb D., Jacobson A., Nowrouzezahrai D., Jarosz W.: Non-linear Sphere Tracing for rendering deformed signed distance fields. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) 38*, 6 (Nov. 2019). doi:10.1145/3355089.3356502.

[SMB*20] Sitzmann V., Martel J. N., Bergman A. W., Lindell D. B., Wetzstein G.: Implicit neural representations with periodic activation functions. In *Proc. NeurIPS* (2020).

[SMH*23] Shen T., Munkberg J., Hasselgren J., Yin K., Wang Z., Chen W., Gojcic Z., Fidler S., Sharp N., Gao J.: Flexible isosurface extraction for gradient-based mesh optimization. *ACM Trans. Graph. 42*, 4 (jul 2023). URL: https://doi.org/10.1145/3592430, doi:10.1145/3592430.

[Sta] Stanford University Computer Graphics Laboratory: Stanford Bunny. URL: http://graphics.stanford.edu/data/3Dscanrep/.

[SWBG06] Sigg C., Weyrich T., Botsch M., Gross M.: GPU-based ray-casting of quadratic surfaces. In *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2006), SPBG'06, Eurographics Association, pp. 59–65. doi:10.2312/SPBG/SPBG06/059-065.

[SWG05] Schmidt R., Wyvill B., Galin E.: Interactive implicit modeling with hierarchical spatial caching. In *Proceedings of the International Conference on Shape Modeling and Applications 2005* (Washington, DC, USA, 2005), SMI '05, IEEE Computer Society, pp. 104–113. doi:10.1109/SMI.2005.25.

[SWR18] Singh J. M., Wasnik P., Ramachandra R.: Hessian-based robust ray-tracing of implicit surfaces on GPU. In *SIGGRAPH Asia 2018 Technical Briefs* (New York, NY, USA, 2018), SA '18, ACM, pp. 16:1–16:4. doi:10.1145/3283254.3283287.

[Thi11] Thibieroz N.: Order-independent transparency using per-pixel linked lists. In *GPU Pro 2*, Engel W., (Ed.). A K Peters, 2011, pp. 409–431.

[VBG*13] Vaillant R., Barthe L., Guennebaud G., Cani M.-P., Rohmer D., Wyvill B., Gourmel O., Paulin M.: Implicit skinning: real-time skin deformation with contact modeling. *ACM Trans. Graph. 32*, 4 (July 2013), 125:1–125:12. doi:10.1145/2461912.2461960.

[Wen05] Wendland H.: *Scattered data approximation*. Cambridge University Press, 2005.

[WGG97] Wyvill B., Galin E., Guy A.: The Blob Tree, warping, blending and boolean operations in an implicit surface modeling system. *University of Calgary technical report* (July 1997).

[WGG99] Wyvill B., Guy A., Galin E.: Extending the CSG tree. Warping, blending and boolean operations in an implicit surface modeling system. *Comput. Graph. Forum 18* (06 1999), 149–158. doi:10.1111/1467-8659.00365.

[WMW86] Wyvill G., McPheeters C., Wyvill B.: Data structure for soft objects. *The Visual Computer - VC 2* (08 1986), 227–234. doi:10.1007/BF01900346.

[WO97] Wyvill B., Overveld K. V.: Warping as a modelling tool for CSG/implicit models. In *Proceedings of the 1997 International Conference on Shape Modeling and Applications (SMI '97)* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 205–214.

[Wom22] Womp3D Inc.: Womp, 2022. URL: https://womp.com/.

[WT90] Wyvill G., Trotman A.: Ray-tracing soft objects. In *CG International '90* (1990), Springer Japan, pp. 469–476. doi:10.1007/978-4-431-68123-6_27.

[ZBQC13] Zanni C., Bernhardt A., Quiblier M., Cani M.-P.: SCALe-invariant integral surfaces. *Computer Graphics Forum 32* (2013).

[ZGC15] Zanni C., Gleicher M., Cani M.-P.: N-ary implicit blends with topology control. *Computers & Graphics 46* (2015), 1–13.